Space Details

Key:	PentahoDoc
Name:	BI Server Documentation - Latest
Description:	Latest version of the Pentaho BI Server
Creator (Creation Date):	admin (Nov 15, 2006)
Last Modifier (Mod. Date):	admin (Nov 27, 2006)

Available Pages

- Building Components
  - Architecture
  - BI Component APIs
    - done
    - executeAction
    - getLogger
    - init
    - validateAction
    - validateSystemSettings
  - Internal API
    - Handling Inputs
      - getDataSource()
      - getInputBooleanValue()
      - getInputLongValue()
      - getInputNames()
      - getInputParameter()
      - getInputStream()
      - getInputStringValue()
      - getInputValue()
      - isDefinedInput()
    - Handling Outputs
      - getDefaultOutputStream() - deprecated
      - getOutputContentItem()
      - getOutputContentItem() - deprecated
      - getOutputNames()
      - getOutputPreference()
      - getOutputStream()
      - isDefinedOutput()
      - setOutputMimeType()
      - setOutputValue()
    - Handling Resources
      - getResource()

        - getResourceAsString()
        - getResourceDataSource()
        - getResourceInputStream()
        - getResourceNames()
        - isDefinedResource()
    - Handling User Interaction
        - createFeedbackParameter() - 3 Parameter
        - createFeedbackParameter() - 5 Parameter
        - createFeedbackParameter() - 7 Parameter
        - feedbackAllowed()
        - getFeedbackOutputStream()
        - isPromptPending()
        - promptNeeded()
        - promptNow()
        - setFeedbackMimeType()
- Why Create a Pentaho Component?

Building Components

This page last changed on Nov 29, 2006 by bseyler.

This guide describes the why and how of adding a new component to the Pentaho Business Intelligence (BI) Suite. It is assumed that you have read and understood the following other guides before reading this one:

- Technical White Paper
- Creating Pentaho Solutions

Or continue on to the pages in this document:

- Architecture
- BI Component APIs
- Internal API
- Why Create a Pentaho Component?

Architecture

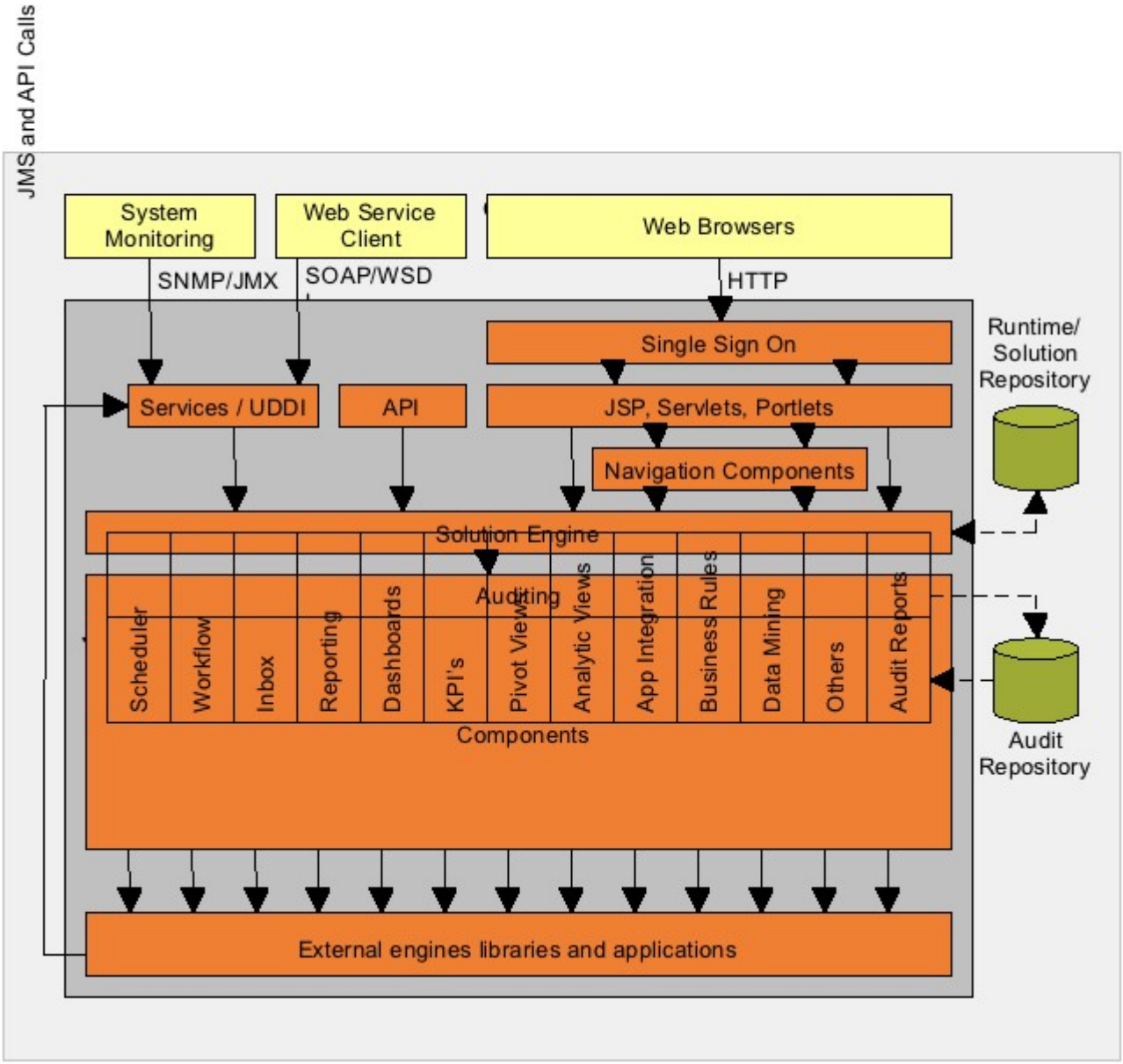
This page last changed on Dec 05, 2006 by bseyler.

