A New Framework to Develop JSP DynPage Based Applications

Applies to:

NW 2004 SE SP9 and above, developed and tested on NW2004SSP18. For more information, visit the Portal and Collaboration homepage.

Summary

This article throws light on the difficulties being faced while developing JSP DynPage based applications using the conventional framework. This article also discusses how to handle these difficulties by introducing changes to the conventional framework and hence, the new framework which reduces the complexity of the application and makes developer's life easier.

Author: Vikash Kumar

Company: HCL AXON

Created on: 10 June, 2011

Author Bio

Vikash Kumar is working as a SAP NetWeaver Consultant for HCL Axon, India. He has been working on SAP NetWeaver Portal Development projects in the JAVA stack. He also has a fair knowledge of ABAP.

Table of Contents

Overview of JSP DynPage

JSP DynPage is a Java IView development model that utilizes an MVC approach of separating application logic, business data and presentational directives into separate entities. The application's core logic can be coded into the Java class, the business data can be transported using Java beans and the UI can be defined in a JSP page.

The strength of the JSP DynPage is the event handling. The JSP DynPage follows the concept of Java controls - Java controls, like the HTMLB controls, can have one or more events. The event can be defined by assigning a method name to the event. The method is called whenever the event is raised (for example, when a button is clicked). The event handling method is coded in the JSP DynPage. The JSP DynPage does the event handling and calls the proper event handling method.

Event Processing

Let us discuss the event processing for a JSP DynPage application.

The very first time, when the application starts, the doInitialization() method is called which is responsible for initialization of data. After initializing the data, doProcessBeforeOutput() is called which decides which jsp page is to be displayed.

