The Action Provider Framework

The Power to Know.

05/03/05 Revision 1

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2005. *The Action Provider Framework*. Cary, NC: SAS Institute Inc.

The Action Provider Framework

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, May 2005

SAS Publishing provides a complete selection of books and electronic products to help customers use SAS software to its fullest potential. For more information about our e-books, e-learning products, CDs, and hard-copy books, visit the SAS Publishing Web site at **support.sas.com/pubs** or call 1-800-727-3228.

05/03/05 Revision 1

The Action Provider Framework

Flavio Ubaldini, SAS Institute, Heidelberg, Germany,
Neil Bell, SAS Institute Inc., Cary, NC

ABSTRACT

Introduced with SAS® AppDev Studio™ Version 3.0, the Action Provider Framework (APF) is a set of Java classes which give application developers the ability to easily customize the functionality and appearance of the Actions presented by both Swing and Java Server Pages (JSP) viewer components.

The purpose of this document is to provide an introduction to the APF, including:

- An introduction to the key pieces of the framework.
- An overview of the framework's three operational phases.
- Examples: how to remove or override a component's default Actions and how to register a new Action for a component.
- A comparison of the APF to the Struts application framework.

INTRODUCTION

When writing Web applications with server-side Java (JSP files and Servlets) or client-side java (applets) the default functionality surfaced by a component viewer may not meet your requirements to one degree or another.

Traditionally, your only course of action would be to begin the time consuming task of extending the existing component, or even worse, writing a new component. Further, if multiple types of component viewers in your application require customizing, each type often implement their functionality in very different ways. This lack of consistency makes implementation of the new or modified components an ad-hoc affair. The solution that SAS offers to solve this problem is the Action Provider Framework (APF).

The APF makes customizing component Actions simpler for application developers by providing the following:

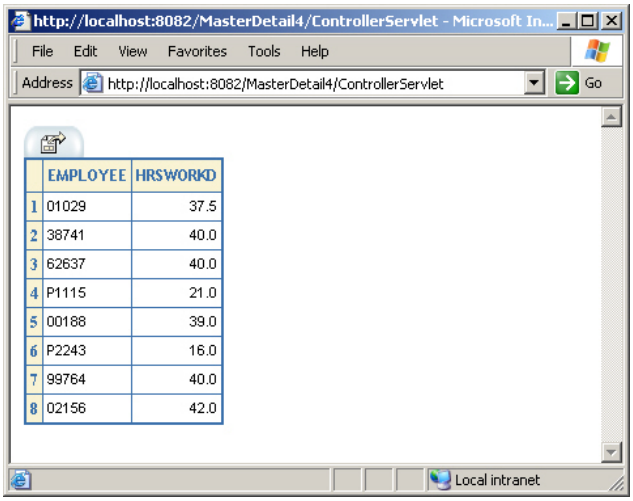
- A common template for Actions in the form of a few Action classes.
- Methods to acquire and modify a component's default Actions.
- Methods to register new Actions for a component.
- Methods to acquire and modify the data structures that determine how Actions are ordered and grouped when presented by the viewer component.

The general concepts outlined in this document are applicable in both Swing and JSP/Servlet environments; however, there are differences in some of the details. Important differences will be explained in each of the applicable topics discussed later. However, for the sake of keeping things as simple as possible in an introductory level document, this paper will use only JSP/Servlet examples.

INTRODUCTION TO THE MAIN EXAMPLE

This document will present many APF concepts in the context of a simple Web application project that may be built using webAF. The steps required to build the application will be presented in detail later, but for now, lets just take a look at a snapshot from the initial phase of the example.

Figure 1



The image shows a default TableViewComposite component displaying data from a JDBC data source. The examples in the paper will demonstrate how to change some of this component's default functionality and appearance. The last example will show how to add new functionality to the data cells of the component.

Before we get to the examples, however, an overview of the various pieces of the APF and how they relate to each other is needed.

THE KEY PIECES OF THE FRAMEWORK

The primary types of objects in the APF are:

- Actions
- Commands
- Viewer components
- Action Providers
- Viewer-specific support classes

Each type is described in the sections that follow.

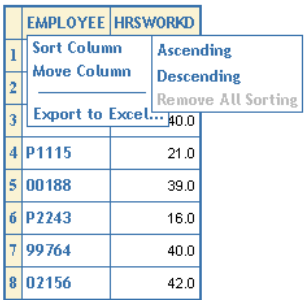
ACTIONS

In general, actions may be thought of as the elements presented by viewers that enable users to perform distinct functions. Some examples of actions:

- sorting or filtering the data in a relational table view component
- viewing the contents of a folder in a file system
- navigating between application views or states.

In the APF, Actions are Objects that are acquired by viewer components from the framework. For example, the TableView component in our sample Web application acquires and presents a number of Actions for each of its column header cells.

Figure 2



The figure above shows some of the Actions acquired by the TableView for its EMPLOYEE column header cell. They include:

- *Ascending*, *Descending* and *Remove All Sorting* which all perform a specific function related to sorting on the viewer's model.
- *Employee*, *Sort Column* and *Move Column* which function only as labels and submenu choices.
- *Export to Excel* which launches a separate dialog for specifying various export options.

If you clicked on the other column header, HRSWORKD, you would see the same types of Actions in its menus. Each set of Actions is acquired separately by the TableView for use within specific header cells.

All *acquired* Actions actually begin as copies of *registered* instances. For example, the *Ascending* Actions that are present beneath the EMPLOYEE and HRSWORKD columns were originally created as copies of the same registered instance.

If necessary, these copies are then modified by the framework so that they work for the specific area they will be presented by the viewer. For example, an Action acquired for the EMPLOYEE column might have a dynamic *columnIndex* attribute with value of 0 while the HRSWORKD version of the Action would have a value of 1 for that attribute.

Dynamic attributes often play a role when an Action is being performed. When a user selects an Action, the APF can pass these attributes to a separate command object which is responsible for carrying out the operation. For example, the task of sorting the model gets delegated to a sort command that needs the value of the *columnIndex* attribute to know which column to sort.

COMMANDS

Commands are the Objects that contain the logic to perform an action. A command may be associated with an Action by simply setting the Action's command attribute. As a result, developers may easily change the behavior of a default Action by simply replacing the command object with an overridden or entirely new command object.