Building Components BI Component APIs

The architecture is designed so that each component only needs to do the smallest amount of work necessary to perform its job. A component does not need to know anything about schedulers, workflow engines, audit logs, or other components.

Control Flow

BI Components are executed as a result of someone or something requesting the Pentaho Solution Engine to execute an action sequence. Action sequences can be executed by users, schedulers, web services, workflow engines and API calls. In all cases the caller can provide parameters to the action sequence. The action sequence can, in addition to parameters from the request, define parameters that are fulfilled from other places such as a session. This section and chart describes the order of events as an action sequence is executed.



1. Start. A call is made to an object in the services layer requesting execution of an action sequence. These objects use a request handler object that implements org.pentaho.core.services.IActionRequestHandler, create an instance of org.pentaho.solution.SolutionEngine, or use the helper methods in org.pentaho.core.solution.SolutionHelper. In each of these cases, parameters providers are passed to the solution engine. The request handlers and solution helper create an instance of the SolutionEngine and call its 'execute' method specifying the path and name of the action sequence to be executed and the parameter provider.
   Examples of objects that do this are:
   - Servlet: org.pentaho.ui.servlet.ViewAction
   - UI Component: org.pentaho.ui.component.ActionComponent
   - Web Service: org.pentaho.ui.servlet.HttpWebService
   - Scheduler: org.pentaho.plugin.quartz.QuartzExecute

    - API: org.pentaho.core.solution.SolutionHelper
2. The Solution Engine's 'execute' method:
    - Checks that both the action sequence path and name have been provided
    - Creates an instance of an IRuntimeElement and an IRuntimeContext. The details of the creation of this instance are added to the audit log along with its association with the caller's session. The Runtime Element stores the state of the action sequence if the process is long running, and the Runtime Context will handle the execution of the components.
    - Calls the Solution Repository to load the action sequence. If the action sequence cannot be found, an error is returned to the caller and the failure is audited.
    - Calls the Runtime Context's 'validateSequence' method.
3. The Runtime Context's 'validateSequence' method:
    - Sets the logging level of the action sequence to the desired level (can be specified by the original caller).
    - Creates an instance of a component for each action defined in the action sequence. If the sequence includes two actions which use the EmailComponent, two instances of the component will be created. The same instance of the component will not be used for both actions. The Runtime Context needs BI Components to have a default constructor that accepts no parameters.
    - Provides each component with various objects such as the caller's session, the requested logging level, and the Runtime Context.
    - Calls each component's 'validate' method.
4. The component's 'validate' method:
    - Calls its own 'validateSystemSettings' method to validate any system-wide settings it needs.
    - Calls its own 'validateAction' method to validate that the inputs, resources, and outputs available to it (as defined in the action sequence) are sufficient for it to execute.
      If either of these validations fail for any component, the execution of the action sequence is terminated and the status is returned to the caller.
5. The Solution Engine's 'execute' method calls the Runtime Context's 'executeSequence' method.
6. The Runtime Context's 'executeSequence' method:
   Steps and loops through the action definitions in the action sequence calling its 'executeAction' method for each action defined.
7. The Runtime Context's 'executeAction' method:
    - Audits the start of component execution.
    - Resolves the parameters to be made available to the component as defined in the action sequence, e.g. from the request parameter provider from the session parameter provider or from the outputs of other components.
    - Calls the component's 'init' method.
8. The component's 'init' method performs any initialization steps it needs.
9. The Runtime Context's 'executeAction' method calls its 'executeComponent' method.
10. The Runtime Context's 'executeComponent' method calls the components 'execute' method.
11. The component's 'execute' method performs the steps it needs to complete its function.
12. The Runtime Context's 'executeComponent' method calls the components 'done method.
13. The component's 'done' method performs any clean-up steps it needs.
14. The Runtime Context's 'executeAction' method audits the end of the component execution.
15. The Solution Engine's 'execute' method
    - Audits the end of the action sequence execution
    - Returns the Runtime Context to the caller.
16. End. The caller can use the Runtime Context to
    - Get the final status of the execution
    - Get the output objects of the action sequence
    - Get the debug or error messages

Solution Engine	Runtime Context	BI Component
1. start	**1. start**	**1. start**

The request handlers and solution helper create an instance of the SolutionEngine and call it's 'execute' method specifying the path and name of the action sequence to be executed and the parameter provider.	The request handlers and solution helper create an instance of the SolutionEngine and call it's 'execute' method specifying the path and name of the action sequence to be executed and the parameter provider.	The request handlers and solution helper create an instance of the SolutionEngine and call it's 'execute' method specifying the path and name of the action sequence to be executed and the parameter provider.
2. execute() • Checks that an action sequence path and name have been provided • Creates an instance of an IRuntimeElement and an IRuntimeContext. • Calls the Solution Repository to load the action sequence. • Calls the Runtime Context's 'validateSequence' method.		
	3. validateSequence() • Sets the logging level. • Creates an instances components. • Provides each component with various system objects it needs. • Calls each component's 'validate' method.	
		4. validate() • Calls its 'validateSystemSettings' method to validate any system-wide settings it needs. • Calls its 'validateAction' method to validate that the inputs, resources, and outputs.
5. execute() Calls the Runtime Context's 'executeSequence' method.		
	6. executeSequence() Steps and loops through the action definitions in the action sequence calling 'executeAction'	
	7. executeAction()	