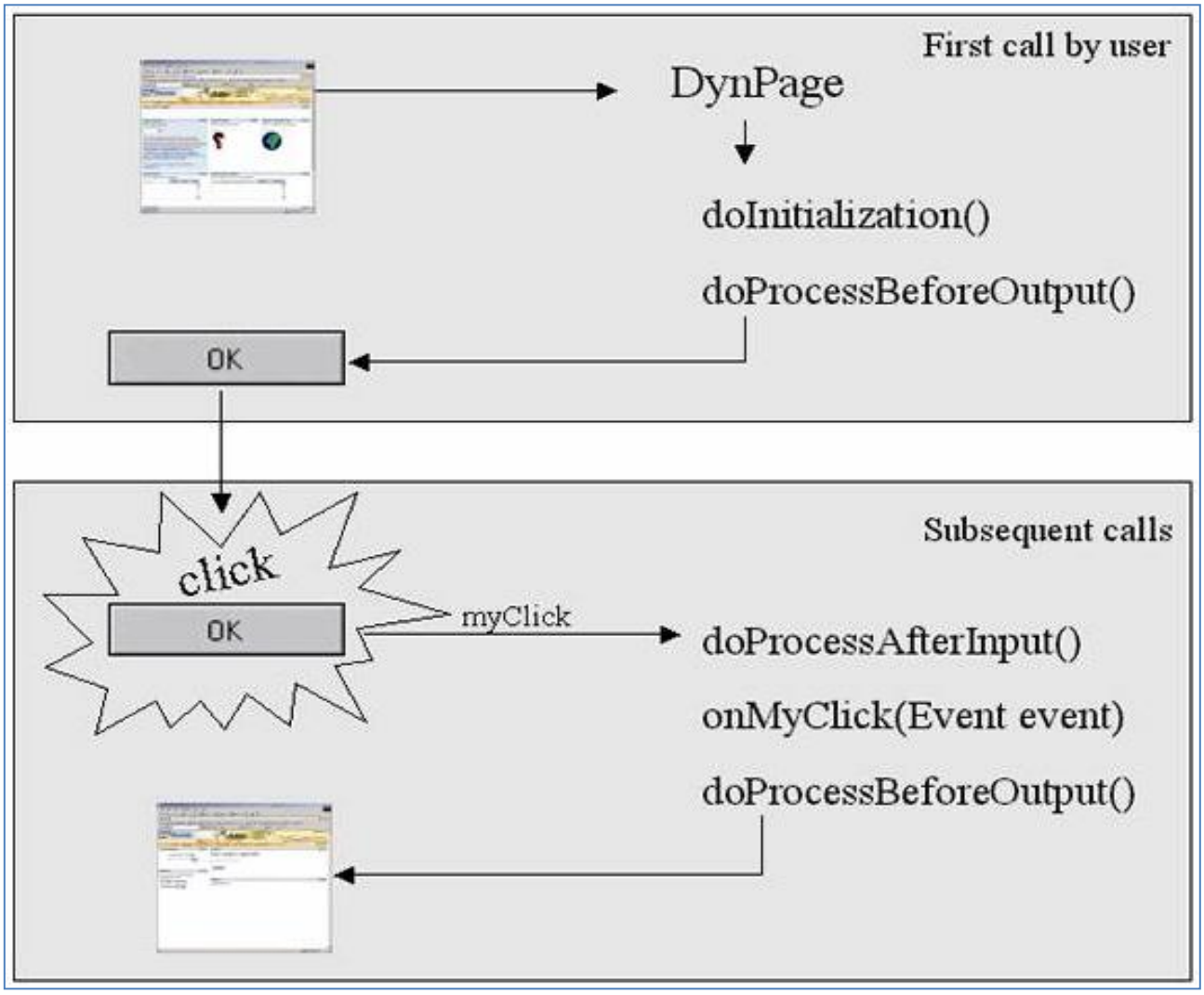

Figure 1 : Event processing

As soon as any button is clicked or any event is triggered by any means, the `doProcessAfterInput()` is called which is responsible for handling the inputs present on the screen. After this, the method associated with the element is called and executed. Lastly, the `doProcessBeforeOutput()` method is called which displays the jsp after the event processing is complete. This is how an event is processed.

Note: Have a look at the event processing diagram. You will notice that the doProcessBeforeOutput() method is called every time while doProcessAfterInput() is called whenever an event is triggered except for the first time when the application starts i.e. when doInitialization() method is called.

Conventional way of developing application

As per the documentations and the tutorials available, here is the way as to how JSP DynPage based applications are developed:

- The doInitialization() method is called first when the application starts.
- If there are input fields (any type, may be a drop down or check boxes etc) on the screen then on button click, the doProcessAfterInput() method is called to capture the value of the input fields which are then set to bean and the corresponding operation is performed.
- After doProcessAfterInput() method, the event handler method ("onClick" attribute mentioned in jsp in the corresponding button tag) is to be coded in the DynPage only.
- After all required operations are performed, the event method changes the state of the application and accordingly, the doProcessBeforeOutput() method redirects to the jsp corresponding to the state changed by the event handler method.

If you are a novice developer and want to see the tutorial from where the above inferences have been drawn please click here.

The framework looks good for the tutorial mentioned above. But there are some drawbacks if the developer wants to code for an application which has multiple jsp pages. Let us discuss the same using an example.

Drawbacks of Conventional Framework

Let us take a very simple application "Student Info Management System" which has three screens namely the 'Welcome Screen', the 'Add Record Screen' and the 'Search Info Screen'. All these screens are discussed here. Figure 2 shows the welcome screen.

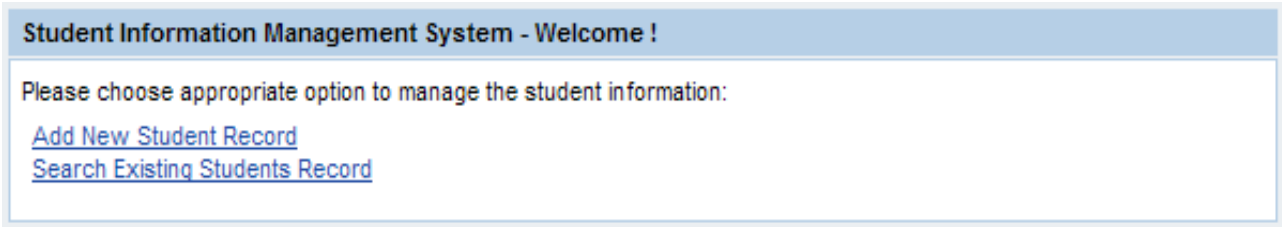

Figure 2 : The Welcome Screen

The welcome screen contains the two links, as shown in Figure 2. The first link redirects to the 'Add Record Screen' while the other one redirects to the 'Search Info Screen'.

These two links have the "`onClick`" attribute in the jsp. This means that by clicking on the links, the corresponding event will be triggered.

```
// Add New Student Record Link in jsp
<hbj:link
   id="optAddRec"
   text="Add New Student Record"
   onClick="optAddRec"
   />
```

```
// Search Existing Students Record Link in jsp
<hbj:link
   id="optSrchRec"
   text="Search Existing Students Record"
   onClick="optSrchRec"
   />
```

Below is how the 'Add Record Screen' looks (Figure 3).