A single instance of a command object can be shared between multiple actions. For example, the sort ascending Actions in each column of the TableView component share the same instance of a sort command.

This is possible because the framework sets the dynamic, area-dependent information (such as the *columnIndex* attribute described above) on Actions, not commands, as they are acquired by a viewer component. Those dynamic attributes are set on commands only after a user has activated an action to perform some function.

VIEWER COMPONENTS

Viewer components are the objects that are responsible for acquiring Actions from the APF and then presenting them to the user. They query the APF with information describing the part of their view for which they want Actions. For example, the TableView component might specify the following as part of their query:

- General area: *areaType* equal to COLUMN_HEADER_AREA
- Specific area: *columnIndex* equal to 1.

The return value for a single query for Actions is a list containing all the Actions, separators and, possibly, other nested lists that satisfy the parameters of that particular query. The viewer then presents the contents of the list in the same order and structure as the returned list.

Examples found later in this document will demonstrate how to control the order and grouping of Actions via the com.sas.actionprovider.ActionOrderList object.

You will find a list of Viewer Components that use the APF in the documentation of the com.sas.actionprovider package:
http://support.sas.com/rnd/gendoc/bi/api/Components/com/sas/actionprovider/package-summary.html

ACTION PROVIDERS

The Action Provider plays the most central role of the APF. In addition to being the object that viewer components query for Actions, developers also interact directly with this object whenever they want to:

- Acquire a default Action.
- Register a new Action.
- Affect Action visibility, order and structure.

There are two different types of Action Providers:

- com.sas.actionprovider.HttpActionProvider
- com.sas.actionprovider.SwingActionProvider

Both serve similar purposes in their respective environments, except that the HttpActionProvider also contributes to the execution of Action commands. More details on this subject can be found in the *Execution Phase* section below.

In the JSP/Servlet environment, developers must instantiate an HttpActionProvider and set it on the viewer component. One HttpActionProvider may be shared by different viewer components. Swing component viewers typically create their own

SwingActionProvider if one is not given to them. However, they also can be forced to share the same ActionProvider via the setActionProvider() method.

Sharing an Action Provider among viewers of different types is possible because all interactions with an Action Provider are actually delegated to viewer-specific APF support classes. The next section contains more details on support classes.

APF SUPPORT CLASSES

APF support classes do the majority of work related to the registration and acquisition of individual Actions. An Action Provider instantiates these objects and, as mentioned above, delegates many of its methods to the appropriate support class. They should never be instantiated by the application developer.

There are different support classes for different viewer/model relationships. For example, there is a relational TableViewSupport class just for the TableView component and its relational model.

Each type of support class is responsible for defining the general areas to which Actions may be registered. The support class declares a unique *areaType* key for every area it defines. For example, on the com.sas.actionprovider.HttpTableViewSupport class:

```
HttpTableViewSupport.COLUMN_HEADER_AREA
```

Secondly, the support class is responsible for defining which default types of Actions are registered to the various *areaTypes*. For each type of Action, the support class declares a unique *actionType* key. For example:

```
HttpTableViewSupport.SORT_COLUMN_ASCENDING_ACTION
```

Most efforts to customize Actions for a particular viewer require that the developer use these types of keys which are documented on each support class. For example, the call to acquire the registered version of the default sort ascending Action for a relational TableView component would be:

```
actionProvider.getDefaultAction(
      ActionProviderSupportTypes.TABLEVIEW_SUPPORT,
      HttpTableViewSupport.COLUMN_HEADER_AREA,
      HttpTableViewSupport.SORT_COLUMN_ASCENDING_ACTION);
```

The first argument is an *actionSupportType* key that identifies the support class. The keys of all APF support classes may be found in the documentation of the ActionProviderSupportTypes class located here:
http://support.sas.com/rnd/gendoc/bi/api/Components/com/sas/actionprovider/support/ActionProviderSupportTypes.html

The documentation of the com.sas.actionprovider package details the names and actionSupportType keys of support classes.
http://support.sas.com/rnd/gendoc/bi/api/Components/com/sas/actionprovider/package-summary.html

In addition to defining the areaTypes and actionTypes that it supports, the class is responsible for the following:

- Creating the default Action instances and registering them to the appropriate areaTypes.
- Provide the logic for setting the dynamic attributes on Actions as they are being acquired by a viewer.

OPERATIONAL FLOW

There are three primary operational phases of the APF that occur in the following order:

1. Action registration
2. Action acquisition
3. Action / Command execution

THE ACTION REGISTRATION PHASE

Default Actions are registered by the support class when it is first instantiated by an ActionProvider. The framework ensures this happens prior to the first rendering of the applicable viewer component.

New Actions may also be registered at any time prior to the first rendering of the viewer component via a call to the ActionProvider's setAction() method. Every new Action must be registered with a separate call to the setAction() method. However, each call need only occur once for the lifetime of the Action Provider and the viewer component that uses it.

THE ACTION ACQUISITION PHASE

In a Swing environment, this phase begins when the component viewer receives an event indicating the user is trying to bring up a pop menu of available Actions. Often this event occurs when the user clicks the right mouse button in a specific area of the view.

In a JSP/Servlet environment, this phase begins when the component viewer's write() method is called. The big difference is that a viewer in this environment has to query for all Actions in all visible areas each time it renders itself.

For example, a relational TableView does a separate query for each cell in the column header area and each cell in the data cell area. The Swing viewer queries for Actions on an as-needed basis when the user clicks the right mouse button in a specific area (cell).

In both the Swing and the JSP/Servlet environments, all queries for Actions are accomplished via a viewer component's call to the Action Provider's getActions() method. First, the method gets delegated to the appropriate support class where the following steps occur:

1. Determine the set of Actions that are registered for the *general* areaType as specified in the viewer's query.
2. For each Action, determine if it supported for the *specific* area requested in the query. For example, a TableView's MoveColumnLeftAction would not be supported if the query was for a column in the first position.
3. For each Action that is supported, make a copy of the registered version of the Action and set all of its dynamic attribute values.

4. Add each supported Action to the list that will be returned to the viewer for this query.

In the JSP/Servlet environment, Actions do not persist after the viewer component has sent their HTML representations to the client. Therefore, the HttpActionProvider must store their commands and, possibly, some of their dynamic attributes so that they may be executed later when a user chooses to perform an action. To facilitate command lookup during the execution phase, the HttpActionProvider generates a unique CMDID identifier and sets it on each Action that has a command object.