	Audits the start of a component execution. Resolves the parameters Calls the component's 'init' method.	
		8. init() Method performs any initialization steps it needs to.
	9. executeAction() Calls 'executeComponent'.	
	10. executeComponent() Calls the component's 'execute' method.	
		11. execute() Performs the steps it needs to complete its function.
	12. executeComponent() Calls the components 'done method.	
		13. done() Performs any clean-up steps it needs to.
	14. executeAction() Audits the end of the component execution.	
15. execute() Audits the end of the action sequence execution. Returns the Runtime Context to the caller.		
16. end • The caller can use the Runtime Context to • Get the final status of the execution • Get the output objects of the action sequence • Get the debug or error messages		

As you can see from the diagram there are only a few levels of method calls between the 'execute' of the Solution Engine and the 'execute' of the components. This lightweight framework is very powerful but does not impose a heavy processing load on the system.

BI Components are not cached between calls to the Solution Engine and must be thread safe.

If a BI Component needs to create any global resources or call any static initialization methods, a class

that implements *IPentahoSystemListener* should be written and registered in 'system/pentaho.xml'. See *org.pentaho.plugin.kettle.KettleSystemListener* and *org.pentaho.plugin.jfreereport.JFreeReportSystemListener* for examples.

Each BI Component needs to know the parameters that it needs to perform its duties. For example a scripting component needs to be provided the script to be executed whereas a report component needs to be provided the report template and the data to be included.

BI Component APIs

This page last changed on Dec 04, 2006 by mlowery.

Architecture Building Components Internal API

There are four ways of creating a BI Component: create a BI Component from scratch, convert an existing class into a BI Component, create a subclass of ComponentBase (org.pentaho.plugin.ComponentBase), or create a subclass of SimpleComponent (org.pentaho.plugin.core.SimpleComponent).

To become a BI Component, a Java object must implement the org.pentaho.core.component.IComponent interface. The IComponent interface extends two other interfaces: org.pentaho.core.audit.IAuditable and org.pentaho.util.logging.ILogger. Pentaho provides classes that implement these interfaces.

The hierarchy of the Pentaho classes looks like this:
org.pentaho.core.system.PentahoBase (implements ILogger and Serializable)
--> org.pentaho.core.system.PentahoMessenger
--> org.pentaho.plugin.ComponentBase (implements IComponent and IAuditable)
--> org.pentaho.plugin.core.SimpleComponent
--> other Pentaho-provided BI Components

You can create your BI Components in whatever package you like whether they are subclasses of Pentaho classes or not. There are no required methods that will be inaccessible if you use a non-Pentaho package.

Extend SimpleComponent

This is the easiest way to create a new component and the recommended place to start.

1. Create a new Java class that is a subclass of (extends) org.pentaho.plugin.core.SimpleComponent.
2. Implement an executeAction method and a getLogger method. See Component Methods below for a description of these methods.

Extend ComponentBase

If your component needs to validate its inputs and/or system settings or needs to perform any initialization and/or cleanup, you should extend org.pentaho.plugin.ComponentBase.

1. Create a new Java class that is a subclass of (extends) org.pentaho.plugin.ComponentBase.
2. Add code to the init, validateSystemSettings, validateAction, executeAction, done, and getLogger methods as appropriate.

Convert a Java Object into a BI Component

If you have an existing object that you wish to convert into a BI Component there are two options. Create a new object that subclasses your existing object and implements the required interfaces

Change your object to implement the required interfaces

Which of these options is the best for your particular object will depend on your specific circumstances; but, we recommend creating a subclass if only to keep your source file size manageable.

If you do need to use this approach most of the code you need to create your new class can be found in these classes: org.pentaho.core.system.PentahoBase, org.pentaho.core.system.PentahoMessenger, and org.pentaho.plugin.ComponentBase. To make this process easier we have created a class, org.pentaho.plugin.ComponentSubclassExample, that contains the code you need.

To convert a Java class to a BI Component:

1. Copy the imports, member variables, and methods from org.pentaho.plugin.ComponentSubclassExample into your Java class.
2. Add code to the init, validateSystemSettings, validateAction, executeAction, done, and getLogger methods as appropriate.

Components from Scratch

In order to create a component from scratch a new class that implements the interface org.pentaho.core.component.IComponent must be created. To implement all three interfaces requires about 30 methods to be implemented. It is not recommended to use this option and there should be no need to do this given the options above.

Component Methods

If using one of the three recommended methods for creating new components, the methods below are the BI Component methods that need to be customized to provide your needed functionality. The methods are listed in the order that they are typically called during normal processing. Depending on the option used to create your component, not all of these methods are required. See above sections for more details.

- done
- executeAction
- getLogger
- init
- validateAction
- validateSystemSettings

done

This method is called to give the component the opportunity to perform any cleanup operations that are necessary.

executeAction

This page last changed on Dec 04, 2006 by mlowery.

This method is called to cause the component to perform its function. Within this method you can call other internal API methods to get the values of inputs, to get resources, to ask users for parameters, and to get output streams. Typically, during the execute method a component will:

1. Gather input values. These might be parameters or component settings. If the input values are not complete, the component can stop executing or prompt the user for additional information.
2. Gather resources. These might be templates or definition files.
3. Get an output pipe of some kind (e.g. an output stream).
4. Create output contents.
5. Return the status of the execution.