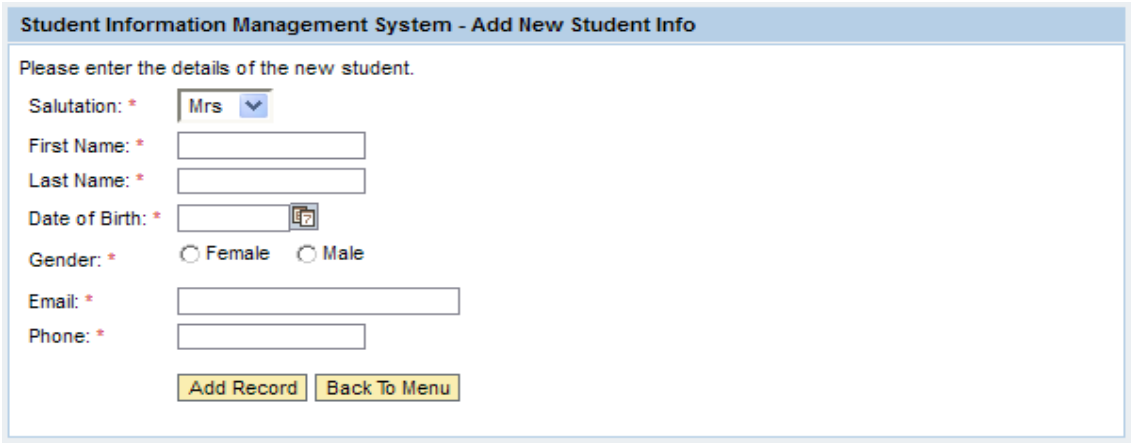

Figure 3 : Add Record Screen

In this screen we have two buttons and both have "`onClick`" attribute in the jsp. This means that by clicking on the buttons, the corresponding event will be triggered.

```
// The "Add Record" button in jsp
<hbj:button
   id="btnAddRecord"
   text="Add Record"
   width="80"
   design="STANDARD"
   onClick="onAddRecordClick">
</hbj:button>

// The "Back To Menu" button in jsp
<hbj:button
   id="btnBackToMenu"
   text="Back To Menu"
   width="100"
   design="STANDARD"
   onClick="home">
</hbj:button>
```

And lastly, we have the 'Search Info Screen'. This is a very simple screen where there is only one input field, the 'First Name' and a 'Search' button. The screen print of the screen is shown in figure 4.

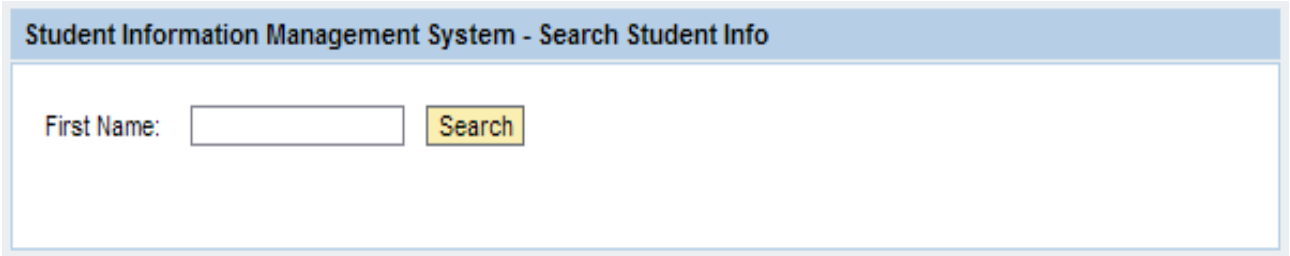

Figure 4 : 'Search Info Screen'

Here, in this screen, the 'Search' button has "`onClick`" attribute in the jsp. This means that by clicking on the buttons, the corresponding event will be triggered.

```
// The 'Search' button in jsp
<hbj:button
   id="btnSrchStdnt"
   text="Search"
   width="40"
   design="STANDARD"
   onClick="onStdntSrch">
</hbj:button>
```

So in total, we have five events and these events will be having five corresponding event handler methods, which according to documentation, must be coded in the component class which extends the JSP DynPage.

I will come back to the event handler methods, but first take a look at the `doProcessAfterInput()` method, as this is the method which is called first whenever an event is triggered.

Problem with doProcessAfterInput()

Let's say I want to code for adding a record. The format of coding could be similar to the one which is shown in figure 5.

```
public void doProcessAfterInput() throws PageException {
      // handle the input for Add New Record jsp
      // set bean
      // perform operation

 }
```

Figure 5 : Coding format for doProcessAfterInput() method for adding new record

After this, the event method `onAddRecordClick()` will be called and then the `doProcessBeforeOutput()` method will forward to the required jsp. I will come to event handler methods and the `doProcessBeforeOutput()` method later.

Suppose the coding for adding a new record is completed. Now I want to code for searching the record. In this screen also we have one input field which needs to be handled. To handle inputs, after the search event is triggered, the `doProcessAfterInput()` method comes into play. But the `doProcessAfterInput()` method is already coded for add record operation. So how can the developer code for handling the inputs of search screen?

Well here, the novice developers get trapped, as was I. This was a problem and the big question was how will the input fields be handled inside the single `doProcessAfterInput()` method for applications having more than one screens?