THE EXECUTION PHASE

The Swing execution phase begins when a user chooses an item out of the Action pop menu and the viewer calls the SwingAction's actionPerformed() method. This method transfers all the dynamic attributes on the Action to the command and then calls the command's execute() method.

The JSP/Servlet execution phase begins when a user selects an Action presented by the viewer component. The HTML tag used to present the action will have a *href* attribute value (URL) containing all the information needed to complete the execution phase.

1. The base of the URL will point to the JSP/Servlet to which the client will send an HttpServletRequest. That JSP/Servlet is responsible for calling the HttpActionProvider's executeCommand() method.
2. The executeCommand() method is passed the request as an argument and its parameters will include the CMDID and, possibly, dynamic attributes for the command.
3. The executeCommand() method looks up a command based on the CMDID parameter.
4. The method then does introspection on the command to get its attribute names and see if any request parameters match. If any match, the method applies them to the command.
5. The method then applies any dynamic command attributes it may have *stored* for the command.
6. The method calls the command's *execute*() method.

With the previous step the execution phase is over. Typically, however, the JSP/Servlet which called the executeCommand() method on the ActionProvider will also need to make sure that control flows back to some presentation JSP where the acquisition phase will start all over again.

BUILDING THE MAIN EXAMPLE

Each of the following examples will be presented in the context of a simple Web application that may be built using the New Project Wizard in webAF. Before getting to the Wizard, create the data sets that the application will need by executing the following code in a SAS session.

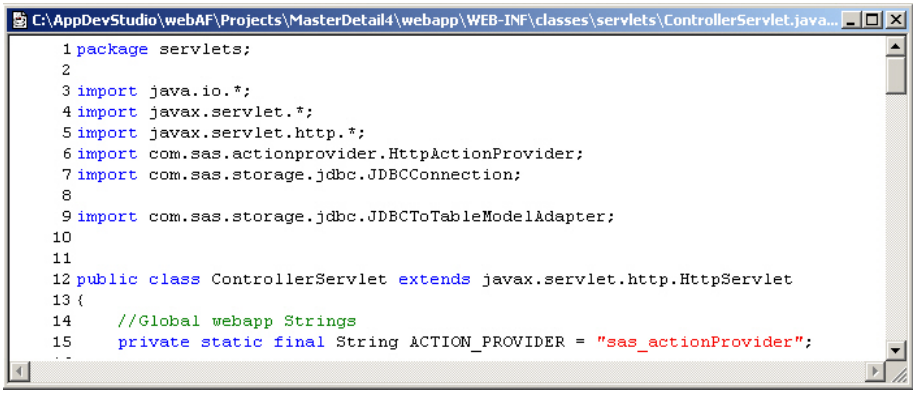
```
data sasuser.timecrd(label='required data only');
  length employee $ 5;
  input employee hrsworkd;
  datalines;
01029 37.5
38741 40
62637 40
P1115 21
00188 39
P2243 16
99764 40
02156 42
;
proc datasets library=sasuser nolist;
  modify timecrd;
    ic create pmk_id = primary key(employee);
  run;
run;
data sasuser.empdata;
  length employee $ 5;
  input employee $ dept $ jobtitle $ payincr sex $ age state $;
  cards;
62637 MKT Marketingrep 4.6  F 42 NY
01029 ISD Technician   2.25 M 39 NC
38741 TS  HelpDesk     .    F 26 NC
P1115 SLS Salesrep     2.2  M 30 GA
00188 MKT Marketingrep 0.5  M 33 TX
P2243 QA  TechnicalSt  .    F 21 NC
99764 TS  Callbackman  0.25 M 25 NC
02156 EXE Executive    10.3 F 48 NC
;
proc datasets library=sasuser nolist;
  modify empdata;
    ic create forkey = foreign key(employee) references timecrd;
  run;
run;
```

Now you are ready to start building the web application with webAF. Follow these steps:

1. Select **File→New** and select the Projects tab if it not already visible.
   1.1. In the list of project types on the left, select **Web Application Project**.
   1.2. In the **Project name** text entry field, enter *MasterDetailExample.*
   1.3. Click **OK** to start the WebApp Project Wizard.
2. Accept the defaults as you go through the WebApp Wizard until you reach step four. There, select **Examples** in the radio box titled *Display list for*. Choose **JDBC TableView Servlet** from the list box titled *Type of initial content*.
3. Continue accepting defaults as you complete the WebApp Wizard.

After the WebApp Project Wizard closes, webAF will create the directories and content files based on the selections you made. One of those files is `JDBCTableViewExampleControllerServlet.java` shown in Figure 3 below.

Figure 3

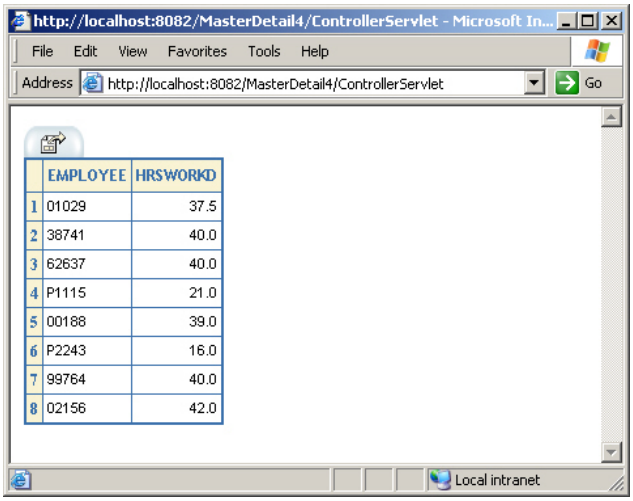


This file contains the code to connect to the JDBC data along with the application control code to process any user interaction and forward to the `index.jsp` to display the results. You need to do just a few more things to get this example working:

1. Around line 92 of the JDBCTableViewExampleControllerServlet, you will need to specify which JDBC data source to use where it has `ENTER_QUERY_STRING_HERE`. Replace that line with:
   `String jdbcQuery = "select * from sasuser.timecrd";`
2. Select **Build→Compile File** to compile the servlet.
3. Start the IOM Spawner by selecting **Start→Programs→SAS AppDev Studio→Services→SAS V9.1→Start SAS V9.1 IOM Spawner**.
4. Start the Java Web Server by selecting **Tools→Services→Start Java Web Server**.
5. Select **Build→Execute** to bring up a browser to execute your *MasterDetail* application.