We will use the executeAction method of the *PrintComponent (org.pentaho.plugin.print.PrintComponent)* as an example (minor modifications have been made for clarity).

```
protected boolean executeAction() {
    String printFileName = null;
    IActionResource printFileResource = null;

    // see if we are printing a file
    if (isDefinedInput("print-file")) {

        // get the name of the file to print
        printFileName = getInputStringValue("print-file");
    }

    InputStream inStream = null;
    // Get the name of the printer to use
    if (isDefinedInput("printer-name")) {
        printerName = getInputStringValue("printer-name");
    }
    // try to find the requested printer
    PrintService printer = getPrinterInternal(printerName);
    if (printer == null) {
        // the requested printer is not available
        if (!feedbackAllowed()) {
            // we are not allowed to prompt the user for a printer, we have to fail
            error( "The requested printer "+printerName+" is not available" );
            return false;
        }
        // prompt the user for an available printer
        // get a list of available print services
        PrintService[] services = PrinterJob.lookupPrintServices();
        ArrayList values = new ArrayList();
        // add each print service to our list of printers
        for (int i = 0; i < services.length; i++) {
            String value = services[i].getName();
            values.add(value);
        }
        // create a parameter for the user to select from
        createFeedbackParameter("printer-name",
            "select a printer", "", null, values, null, "select");
            return null;
        }
        promptNeeded();
        return true;
    }

    // Get the number of copies
    int copies = 1;
    if (isDefinedInput("copies")) {
        copies = Integer.valueOf(getInputStringValue("copies")).intValue();
    }
```

```
        // Check for a valid printFileName or printFile Resource
        if (printFileName != null) {
            try {
                inStream = new FileInputStream(printFileName);
            } catch (FileNotFoundException fnfe) {
                error(fnfe.toString(), fnfe);
                return false;
            }
            // Set the input source for sending to the driver.
            InputSource source = new InputSource(inStream);
            try {
                Driver driver = new Driver(source, null);
                PrinterJob pj = PrinterJob.getPrinterJob();
                pj.setPrintService(printer);
                PrintRenderer renderer = new PrintRenderer(pj, copies);
                driver.setRenderer(renderer);
                driver.run();
            } catch (Exception ex) {
                 error( "Could not print the document", ex);
                return false;
            }
            return true;
        }
```

getLogger

This page last changed on Nov 29, 2006 by mbatchelor.

This method creates a Log object that will be use to log messages from your component. It has to be created by your component (instead of a superclass) so that the log correctly records the class name of your object.

This method is implemented with a single line.

```
public Log getLogger() {
    return LogFactory.getLog(this.class);
}
```

init

This method is called to allow the component to perform any initialization operations that are required before the executeAction method is called.

validateAction

This method is called to give your component the chance to verify that it has the inputs and resources that it needs to complete successfully. If you return 'false' from this method execution of the action sequence will terminate.

This method should only be used to check that the inputs are available; it should not check the values of inputs because they are not guaranteed to be available at this point during execution. If your component needs resources such as a template file referred to in the action sequence as "new-employee-greeting", you can check that the resource is available by calling isDefinedResource() with "new-employee-greeting" as the argument.

If your component needs an input called 'employee-id', you can check that it has been defined in the action sequence by calling isDefinedInput() with "employee-id" as the argument.

If your component needs an output called "employee-greeting-email-text", you can check that it has been defined in the action sequence by calling isDefinedOutput() with "employee-greeting-email-text" as the argument.

Your validateAction method would now look like this

```
public boolean validateAction() {
    if (!isDefinedResource("new-employee-greeting")) {
        error( "A template called 'new-employee-greeting' has not been defined" );
        return false;
    }
    if (!isDefinedInput("employee-id")) {
        error( "An employee id is needed" );
        return false;
    }
    if (!isDefinedOutput("employee-greeting-email-text") ) {
        error("An output called 'employee-greeting-email-text' has not been defined" );
        return false;
    }
    return true;
}
```

Notice that we are only checking to see if the input has been defined and we are not trying to get the input value.

The validateAction of the EmailComponent looks like this:

```
public boolean validateAction() {
    // make sure that we can get a 'to' email address
    if (!isDefinedInput("to")) {
        error(Messages.getErrorString("Email.ERROR_0001_TO_NOT_DEFINED",
              getActionName()));
        return false;
    }
    // make sure that we can get a subject for the email
    if (!isDefinedInput("subject")) {
        error(Messages.getErrorString("Email.ERROR_0002_SUBJECT_NOT_DEFINED",
              getActionName()));
        return false;
    }
```

```
        // make sure that we have either a plain text or html message for the email
        if (!isDefinedInput("message-plain") && !isDefinedInput("message-html")) {
            error(Messages.getErrorString("Email.ERROR_0003_BODY_NOT_DEFINED",
                  getActionName()));
            return false;
        }
        return true;
    }
```

Notice that this method does not check for the optional parameters such as 'cc', 'bcc', or email attachments but only checks for the minimum parameters that are required for the component to execute successfully.

You do not have to call for each individual resource, input, or output. If you prefer, you can work with java.util.Set objects that contain the names of the available items. These sets can be obtained by calling getResourceNames(), getInputNames(), and getOutputNames(). For example

```
    Set inputNames = getInputNames();
    if( !inputNames.contains( "param1" ) && !inputNames.contains( "param2" ) ) {
        error( "I need both param1 and param2" );
        return false;
    }
```

validateSystemSettings

This page last changed on Dec 04, 2006 by mlowery.