But, if you look at the event processing diagram, figure 1, you will come to know that this problem can be resolved if the whole stuff of the `doProcessAfterInput()` method is moved to the event methods. This is because, whenever an event is triggered, `doProcessAfterInput()` is called and then the specific event handler method is called. So, what if the `doProcessAfterInput()` method is left blank and all the inputs are handled separately by the specific event handler methods? Nice idea. So let's move on to the event handler methods.

Problem with Event Handler Methods

Here, as discussed earlier, we have five events, and hence we have five event methods corresponding to the five events. Now the documentation says that all these event methods must be coded in the DynPage class which extends the JSP DynPage.

Think of an application which has multiple jsp pages. Since the pages are more, it is a possibility that controls will be more and hence events will be more. To handle these events there will be corresponding

event handler methods. Let's say there are n no of event handler methods. The problem is all the event handler methods will be in the same class. You can think of the complexity of the application. It will be very difficult to maintain the application and if the events increase, it will make the application more complex.

So here is the second problem. The question is will it be easy to maintain the application if all the event handlers are coded in the same component class?

The next method which is called is the `doProcessBeforeOutput()` method. Let's discuss the last method now.

Problem with doProcessBeforeOutput() Method

As discussed earlier, this method is called every time. This method is responsible for redirecting to the appropriate jsp based on the inputs received from the event handler methods (when an event is triggered) or the `doInitialization()` method (at the time of application start).

If you have followed the tutorial which I mentioned earlier (if not, click here), you will notice that for applications having two jsp pages, it is very easy.

```
public void doProcessBeforeOutput() throws PageException {

    // set the JSP which builds the GUI
    this.setJspName("OutputText.jsp");

}
```

Figure 6 : doProcessBeforeOutput() method for application having two jsp pages

As the no of jsp pages increase, the "Switch Case" logic is implemented here. If there are n no of jsp pages then it will look somewhat like figure 7.

```
public void doProcessBeforeOutput() throws PageException {
   switch (iState){
           case WELCOME_STATE :
               this.setJspName("firstJspDynPageSuccess.jsp");
               break;
           case
               BACK_STATE    :
               this.setJspName("userInfoForm.jsp");
               break;
                 *
                 *
                 *
         default :
             this.setJspName("userInfoForm.jsp");
             break;
       }
}
```

Figure 7 : doProcessBeforeOutput() method for multipage application

The question here is that is this the only way to handle multipage application?

These were the main drawbacks of the conventional way of coding an application with multiple jsp pages. To tackle these a new approach was needed.

Need of a new Framework?

The three drawbacks of the conventional way of coding, as discussed above forced to think of a new approach. To summarize, the problems with conventional way of coding are:

1. Single `doProcessAfterInput()` method cannot be used to handle inputs for multiple events.
2. Event handler methods are placed in a single class which makes the application difficult to understand and hence maintain.
3. `doProcessBeforeOutput()` method coding logic needs to be changed if the application has multiple jsp pages.

The New Framework

To overcome the drawbacks of the conventional way of coding, some changes have been introduced which led to a new framework.

The changes included are:

1. To substitute n no of event handler methods, DynPage class method `doProcessCurrentEvent()` has been overridden. Actually, this is the method which is called internally whenever an event is triggered. To handle n no of events and their operations, the factory pattern has been introduced which instantiates the appropriate action class based on the current event.
2. The inputs are now handled in the action classes separately. So `doProcessAfterInput()` method is now empty.
3. These action classes return the name of jsp which is to be shown on screen after the event is completed. So the `doProcessBeforeOutput()` method can be coded in a more efficient way.

A Look at the Changes in the Component Page

Here is a glimpse of the changes which have been made.

The doProcessAfterInput() Method

The `doProcessAfterInput()` method has been kept empty. All the stuff has been moved to the corresponding action classes of the events.

The Event Handler Method

The method `doProcessCurrentEvent()` of class DynPage has been overridden. Actually, this is the method which is called internally whenever an event is triggered.

In this method, below are the steps which are performed to execute the event.

1. Action for the current event is fetched. The value of this action is same as it is mention in the "`onClick`" attribute of the element responsible for the event.
2. Read all the values of the fields present on the page and store it so that the value of fields can be fetched later whenever required.
3. It instantiates the `ActionFactory` class and executes the corresponding action class for the action fetched in the first step.
4. These action classes return the name of jsp which is to be displayed after this event.