You should see results similar to the following:

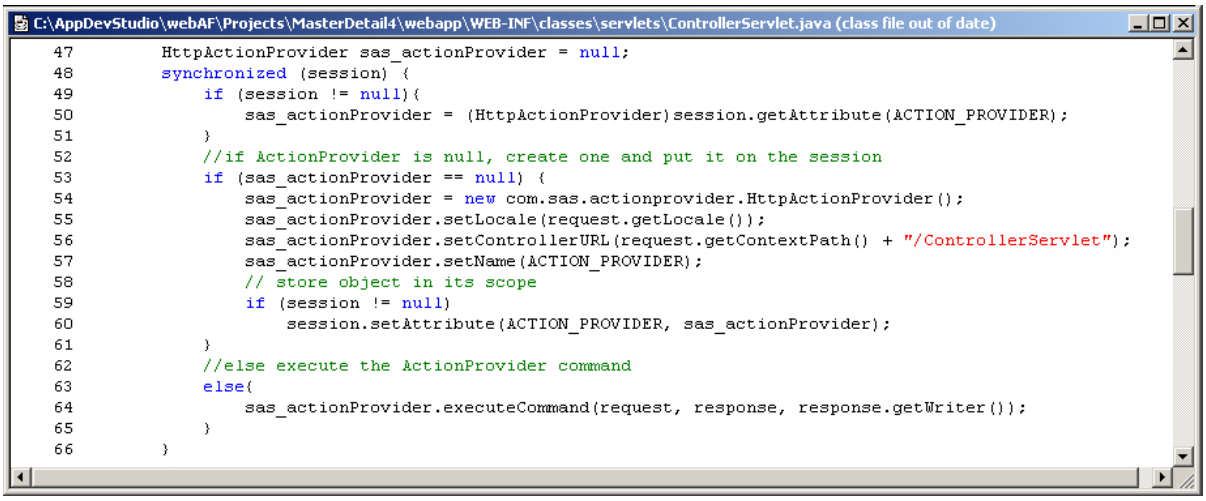
Figure 4



	EMPLOYEE	HRSWORKD
1	01029	37.5
2	38741	40.0
3	62637	40.0
4	P1115	21.0
5	00188	39.0
6	P2243	16.0
7	99764	40.0
8	02156	42.0

There are certain initial tasks that must be accomplished to have an ActionProvider-dependent viewer work properly. In `JDBCTableViewExampleControllerServlet.java` there is a section of code that demonstrates a few of these things.

Figure 5



They include:

- How to instantiate an HttpActionProvider and place it on the session.
- How to set the HttpActionProvider's *controllerURL* attribute so that, during the execution phase, control is forwarded to a JSP/Servlet that has code for executing commands on the HttpActionProvider.
- How to make the call on the HttpActionProvider that executes commands.

The remaining task is assigning the Action Provider to the viewer component. In this example, that is currently being accomplished via the *actionProvider* attribute on the TableViewComposite tag in the index.jsp file that webAF created.

Figure 6

Note that the TableViewComposite bean wrapped by this tag is not actually an APF viewer component because it does not, itself, query for any Actions. However, as a composite, it may have sub-components that are APF viewers and do need an Action Provider. They include the TableView, MenuBar and NavigationBar Scrolling elements.

As a convenience, you may set the Action Provider at the composite level and it will assign it to the proper sub-components.

In the browser, if you navigate down through one of the column header menus and hover over the *Ascending* action, you will see that the URL looks something like this:

```
      http://localhost:8082/MasterDetail/JDBCTableViewExample?CMDID=sas_
TableView1_JDBCTableViewExample_tv_c1_SCA&APNAME=sas_actionProvider_JDBC
TableViewExample
```

Once the action is activated, the client puts the CMDID and APNAME parameters on a request object and sends it to the JDBCTableViewExampleControllerServlet which is specified in the base part of the URL.

The JDBCTableViewExampleControllerServlet calls the HttpActionProvider's executeCommand() method and that method will lookup the sort ascending command based on the value of CMDID, apply any dynamic attributes that is has stored for that command and then call the command's execute() method.

EXTENDING THE MAIN EXAMPLE

Now that you have an application with a default TableView component working, it is time to start making some customizations. The next couple of examples demonstrate how to affect Action visibility and order via interactions with an ActionOrderList.

CUSTOMIZING THE VISIBILITY OF ACTIONS

The ActionOrderList is a mechanism for defining the order of Actions and SEPARATORS as well as nested lists containing those items. By modifying or replacing an ActionOrderList, a user may customize the order and structure of those items that are returned to the viewer component during the acquisition phase. Further, a user is able to hide unwanted Actions, SEPARATORS or sub-lists by removing their respective entries from the appropriate ActionOrderList.

There is a default ActionOrderList for each *areaType* that an APF support class defines. This list is used for all viewers that have the same Action Provider. However, an ActionOrderList for a specific viewer instance may also be registered with the framework when required.

This example will demonstrate how to hide the *ExportToExcel* Action, the *SEPARATOR* and all the Actions related to *Move* operations. Add the following import statements to the existing import section in `JDBCTableViewExampleControllerServlet.java`.

```
import com.sas.actionprovider.ActionOrderList;
import com.sas.actionprovider.ActionList;
import com.sas.actionprovider.HttpAction;
import com.sas.actionprovider.support.ActionProviderSupportTypes;
import com.sas.actionprovider.support.tableview.HttpTableViewSupport;
import com.sas.servlet.tbeans.tableview.html.TableView;
import com.sas.servlet.tbeans.tableview.html.TableViewComposite;
```

All the example code below should be added to `JDBCTableViewExampleControllerServlet.java` within the block of code shown here:

```
if (adapter == null){
      try{
            ...
      }
      catch(Exception e){
            throw new RuntimeException(e);
      }

      /* Add all example code here */
}
```