This method allows your component to verify that any system settings it needs in order to operate are set correctly. For example, the Kettle component (org.pentaho.plugin.kettle.KettleComponent) checks it system settings to see if it is configured for a file-based or RBDMS-based repository. In another example, the email component (org.pentaho.plugin.email.EmailComponent) uses its system settings to connect to your email server.

System-wide settings for components should be stored in XML documents in the 'system' folder in the Pentaho solution root folder. System settings are ones that will be the same for every instance of your component and that won't change while the server or application is running (unless manually changed by an administrator).

In the system folder you will see subdirectories such as 'kettle', 'quartz', and 'smtp-email'. If your component needs system settings you can create a directory to store your settings in and place an XML file in the directory. The XML file can be named anything that you like. Within the XML document you can arrange the settings in any format that you like. The settings file for the email component provides a good example of a settings file. Within the root solution folder the path to the email settings file is 'system/smtp-email/email_config.xml'.

```
<email-smtp>

   <!-- The values within <properties> are passed directly to the JavaMail API.
   For a list of valid properties see
   http://java.sun.com/products/javamail/javadocs/index.html -->
   <properties>
      <!-- This is the address of your SMTP email server for sending email. e.g.
           smtp.pentaho.org -->
      <mail.smtp.host></mail.smtp.host>

      <!-- This is the port of your SMTP email server. Usually this is 25.
           For GMail this is 587 -->
      <mail.smtp.port>25</mail.smtp.port>

      <!-- The transport for accessing the email server. Usually this is smtp.
           For GMail this is smtps -->
      <mail.transport.protocol>smtp</mail.transport.protocol>

      <!-- Usually this is 'false'. For GMail it is 'true' -->
      <mail.smtp.starttls.enable>false</mail.smtp.starttls.enable>

      <!-- Set to true if the email server requires the sender to authenticate -->
      <mail.smtp.auth>true</mail.smtp.auth>

      <!-- This is true if the email server requires an SSL connection.
           Usually 'false'. For GMail this is 'true' -->
      <mail.smtp.ssl>false</mail.smtp.ssl>

      <!-- Output debug information from the JavaMail API -->
      <mail.debug>false</mail.debug>
   </properties>

   <!-- This is the default 'from' address that emails from the Pentaho BI Suite
        will appear to come from e.g. joe.pentaho@pentaho.org -->
   <mail.from.default>joe.pentaho@pentaho.org</mail.from.default>

   <!-- This is the user id used to connect to the email server for sending email
        It is only required if email-authenticate is set to true
        This is never sent or shown to anyone -->
   <mail.userid></mail.userid>

   <!-- This is the password used to connect to the email server for sending email
        It is only required if email-authenticate is set to true
```

```
          This is never sent or shown to anyone -->
     <mail.password></mail.password>

   </email-smtp>
```

The email component accesses these settings by calling PentahoSystem.getSystemSetting(). For example:

```
String mailhost = PentahoSystem.getSystemSetting("smtp-email/email_config.xml",
"mail.smtp.host", null);
```

The getSystemSetting() method finds an entry in a system setting file and returns it to the component. The component does not need to know where the system settings files are located.

Internal API

If one of the three recommended ways of creating a component above has been used, the internal component API will be available. This API provides many methods that enable components to interact with the process within which they are running.

- Handling Inputs
- Handling Outputs
- Handling Resources
- Handling User Interaction

Handling Inputs

Internal API Handling Outputs

The methods provide a mechanism for getting inputs into a component.

- getDataSource()
- getInputBooleanValue()
- getInputLongValue()
- getInputNames()
- getInputParameter()
- getInputStream()
- getInputStringValue()
- getInputValue()
- isDefinedInput()

getDataSource()

This page last changed on Dec 04, 2006 by mlowery.

Handling Inputs getInputBooleanValue()

Parameters:	String
Result:	javax.activation.DataSource

Returns a DataSource for the specified input. This method only works for inputs that represent items stored in the Content Repository. It is important to call isDefinedInput() or getInputNames() before asking for access to an output as an error will occur if you ask for an output that has not been defined in the action sequence.

getInputBooleanValue()

This page last changed on Dec 01, 2006 by bseyler.

getDataSource() Handling Inputs getInputLongValue()

Parameters:	String	inputName
	boolean	defaultValue
Result:	boolean	

Returns the value of an input as a boolean. If the value of the input is not a valid boolean the defaultValue will be returned. It is important to call isDefinedInput() or getInputNames() before asking for access to an output as an error will occur if you ask for an output that has not been defined in the action sequence.

getInputLongValue()

This page last changed on Dec 01, 2006 by bseyler.

getInputBooleanValue() Handling Inputs getInputNames()

Parameters:	String	inputName
	long	defaultValue
Result:	long	

Returns the value of an input as a long. If the value of the input is not a valid long the defaultValue will be returned. It is important to call isDefinedInput() or getInputNames() before asking for access to an output as an error will occur if you ask for an output that has not been defined in the action sequence.

getInputNames()

This page last changed on Dec 01, 2006 by bseyler.

getInputLongValue() Handling Inputs getInputParameter()

Parameters	none
Result:	java.util.Set

Returns the names of the available inputs as defined in the action sequence. Components should use this method (or isDefinedInput()) to validate that the inputs the component is expecting are available before trying to use them.

getInputParameter()

This page last changed on Dec 01, 2006 by bseyler.

getInputNames() Handling Inputs getInputStream()

Parameters:	String	parameterName
Result:	org.pentaho.core.runtime.IActionParameter	

Returns the IActionParameter that represents an input. Using this the IActionParameter object you can:

- Check to see if a prompt is already pending for this input.
- Find out if the input value has a list of valid values.
- Find out if the input value has been specified.
- Find out if the input has a default value.
- Find out the type of the input.
- Find out the value of the input.