To handle n no of events and their operations, the factory pattern has been introduced which instantiates the appropriate action class based on the current event. The factory pattern will be discussed later. Let's first see the code for the above mentioned method.

```
/**
 * @Overridden
 * @see com.sapportals.htmlb.page.DynPage#doProcessCurrentEvent(com.sapportals.htmlb.event.Event)
 */
public void doProcessCurrentEvent(Event event) throws PageException {
    logger.log("Entering doProcessCurrentEvent of "+getClass().getName());
    try{
        IPortalComponentRequest request = (IPortalComponentRequest) this.getRequest();
        IPortalComponentResponse response = (IPortalComponentResponse)this.getResponse();
        String sAction = event.getAction();

        logger.log("Preparing Component Information for Command Classes");
        ComponentBean compInfoBean = new ComponentBean();
        compInfoBean.addComponent(getComponents());
        logger.log("Prepared Component Information for Command Classes");

        logger.log("Delegating Processing to Command Class for Action = '"+sAction+"'");
        ActionsFactory aFactory = ActionsFactory.getInstance();
        IActionProcessor aProcessor = aFactory.getActionClass(sAction);
        logger.log("Command Class To Execute :"+aProcessor.getClass().getName());
        FORWARD_PAGE = aProcessor.execute(request,response,event,compInfoBean);
        logger.log("Command Class Executed Successfully... forwarding to "+FORWARD_PAGE);

    }catch(Exception ex){
        logger.log(LogLevel.SEVERE,"Exception in doProcessCurrentEvent "+ex);
    }finally{
        logger.log("Exiting doProcessCurrentEvent of "+getClass().getName());
    }
}
```

Figure 8 : Substitute method for event methods

The doProcessBeforeOutput() Method

As discussed in “The event handler method” step 4, all the action classes return the name of jsp which is to be displayed after the event, no more switch case coding is required. It directly points to the jsp. No switch cases required anymore. Figure 9 shows the replacement for the switch cases. The code has been reduced to three lines only.

```
/**
 * PBO
 */
public void doProcessBeforeOutput() throws PageException {
    if(null!=FORWARD_PAGE){
        this.setJspName(FORWARD_PAGE);
    }
}
```

Figure 9 : the changed doProcessBeforeOutput() method

Let’s now come to the factory pattern.

The Factory Pattern

The ingredients of a factory pattern include:

- An Interface,
- N no of classes which implement the interface
- A singleton class

Let’s discuss this in the present scenario.

For factory pattern, let us create an Interface `IActionProcessor`, different classes for different actions which implement the interface `IActionProcessor` and lastly, one singleton class `ActionsFactory` which decides based on the runtime input of action that which action class is to be instantiated.

Let us see how these look like.

The Interface IActionProcessor

The interface `IActionProcessor` has a single method execute.

```
public interface IActionProcessor {
   public String execute(IPortalComponentRequest request, IPortalComponentResponse
   response, Event event, ComponentBean component) throws Exception;
```

The ActionsFactory Class

The factory class `ActionsFactory`, based on the action, reads the name of action class from the property file and instantiates the corresponding action class. Remember this is a singleton class i.e. single instance at a particular point of time is possible.

```
public class ActionsFactory {
    private static ActionsFactory instance = null;
    private Properties factoryProps;
    private static final String DEFAULT_ACTION  = "NavigateToHome";
    protected static final TestLogger logger = LogHelper.getTestLogger(LogConstants.PORTAL_TIER);
    protected ActionsFactory() {
        try{
            this.factoryProps = new Properties();
            this.factoryProps.load(getClass().getClassLoader().getResourceAsStream("...."));
        }catch(Exception ex){
            System.err.println("Error:"+ex);
        }
    }

    public static ActionsFactory getInstance() {
       if(null == instance) {
          instance = new ActionsFactory();
       }
       return instance;
    }

    public IActionProcessor getActionClass(String action)throws Exception {
        logger.log("Inside getActionClass in factory");
        if(!this.factoryProps.containsKey(action)){
            logger.log("No match found in property file");
            action = DEFAULT_ACTION;
        }
        logger.log("Match found in property file");
        String implClazzName = this.factoryProps.getProperty(action);
        Class newClazz = Class.forName(implClazzName);
        return (IActionProcessor)newClazz.newInstance();
    }
```

Figure 10 : The ActionsFactory Class

Note: Proper path of the property file should be passed to the place which is replaced by multiple dots.

Here the `doProcessCurrentEvent()` calls `getActionClass()` method of this class which reads property file and gets the name of the action class corresponding to the action passed and hence the class is instantiated.

ActionClassMapping.properties

```
optAddRec=com.hcl.portal.sim.actions.NavigateToAddInfo
optSrchRec=com.hcl.portal.sim.actions.NavigateToSearchInfo
home=com.hcl.portal.sim.actions.NavigateToHome
onStdntSrch = com.hcl.portal.sim.actions.SrchStdntInfo
onAddRecordClick = com.hcl.portal.sim.actions.AddStdntInfo
```

Figure 11 : The property file

The Action Class (an example)

Here is an example of action class.