Example code:

```
TableViewComposite sas_TableView1 = null;
if (session != null){
      sas_TableView1 =
      (TableViewComposite)session.getAttribute("sas_TableView1_JDBCTable
      ViewExample");
}
if (sas_TableView1 == null){
      sas_TableView1 = new TableViewComposite();
      sas_TableView1.setModel(adapter);
      sas_TableView1.setActionProvider(sas_actionProvider);
      session.setAttribute("sas_TableView1_JDBCTableViewExample",
      sas_TableView1);

      /* Get the TableView sub-component from the TableViewComposite */
      TableView table =
      (TableView)sas_TableView1.getComponent(sas_TableView1.TABLEVIEW_TA
      BLEDATA);

      /* Get the table's ActionOrderList for the COLUMN_HEADER_AREA */
      ActionOrderList columnHeaderList =
      sas_actionProvider.getActionOrderList(
            ActionProviderSupportTypes.TABLEVIEW_SUPPORT,
            table,
            HttpTableViewSupport.COLUMN_HEADER_AREA);

      /* Get the sub-list for the first drop down menu */
      ActionOrderList actionSubList
      =(ActionOrderList)columnHeaderList.get(0);

      /* Hide the Export Action and the SEPARATOR */
      actionSubList.remove(HttpTableViewSupport.EXPORT_TO_ACTION);
      actionSubList.remove(ActionList.SEPARATOR);

      /* Hide the entire sublist of move-type Actions */
      actionSubList.remove(1);
}
```

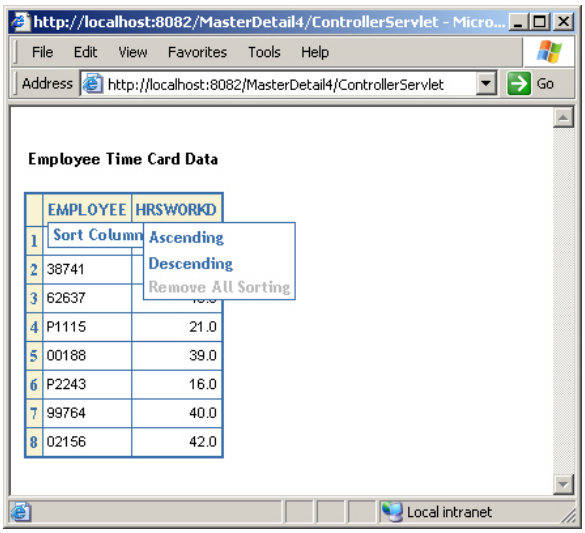
Since the code that customizes the ActionOrderList needs the instance of TableView component (*table*), we also need to add code that sets up the TableViewComposite bean in this file so that we can get a handle to this object.

Consequently, we no longer need the TableViewComposite tag in `index.jsp` to create the bean and apply some of the attributes. This is accomplished by using the *ref* attribute instead of the *id* attribute. We should also add a title for the TableViewComposite. Replace the tag in that file with following:

```
<sas:TableViewComposite ref="sas_TableView1_JDBCTableViewExample"
scope="session">
      <sas:TableTitle text="Employee Time Card Data" />
</sas:TableViewComposite>
```

After adding these customizations, choose **Build→Rebuild All** to compile your changes. Then select **Build→Execute in browser**. You should see the following in your browser.

Figure 7



CHANGING THE ORDER OF ACTIONS

The code below will change the order of the Actions in the sorting submenu such that *Remove All Sorting* is at the top. Add it right after the last line:

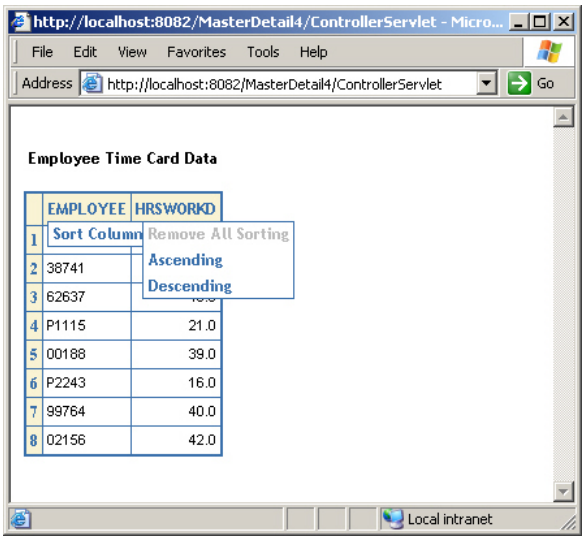
```
actionSubList.remove(1);
```

that was added to JDBCTableViewExampleControllerServlet.java in the previous example.

```
/* Get sorting sublist and re-order so that ClearSort is first */
ActionOrderList sortingSubList = (ActionOrderList)actionSubList.get(0);
sortingSubList.clear();
sortingSubList.add(HttpTableViewSupport.CLEAR_SORT_ACTION);
sortingSubList.add(HttpTableViewSupport.SORT_COLUMN_ASCENDING_ACTION);
sortingSubList.add(HttpTableViewSupport.SORT_COLUMN_DESCENDING_ACTION);
```

Compile these changes and execute in your browser. You will see the following:

Figure 8



OVERRIDING DEFAULT ACTION ATTRIBUTES

The code below demonstrates how to obtain a default Action and override its attributes to change its displayed text and give it tool tip text. Add this code right after the line:

```
sortingSubList.add(HttpTableViewSupport.SORT_COLUMN_DESCENDING_ACT
ION);
```

that was added to `JDBCTableViewExampleControllerServlet.java` in the previous example.

```
/* Get the registered version of the sort ascending Action */
HttpAction sortAscendingAction =
(HttpAction)sas_actionProvider.getDefaultAction(
      ActionProviderSupportTypes.TABLEVIEW_SUPPORT,
      HttpTableViewSupport.COLUMN_HEADER_AREA,
      HttpTableViewSupport.SORT_COLUMN_ASCENDING_ACTION );

/* Change the Action's text from "Ascending" to "Sort ascending" */
sortAscendingAction.putValue( HttpAction.NAME, "Sort ascending");

/* Give the Action tool-tip text. */
sortAscendingAction.putValue( HttpAction.SHORT_DESCRIPTION, "Sort in
Ascending Order");
```