It is important to call isDefinedInput() or getInputNames() before asking for access to an output as an error will occur if you ask for an output that has not been defined in the action sequence.

getInputStream()

Parameters:	String	inputName
Result:	java.io.InputStream	

Returns an InputStream for a specified input. This only works if the input value is an IContentItem. It is important to call isDefinedInput() or getInputNames() before asking for access to an output as an error will occur if you ask for an output that has not been defined in the action sequence.

getInputStringValue()

This page last changed on Dec 01, 2006 by bseyler.

getInputStream() Handling Inputs getInputValue()

Parameters:	String	inputName
Result:	String	

Returns the value of an input as a String. If the value is not a String, object its toString() method will be called. It is important to call isDefinedInput() or getInputNames() before asking for access to an output as an error will occur if you ask for an output that has not been defined in the action sequence.

getInputValue()

This page last changed on Dec 05, 2006 by bseyler.

getInputStringValue() Handling Inputs isDefinedInput()

Parameters:	String	inputName
Result:	Object	

Returns the value of an input as an Object. It is important to call isDefinedInput() or getInputNames() before asking for access to an output as an error will occur if you ask for an input that has not been defined in the action sequence.

isDefinedInput()

This page last changed on Dec 01, 2006 by bseyler.

getInputValue() Handling Inputs

Parameters	String	inputName
Result:	boolean	

Returns whether the specified input has been defined in the action sequence. Components should use this method (or getInputNames()) to validate that the inputs the component is expecting are available before trying to use them.

Handling Outputs

This page last changed on Dec 01, 2006 by bseyler.

Handling Inputs Internal API Handling Resources

These are methods for finding out information about the output from a component. Including but not limited to the content and name.

- getDefaultOutputStream() - deprecated
- getOutputContentItem()
- getOutputContentItem() - deprecated
- getOutputNames()
- getOutputPreference()
- getOutputStream()
- isDefinedOutput()
- setOutputMimeType()
- setOutputValue()

getDefaultOutputStream() - deprecated

Handling Outputs getOutputContentItem()

Parameters:	none
Result:	java.io.OutputStream

This method is deprecated and the system will log a warning in the audit log and server log if you use it. It is provided for backwards compatibility only. New components should not use this method. Use getOutputStream() instead.

getOutputContentItem()

getDefaultOutputStream() - deprecated | Handling Outputs | getOutputContentItem() - deprecated

Parameters:	String	outputName
Result:	org.pentaho.core.repository.IContentItem	

Returns an IContentItem object for the specified output. The content item can be used to access the output stream for the specified output. It is important to call isDefinedOutput() or getOutputNames() before asking for access to an output as an error will occur if you ask for an output that has not been defined in the action sequence.

getOutputContentItem() - deprecated

This page last changed on Dec 01, 2006 by bseyler.

getOutputContentItem() Handling Outputs getOutputNames()

Parameters:	none
Result:	org.pentaho.core.repository.IContentItem

This method is deprecated, and the system will log a warning in the audit log and server log if you use it. It is provided for backwards compatibility only. New components should not use this method. Use getOutputStream() and pass in the name of a specific output.

getOutputNames()

This page last changed on Dec 01, 2006 by bseyler.

getOutputContentItem() - deprecated | Handling Outputs | getOutputPreference()

Parameters:	none
Result:	java.util.Set

Returns the names of the available outputs as defined in the action sequence. Components should use this method (or isDefinedOutput()) to validate that the outputs the component is expecting are available before trying to use them.

getOutputPreference()

This page last changed on Dec 01, 2006 by bseyler.

getOutputNames() Handling Outputs getOutputStream()

Parameters:	none
Result:	int

A component can use this method to find out what kind of output the Runtime Context is most interested in. The valid values are stored in org.pentaho.core.solution.IOutputHandler and are
OUTPUT_TYPE_PARAMETERS
OUTPUT_TYPE_CONTENT
OUTPUT_TYPE_DEFAULT

getOutputStream()

Parameters:	String	outputName
	String	mimeType
	String	extension
Result:	java.io.OutputStream	

Gets an output stream for the requested output. Calling this will create a file in the Content Repository. It is important to call isDefinedOutput() or getOutputNames() before asking for access to an output as an error will occur if you ask for an output that has not been defined in the action sequence.

isDefinedOutput()

This page last changed on Dec 01, 2006 by bseyler.

getOutputStream() Handling Outputs setOutputMimeType()

Parameters:	String	outputName
Resut:	boolean	

Returns whether the specified output has been defined in the action sequence. Components should use this method (or getOutputNames()) to validate that the outputs the component is expecting are available before trying to use them.

setOutputMimeType()

This page last changed on Dec 01, 2006 by bseyler.

isDefinedOutput() Handling Outputs setOutputValue()

Parameters:	String	outputName
Result:	String	mimeType

This sets the mime type of the specified output. This is used for streamed outputs. Some standard mime types are:

- HTML: text/html
- Plain text: text/plain
- XML: text/xml

setOutputValue()

This page last changed on Dec 01, 2006 by bseyler.

setOutputMimeType() Handling Outputs

Parameters	String	outputName
	Object	value
Result:	none	

Sets the value for an output. This call is used to set atomic output values such as a String.