Remember 'The Search Screen' of 'Student Information Management System'?

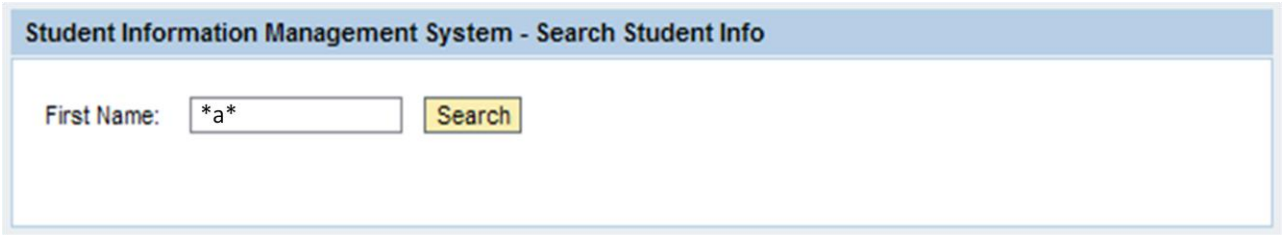

Figure 12 : The search screen

Let's say that the user wants to search for students whose first name contains the alphabet 'a'. So the user puts `*a*` in the first name input box and clicks on 'Search'. After clicking search, the overridden method first captures the action corresponding to the "`onClick`" event on 'Search' button.

Let's see the action for the "`onClick`" event of 'Search' button which was coded earlier in jsp using HTMLB.

```
<hbj:gridLayoutCell rowIndex="1" columnIndex="2" horizontalAlignment="LEFT">
 <hbj:inputField id="txtFirstName" type="STRING" design="STANDARD" width="100" maxlength="50"/>
</hbj:gridLayoutCell>
<hbj:gridLayoutCell rowIndex="1" columnIndex="3" horizontalAlignment="LEFT">
 <hbj:button id="btnSrchStdnt" text="Search" width="40" design="STANDARD" onClick="onStdntSrch" </hbj:bu
</hbj:gridLayoutCell>
</hbj:gridLayout>
<hbj:gridLayout id="grdUserSrchRes" debugMode="False" cellSpacing="10" width="75%">
```

Figure 13 : Action corresponding to onClick event of Search button

The corresponding action has been highlighted. It is `onStdntSrch`.

After capturing this, the overridden method keeps the id and the component corresponding to the input field 'first name' in the component bean for further processing. The `addComponent(Component[] components)` method of the component bean basically takes component array as input and adds all the components and their ids into a hash map. The component array is returned by another inbuilt method `getComponents()`. The `addComponent(Component[] components)` method can be coded as:

```
public void addComponent(Component[] components){
        if(null==components || components.length==0) return;
        for(int i=0;i<components.length;i++){
              addComponent(components[i]);
        }
  }
```

Where `addComponent(components[i])` puts the components to hash map. It can be coded as:

```
public void addComponent(Component comp){
        if(null==comp) return;
        map.put(comp.getId(),comp);
  }
```

After this, the singleton class, `ActionsFactory is instantiated` and the method `getActionClass is called` by passing the action `onStdntSrch`.

The method first reads the property file. The action Class which implements the interface, must have the execute method which gets executed and returns the forward jsp name.

```
optAddRec=com.hcl.portal.sim.actions.NavigateToAddInfo
optSrchRec=com.hcl.portal.sim.actions.NavigateToSearchInfo
home=com.hcl.portal.sim.actions.NavigateToHome
onStdntSrch = com.hcl.portal.sim.actions.SrchStdntInfo
onAddRecordClick = com.hcl.portal.sim.actions.AddStdntInfo
```

Figure 14 : The property file

According to this property file, the action class corresponding to `onStdntSrch` action is `SrchStdntInfo` which has been highlighted in figure 14. Below you can see the coding for the class:

```
/**
 * @author vikash_kumar
 *
 * To change the template for this generated type comment go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
public class SrchStdntInfo implements IActionProcessor{
    protected static final TestLogger logger = LogHelper.getTestLogger(LogConstants.PORTAL_TIER);

    public String execute(IPortalComponentRequest request, IPortalComponentResponse response, Event event, Componen
        logger.log("Entering execute() of "+getClass().getName());
        String fName=null;
        StudentInfoBean bean = new StudentInfoBean();

        InputField ifFName = (InputField)component.getComponentById("txtFirstName");
        if(ifFName != null){
            fName = ifFName.getValueAsDataType().toString();
        }

        bean.setFName(fName);

        BusinessDelegate delegate = new BusinessDelegate();
        CustomTableModel tableModel = delegate.srchStudentRecord(bean);

        request.getServletRequest().setAttribute("stdntModel",tableModel);
        logger.log("model set to request... \n the row count is :::: "+tableModel.getRowCount()+"\n and column coun

        String frwdJsp = "SrchRecord.jsp";

        logger.log("Exiting execute() of "+getClass().getName());
        return frwdJsp;
    }
}
```

Figure 15 : A glimpse of the 'SrchStdntInfo.java', the action class for 'onStdntSrch' action as per the property file

The method first retrieves the value which was entered in the ‘first name’ input field using the component bean by passing the id of the input field. By passing the id of the component one can retrieve the component by using getComponentById(String componentId) method of component bean.