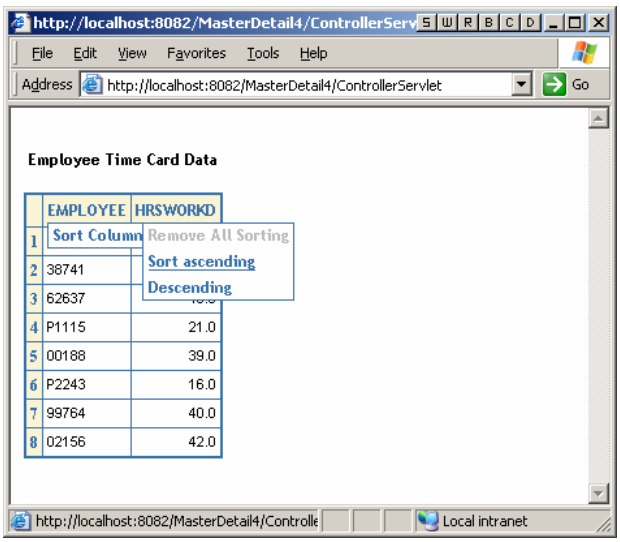
Note that the Action acquired here is the default *registered* instance. Changes made to it will affect all versions of this *actionType* (`SORT_COLUMN_ASCENDING_ACTION`) acquired by all TableView components using this ActionProvider.

If you want to make changes that affect only the Actions acquired by a specific instance of a viewer, you would register you own *copy* of the default Action instance via the Action Provider's setAction() method. You may acquire a *copy* of any default Action via the ActionProvider's newActionInstance() method. Refer to the next section on

Registering New Actions for more information on the setAction() method.

Compile these changes and execute in your browser. You will see the following:

Figure 9

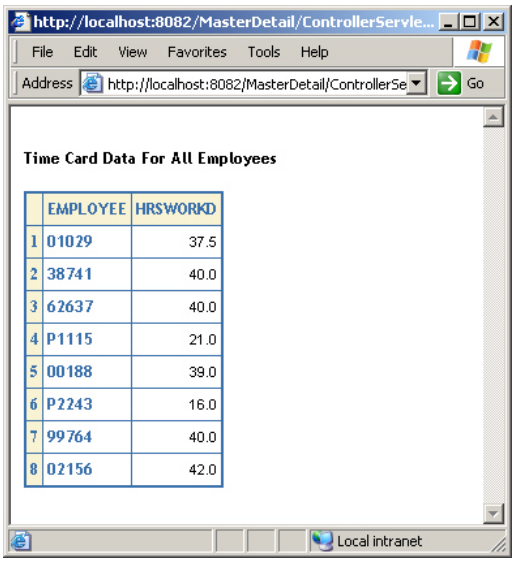


You will have to hover over *Sort ascending* in your browser to see the new tool tip text.

REGISTERING NEW ACTIONS

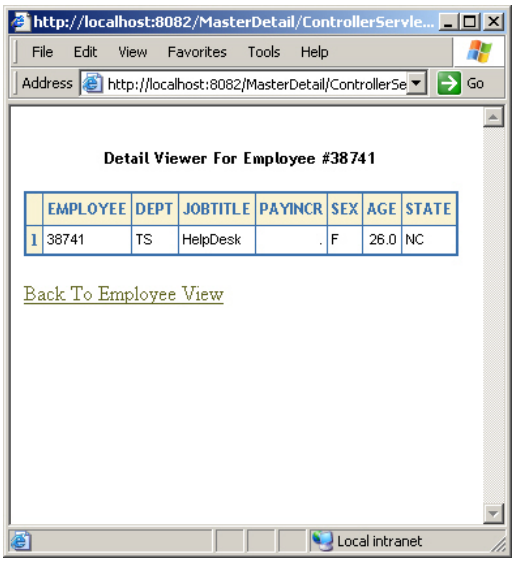
This example demonstrates how to use the APF to add new functionality for a TableView component. Specifically, it shows how to setup and register a new Action for the data cells of the EMPLOYEE column as seen in the image below.

Figure 10



When a user clicks on an employee number they will be taken to the detail view for that employee. The detail view will be presented by a different TableView component as demonstrated by figure 11.

Figure 11



The first task is to write a command class that can perform a subset operation on the employee detail data. The BaseCommand class is a good choice as the subclass for the new command because it implements the com.sas.commands. DynamicAttributeCommandInterface which is required of all Action commands.

Follow these steps to add the command to your webAF project.

- Select **File→New** to open the *New* window and on the *Files* tab select **Java Source File**.
- Name it *SubsetCommand.java* and specify *commands* for the package name.
- Choose the **Blank Java File** option on the second page of the wizard.
- Enter the following code into the java source file.

```
package commands;
import com.sas.commands.BaseCommand;
import com.sas.storage.jdbc.JDBCToTableModelAdapter;

public class SubsetCommand extends BaseCommand {
      private JDBCToTableModelAdapter model;
      private String subsetValue;

      public SubsetCommand(JDBCToTableModelAdapter adapter) {
            this.model = adapter;}

      public void setSubsetValue(String subsetValue) {
            this.subsetValue = subsetValue;}

      public String getSubsetValue() {
            return subsetValue;}

      public void execute(Object o) {
```

```
                String newQuery = "SELECT * FROM sasuser.empdata WHERE
                employee = '"+subsetValue+"'";
                model.setQueryStatement(newQuery);
        }
    }
```

Note that the command's subsetValue attribute is the dynamic piece of information needed to perform the operation. The new Action you set up will need an attribute for that information so that it can be passed to the command during the execution phase.

The next step, though, is to create a separate JSP that uses a TableViewComposite to present the employee detail data.

- Select **File→New** to open the *New* window.
- In the *Files* tab select **Java Server Page**.
- Name the file *index2.jsp* and then enter the following code:

```
<%@ taglib uri="http://www.sas.com/taglib/sas" prefix="sas" %>
<%@ page pageEncoding="UTF-8"%>
<html>
<head>
   <link href="styles/sasComponents.css" rel="STYLESHEET"
   type="text/css">
</head>
<body>
   <sas:TableViewComposite ref="sas_TableView2" scope="session">
      <sas:TableTitle text="<%="Detail Viewer For Employee #" +
      request.getParameter("sas_actionProvider_subsetValue")%>" />
   </sas:TableViewComposite>
   <br>
      <sas:Label id="label1" text="Back To Employee View" URL="
      JDBCTableViewExample"/>
</body>
</html>
```

Now you are ready to begin the final customizations to `JDBCTableViewExampleControllerServlet.java`. In this file, you will add code that does the following:

- Create another JDBC adapter for the employee detail data.
- Create a new TableViewComposite bean for presenting the detail data.
- Create and register the new Action for the first TableView component.