It is important to call isDefinedOutput() or getOutputNames() before asking for access to an output as an error will occur if you ask for an output that has not been defined in the action sequence.

Handling Resources

This page last changed on Dec 01, 2006 by bseyler.

Handling Outputs | Internal API | Handling User Interaction

These methods provide resource handling and feedback parameter management.

- getResource()
- getResourceAsString()
- getResourceDataSource()
- getResourceInputStream()
- getResourceNames()
- isDefinedResource()

getResource()

This page last changed on Dec 05, 2006 by bseyler.

Handling Resources getResourceAsString()

Parameters:	String	resourceName
Result:	org.pentaho.core.solution.IActionResource	

It is important to call isDefinedResource() or getResourceNames() before asking for access to an output as an error will occur if you ask for an input that has not been defined in the action sequence.

getResourceAsString()

This page last changed on Dec 01, 2006 by bseyler.

getResource() Handling Resources getResourceDataSource()

Parameters:	org.pentaho.core.solution.IActionResource	resource
Result:	String	

Returns the contents of the specific resource as a String. It is important to call isDefinedResource() or getResourceNames() before asking for access to an output as an error will occur if you ask for an output that has not been defined in the action sequence.

```
if( isDefinedResource("message-template") ) {
    String template = getResourceAsString("message-template" );
}
```

getResourceDataSource()

Parameters:	org.pentaho.core.solution.IActionResource	resource
Result:	javax.activation.DataSource	

Returns the contents of the specified resource as a DataSource object. It is important to call isDefinedResource() or getResourceNames() before asking for access to an output as an error will occur if you ask for an output that has not been defined in the action sequence.

```
if( isDefinedResource("message-template") ) {
    DataSource source = getResourceDataSource("message-template" );
}
```

getResourceInputStream()

This page last changed on Dec 01, 2006 by bseyler.

getResourceDataSource() Handling Resources getResourceNames()

Parameters:	org.pentaho.core.solution.IActionResource	resource
Result:	java.io.InputStream	

Returns an input stream to the contents of the specified resource. It is important to call isDefinedResource() or getResourceNames() before asking for access to an output as an error will occur if you ask for an output that has not been defined in the action sequence.

```
if( isDefinedResource("message-template") ) {
    InputStream stream = getResourceInputStream("message-template" );
}
```

getResourceNames()

Parameters:	none
Result:	java.util.Set

Returns the names of all the available resources defined in the action sequence. This is useful for examining the names of the available resources so the component does not request resources that are not available.

isDefinedResource()

This page last changed on Dec 01, 2006 by bseyler.

getResourceNames() Handling Resources

Parameters:	String	resourceName
Result:	boolean	

Returns whether the specified resource has been defined in the action sequence.

Handling User Interaction

This page last changed on Dec 01, 2006 by bseyler.

Handling Resources Internal API

These methods are used to create prompts that will be shown to the user. They are useful for automatically generating parameter pages. The template that is used to create the layout of the page is specified by the action sequence and can be either an HTML template or an XSLT template. If a template is not specified a default XSLT (system/custom/xsl/DefaultParameterForm.xsl in the solution folders) is used.

- createFeedbackParameter() - 3 Parameter
- createFeedbackParameter() - 5 Parameter
- createFeedbackParameter() - 7 Parameter
- feedbackAllowed()
- getFeedbackOutputStream()
- isPromptPending()
- promptNeeded()
- promptNow()
- setFeedbackMimeType()

createFeedbackParameter() - 3 Parameter

Parameters:	org.pentaho.core.runtime.SelectionMapper	selMap
	String	fieldname
	Object	defaultValues
Result:	none	

TODO Insert explanation.

createFeedbackParameter() - 5 Parameter

This page last changed on Dec 01, 2006 by bseyler.

createFeedbackParameter() - 3 Parameter | Handling User Interaction | createFeedbackParameter() - 7 Parameter

Parameters:	String	fieldname
	String	displayName
	String	hint
	String	defaultValue
	boolean	visible
Result:	none	

This creates a parameter prompt that presents a text box to the user.

fieldname: specifies the id of the form element for this prompt. This id will appear on the request when the user clicks to submit their selections. If the action sequence does not define an input from the request that matches this id, the component will never receive the value selected by the user, and an infinite loop can occur where the user is continually prompted for the same thing without end.

displayName: the text that the user will see as the label for the parameter.

hint: a help tip that the user will see next to the parameter.

defaultValues: the default value that will be automatically displayed for the user when they see the prompt page.

visible: an flag that determines whether the user can see the form control. If this is set to 'false' a hidden form field is created.

createFeedbackParameter() - 7 Parameter

This page last changed on Dec 01, 2006 by bseyler.

createFeedbackParameter() - 5 Parameter | Handling User Interaction | feedbackAllowed()

Parameters:	String	fieldname
	String	displayName
	String	hint
	Object	defaultValues
	java/util/Set.html@javajava.util.Set	values
	java.util.Map	dispNames
	String	displayStyle
Result:	none	

This creates a parameter prompt that will be present the user with a collection of items to choose from. The component has to specify the list of options. A good example of this is the print component (org.pentaho.plugin.print.PrintComponent) which creates a list of printers available to the component and then calls createFeedbackParameter() to let the user select a printer.