This method can be coded as:

```
public Component getComponentById(String componentId){
      if(!map.containsKey(componentId))throw new RuntimeException("Component With
'"+componentId+"', Not Defined");
      return (Component)map.get(componentId);
}
```

In the last part of the coding, after calling DAO through Business Delegate, the method sets the returned table model containing the search result in the request to show the same on jsp. The method then returns the name of jsp to which the result is to be shown which is the same jsp in this case.

Figure 17 shows the search screen with search result.

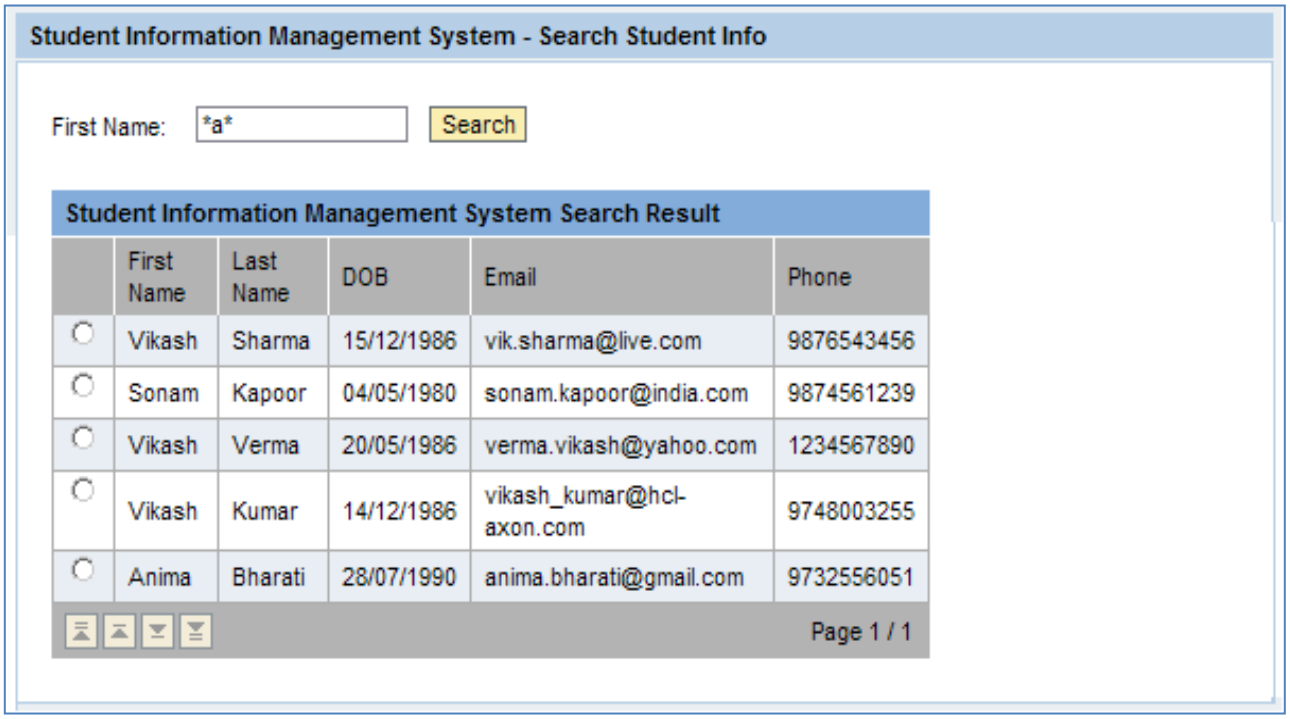

Figure 16 : The result page

The result has been shown in tableView control of HTMLB.

For more on HTMLB controls and their APIs you can refer to SAP HTMLB Guidelines.

If you want to customize this control by implementing the sorting and pagination functionality, you can do so by reading the article "Customizing HTMLB Table View Component" by Jithin R B.

So, this is how the factory pattern works and hence one can develop applications using the above mentioned new approach. This approach can be converted to a framework by moving the changes to another DC and exposing it as a public part.

The Architecture Diagram

Here is a look at the architecture diagram which can be used to develop JSP DynPage based applications having large no of jsp pages and hence large no of events. The figure 17 shows the new approach, the new framework.

The diagram is on next page.