First, add these import statements to the existing import section of the file:

```
import commands.SubsetCommand;
import com.sas.actionprovider.Area;
import com.sas.entities.AttributeDescriptorInterface;
import com.sas.servlet.tbeans.StyleInfo;
import com.sas.servlet.tbeans.menubar.html.MenuBar;
import java.util.HashMap;
import java.util.Map;
import java.util.Vector;
```

Next, add the code section below right after the line:

```
   sortAscendingAction.putValue( HttpAction.SHORT_DESCRIPTION, "Sort
   in Ascending Order");
```

that was added to JDBCTableViewExampleControllerServlet.java in the previous example:

```
❶ String jdbcQuery2 = "select * from sasuser.empdata";
JDBCToTableModelAdapter adapter2 = null;
if (session != null) {
      adapter2 =
      (JDBCToTableModelAdapter)session.getAttribute("sas_model2");
}
if (adapter2 == null){
   try{
      adapter2 = new JDBCToTableModelAdapter(sas_JDBCConnection,
      jdbcQuery2);
      if (session != null)
         session.setAttribute("sas_model2", adapter2);
   }
   catch(Exception e){throw new RuntimeException(e);}

   /* Create the second TableViewComposite bean */
   TableViewComposite sas_TableView2 = null;
   if (session != null){
      sas_TableView2 =
      (TableViewComposite)session.getAttribute("sas_TableView2");
   }

   if (sas_TableView2 == null){
      sas_TableView2 = new TableViewComposite();
      sas_TableView2.setModel(adapter2);
      sas_TableView2.setActionProvider(sas_actionProvider);
      session.setAttribute("sas_TableView2", sas_TableView2);
   }
}

❷ HttpAction subsetAction = new HttpAction(new SubsetCommand(adapter2));
subsetAction.setActionType("SUBSET_ACTION");

❸ subsetAction.putValue( subsetAction.NAME,
                        "%"+HttpTableViewSupport.AREA_VALUE_ATTRKEY);

❹ subsetAction.putValue("employee", null );
AttributeDescriptorInterface adi =
subsetAction.getAttributeDescriptor("employee");
adi.setSupplementalProperty(subsetAction.CUSTOM, Boolean.TRUE);
adi.setLabel(null,"subsetValue");

❺ subsetAction.putValue(ComponentKeys.FORWARD_LOCATION, "/index2.jsp" );
adi = subsetAction.getAttributeDescriptor(ComponentKeys.FORWARD_LOCATION);
adi.setSupplementalProperty(subsetAction.DYNAMIC, Boolean.FALSE);
adi.setSupplementalProperty(subsetAction.EXTERNAL, Boolean.TRUE);

❻ Map styleMap = new HashMap();
styleMap.put(MenuBar.MENU_LINK,new StyleInfo("menuItemLink"));
subsetAction.putValue(subsetAction.STYLE_MAP, styleMap);

❼ Vector viewers = new Vector(1);
viewers.add(
sas_TableView1.getComponent(sas_TableView1.TABLEVIEW_TABLEDATA) );
sas_actionProvider.setAction( subsetAction, viewers,
         new Area( HttpTableViewSupport.DATA_CELL_AREA, "EMPLOYEE" ));
```

EXAMPLE CODE EXPLAINED

Section 1:

In this section, you are adding code to create and setup the second TableViewComposite and the adapter it needs to connect to the employee detail data. Notice that the second TableViewComposite shares the Action Provider with the first TableViewComposite.

Section 2:

These lines instantiate the new Action, give it an instance of your new Command and assign it a unique actionType. The actionType may be any String of your own choosing as long as it does not match any of the existing actionTypes defined by the support class (HttpTableViewSupport, in this case).

Section 3:

This section is where we setup the Action's NAME attribute so that it gets generated dynamically. The APF uses the NAME attribute as the displayed text for an Action. In this case, the text should be the data cell value.

The code is building a template pattern for use as the Action's dynamic value. Wherever the APF sees the '%' symbol in an attribute *value*, the subsequent token is expected to be a reserved key that the support class knows how to determine dynamically. The APF will substitute the dynamic value for the token during the Action acquisition phase. Here, we are using the AREA_VALUE_ATTRKEY token which is available on most support classes for acquiring the default value for a specific area.

This key is also useful for use as the *name* of a dynamic Action attribute. Whenever the APF sees a reserved key as the name of an Action attribute, it will set the attribute's value with the dynamically determined value. For example, you could get the name of a column passed as a request parameter if you used the AREA_VALUE_ATTRKEY as an attribute name for an Action in the COLUMN_HEADER_AREA.

Section 4:

In this section we are setting up the dynamic Action attribute that the command will need during the execution phase. First, notice that the code is getting an AttributeDescriptorInterface for the attribute. These objects contain important metadata-type information with respect to how the APF treats individual Action attributes. Refer to the *More on Action Attributes And Their Descriptors* section below for more information.

Here the code is setting the CUSTOM property to true so that the APF knows this is a special type of attribute that is handled differently by individual support classes. In this case, the HttpTableViewSupport expects that CUSTOM Action attributes have names that match column names. When that is so, the attribute is set with the data cell value in that column.

We want the data values from the first column so we specify *employee* as the attribute name. However, we also need the Action attribute name to match the

command's subsetValue attribute. We can give the attribute an alias via the line that sets the AttributeDescriptorInterface label attribute.

Section 5:

Here the code is setting up a non-dynamic attribute that will enable the JDBCTableViewExampleControllerServlet to forward control to the right jsp (index2.jsp) when this particular Action is performed.

The attribute is marked as not DYNAMIC so that the APF does not attempt to determine the attribute's value during the acquisition phase. It is also marked as an EXTERNAL attribute which indicates to the framework that it may be ignored by the HttpActionProvider during the execution phase. The framework does not prefix the names of external attributes on the request with the HttpActionProvider's parameterPrefix.

Section 6:

The default style class used for an Action link on the first level of a query is *viewerSpecificPrefix* + *menuLink*. However, that resolves to `tableviewmenuLink` for this viewer which is not currently defined in `sasComponents.css`. This code tells the APF to use `tableviewmenuItemLink` instead.