fieldname: specifies the id of the form element for this prompt. This id will appear on the request when the user clicks to submit their selections. If the action sequence does not define an input from the request that matches this id, the component will never receive the value selected by the user, and an infinite loop can occur where the user is continually prompted for the same thing without end.

displayName: the text that the user will see as the label for the parameter.

hint: a help tip that the user will see next to the parameter.

defaultValues: an object that stores the default value(s) that will be automatically selected for the user when they see the prompt page. This object can be a String, or String array. If the control type specified in displayStyle is not capable of selecting multiple items but a String array is provided for the defaultValues, the single item selected cannot be predicted.

values: an ordered list of values that the user will select from. This collection stores the values that you want the component to receive back when the user selects something. If you want the user to see the items described with different names to the ones in this list the component must supply a dispNames map.

dispNames: a Map that stores a user-friendly description for each value. If the map does not store a description for a given value the value will be shown to the user. If there are no user-friendly descriptions for any values you can pass a null for this parameter.

displayStyle: specifies the control that should be used to present the choices to the user. The allowed values are:

Style Tag	Control
radio:	Radio buttons
select:	A dropdown select/combo box
list:	A singl select list box
list-multi:	A multi-select list box
check-multi:	A multi-select collection of check boxes
check-multi-scroll:	A scrolling, multi-select set of check boxes
check-multi-scroll-2-column:	A scrolling, two-column, multi-select set of check boxes
check-multi-scroll-3-column:	A scrolling, three-column, multi-select set of check boxes
check-multi-scroll-4-column:	A scrolling, four-column, multi-select set of check boxes

Example
The printer component creates a list of printer names for the user to select from. It uses the last printer used as the default selection. It does not provide a hint to the user or provide a mapping of display names for the printer names. It specifies that a select list (combo box) should be used to get the users input.

```
ArrayList values = new ArrayList();
for (int i = 0; i < services.length; i++) {
    String value = services\[i\].getName();
    values.add(value);
}

createFeedbackParameter("printer-name", "Printer Name", "",
lastPrinterName, values, null, "select");
promptNeeded();
```

feedbackAllowed()

Parameters:	none
Result:	boolean

This returns true if the component is allowed to ask the use for parameters. If the request to execute an action sequence originated from a web application, this will return 'true'. If the request comes from a web service client, a scheduler, or a workflow engine there is no user to request anything from, and this method will return 'false';

getFeedbackOutputStream()

This page last changed on Dec 01, 2006 by bseyler.

feedbackAllowed() Handling User Interaction isPromptPending()

Parameters:	none
Result:	java.io.OutputStream

This is used in cases where a component needs to provide feedback to the user about a problem other than needing input. For example, the email component uses this method when the email server settings have not been defined to present an informative message to the user about how to fix the problem.

isPromptPending()

This page last changed on Dec 01, 2006 by bseyler.

getFeedbackOutputStream() Handling User Interaction promptNeeded()

Parameters:	none
Result:	boolean

This enables the component to find out if other components have requested parameters from the user. If this method returns true a component should not attempt to execute or create content but should only determine if it needs to prompt for parameters itself.

promptNeeded()

isPromptPending() Handling User Interaction promptNow()

Parameters:	none
Result:	none

This informs the Runtime Context that the user needs to be prompted for parameters that the component needs. Other components in the action sequence will be executed to see if they also have parameters to for which to prompt. This enables the system to generate a single input form for users instead of presenting the user with a succession of input forms. For example, if the first component in an action sequence is a report component that needs a department id and the second component in the sequence is a print component that needs a printer name, the user will be presented with a single form that lets them select a department and a printer.

promptNow()

promptNeeded() Handling User Interaction setFeedbackMimeType()

Parameters:	none
Result:	none

This informs the Runtime Context that the user needs to be prompted for parameters that the component needs and that components later in the action sequence should not be given the opportunity to provide parameters. This can be used for security purposes and can be used to prevent needless processing. For example consider an action sequence with two action definitions. The first is a Web Service business rule that needs to prompt for a customer id if it is not provided on the request, and the second is a report component that takes the result of the first component as its input. If the customer id has not been provided, the first component needs to prompt for it. In this case, there is no point executing the report component (causing the report definition to be parsed) to see if it needs to prompt for any parameters because the context of the action sequence tells us it does not.

setFeedbackMimeType()

This page last changed on Dec 01, 2006 by bseyler.

promptNow() Handling User Interaction

Parameters:	String	mimeType
Result:	none	

This sets the mime type of the feedback page. Currently this needs to be set to 'text/html'. In the future mime types for other devices could be supported.

Why Create a Pentaho Component?

This page last changed on Dec 04, 2006 by mlowery.

Internal API Building Components

The infrastructure of the Pentaho BI Suite is a lightweight, but flexible and powerful, framework for executing Business Intelligence related components. The infrastructure allows components to be sequenced together and the outputs of one to be passed as an input to another. The components can be used individually or can be used together to create an endless number of results. Execution of components can be controlled using Java APIs, Web Services, schedulers, and workflow engines.

The infrastructure is efficient and compact and can be deployed as a library into a Java application.

By creating a single Java class that integrates the features of a library or engine into the Pentaho component layer, the new features can immediately be combined with the features of the existing components, and all of the platforms execution methods become available. For example, once a new component is created:

- It can be scheduled
- It can be executed as a web service
- It can be sequenced with other components and pass data to and accept data from other components
- It can prompt users for parameters it needs
- Its usage is automatically audited and logged
- BI Components are easy to create and easy to deploy into the Pentaho BI Suite.