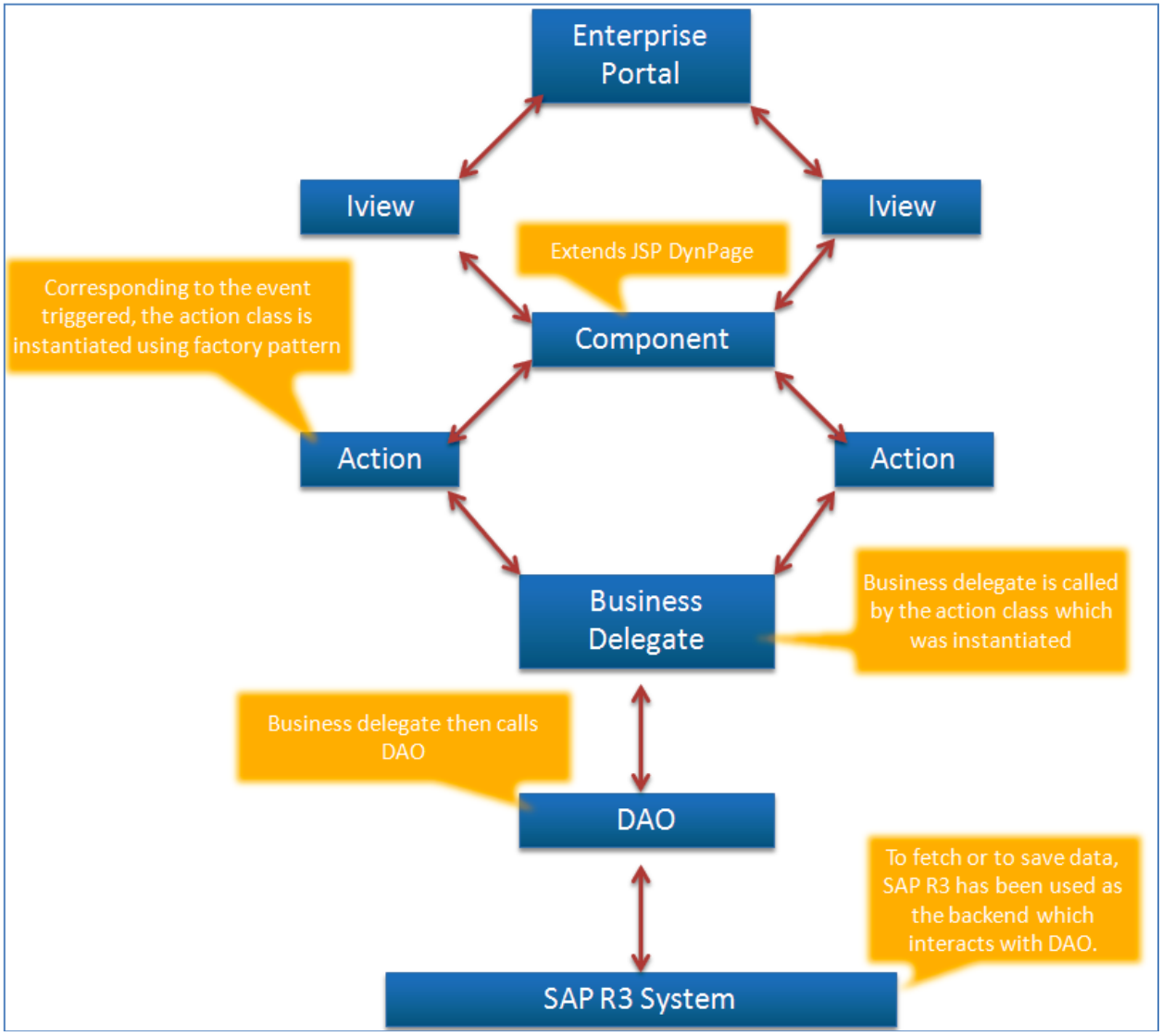

Figure 17 : The Architecture Diagram

Related Content

Creating the JSP DynPage

SAP HTMLB Guidelines

Customizing HTMLB Table View Component by Jithin R B

For more information, visit the Portal and Collaboration homepage.

Disclaimer and Liability Notice

This document may discuss sample coding or other information that does not include SAP official interfaces and therefore is not supported by SAP. Changes made based on this information are not supported and can be overwritten during an upgrade.

SAP will not be held liable for any damages caused by using or misusing the information, code or methods suggested in this document, and anyone using these methods does so at his/her own risk.

SAP offers no guarantees and assumes no responsibility or liability of any type with respect to the content of this technical article or code sample, including any liability resulting from incompatibility between the content within this document and the materials and services offered by SAP. You agree that you will not hold, or seek to hold, SAP responsible or liable with respect to the content of this document.