Section 7:

This section registers the new Action with the framework. The code specifies the TableView sub-component of the TableViewComposite as the only viewer it wants to affect.

It also specifies which area to register the Action via the Area object. The first argument to this object's constructor is the areaType key. If we wanted the Action to appear in all data cells, the code would not need to specify a second argument to the constructor. However, we want the Action to only appear in one column so the code must specify a particular area *value*. Additional values may be specified via the Area's addValue() method.

After adding these customizations, choose **Build→Rebuild All** to compile your changes. Then select **Build→Execute in browser**. You should see the TableViewComposite presenting timecard data for all employees in the first view as shown in Figure 12 and be able to click on an employee number to be transitioned to the detail view for an employee as shown in Figure 13.

MORE ON ACTION ATTRIBUTES AND THEIR DESCRIPTORS

There are two basic types of Action attributes:

- **Class-based**: These are attributes defined by the Action class. These include the common attributes you will find documented in the API for the *com.sas.actionprovider.SwingAction* and *com.sas.actionprovider.HttpAction* classes.
- **Instance-based**: These are attributes that are not defined by any Action class. Instead, these attributes are added to an Action after it is instantiated to serve a unique purpose for the particular type of Action. These include the *employee*

and *ComponentKeys.FORWARD_LOCATION* attributes you added in the example above.

Every attribute (class or instance-based) has an AttributeDescriptor object that stores metadata-type information in the form of properties.

The first of these is the *visible* property. When set to true, the attribute is included on the URL generated for the Action and is passed on the request object during the execution phase. Unless also marked as EXTERNAL (see below), the APF adds a parameter prefix to the name of the request parameter. By default, the parameter prefix is the name of the Action Provider followed by the underscore character. This is done so that the Action Provider can distinguish APF request parameters from non-APF parameters during the execution phase.

When the visible property is false, the HttpActionProvider stores the attribute until it is needed during the next execution phase. Attributes whose values can not be transformed to a meaningful String value must be stored (visible=false) since all request parameters have String values. The default visible property value for instance-based attributes is true.

The visible property may be set via the following:

```
AttributeDescriptorInterface.setVisible(true/false);
```

The remaining properties are all supplemental properties and are set via the following:

```
AttributeDescriptorInterface.setSupplementalProperty(
Action.PropertyKey, Boolean.TRUE/Boolean.FALSE);
```

Where PropertyKey is one of the following:

- DYNAMIC: When true, the APF attempts to determine the value of the attribute during the Action acquisition phase. The default for instance-based attributes is true.
- EXTERNAL: When true, the APF will include the attribute on the Action's URL but will not prefix the parameter name with the HttpActionProvider's parameterPrefix. This allows the HttpActionProvider to ignore the parameter during the execution phase. The default for instance-based attributes is false.
- CUSTOM: When true, the APF expects that the support class has special rules for determining a dynamic value for the attribute based on its name. The support classes for the relational TableView components are the only ones that currently support this feature. They expect the attribute name will match the name of a column and will set the attributes value to that of data cell value in that column. The default for instance-base attributes is false.

For more information on these and other supplemental property keys, refer to documentation in the API specification for the com.sas.actionprovider.BaseAction class found here:
http://support.sas.com/rnd/gendoc/bi/api/Components/com/sas/actionprovider/BaseAction.html

APF VERSUS STRUTS

Struts is web application development framework that is based on the MVC architecture. It simplifies many *application scope* issues for developers so that they may focus on developing just the business logic and presentation layers.

In contrast, the APF is a framework that simplifies *component scope* issues for developers. It is a complementary framework that integrates easily with Struts based applications.

Integration is simple because of the options the developer has to affect the URLs of APF Actions so that they may be handled via one or multiple custom Struts Actions. For example,

- Action Provider level: HttpActionProvider.controllerURL
- Viewer component level: URLTemplateViewInterface.URLTemplate
- Action level: HttpAction.URLBase and URLTemplate

The developer may choose (via the *controllerURL* attribute) to have a single Struts Action for handling all APF Action executions like the JDBCTableViewExampleControllerServlet does in the examples above. Alternatively, the developer may choose to handle APF Actions on a per-Viewer or per-Action basis via the attributes at the other levels.

The *useReferringURI* property on the HttpActionProvider may also be used to force all APF Actions to include the URI of the viewer component that presents the Action as a request parameter. This makes it possible for Struts Action to forward control back to the presentation JSP when appropriate.

See the documentation of these attributes and others on the HttpActionProvider, HttpAction and URLTemplateViewInterface classes in the com.sas.actionprovider package at the AppDevStudio API.
http://support.sas.com/rnd/gendoc/bi/api/

CONCLUSION

The APF is an integrated set of Java classes introduced with AppDev Studio ® 3. We saw how the APF simplifies the work involved in customizing or defining new actions for viewers in client side and server side Java applications. We saw how the key pieces of the framework work and proposed some simple examples. The developer who would like to know more about the APF can read the documentation which is referenced in the section below.

ADDITIONAL RESOURCES AVAILABLE

At http://support.sas.com/rnd/gendoc/bi/api/ you will find the SAS Business Intelligence API documentation.

The following sections are important ones with respect to the APF:

- The package documentation of the com.sas.actionprovider package.
- The introductory documentation for each of the following classes:
    - com.sas.actionprovider.HttpAction

- com.sas.actionprovider.SwingAction
- com.sas.actionprovider.HttpActionProvider
- com.sas.actionprovider.SwingActionProvider
- com.sas.actionprovider.ActionOrderList.

At: http://support.sas.com/rnd/appdev/examples/index.html you will find the SAS AppDevStudio Developer examples site with some interesting examples about the APF.

ACKNOWLEDGMENTS
We would like to thank Tammy Gagliano, Anton Fuchs, Chris Barrett, Brian Durham and Corey Benson for the valuable information and examples they provided for this document.

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

Flavio Ubaldini
SAS Institute
Neuenheimer Landstr. 28-30
69120 Heidelberg, Germany
Work Phone: (+49) (0)6221 416 - 0
Email: Flavio.Ubaldini@eur.sas.com

Neil Bell
SAS Institute Inc
100 SAS Campus Dr
Cary NC 27513-8617, USA
Email: Neil.Bell@sas.com