Contents

Solution Oriented Approach .................................................................................................................................1
  Introduction ..........................................................................................................................................................1
  Workflow At The Core ......................................................................................................................................1
  Service-Oriented Architecture (SOA) ..................................................................................................................1
  Process Integration .............................................................................................................................................2
  Rules, Rules, Rules ............................................................................................................................................2
  Business Intelligence / Business Process Boundary ............................................................................................2
  Case Study ..........................................................................................................................................................2
  Solution 1: Give them what they ask for ............................................................................................................2
  Solution 2: Give them what they need ................................................................................................................3
  Conclusion ..........................................................................................................................................................3

Building Solutions ..............................................................................................................................................5
  Audience ............................................................................................................................................................5
  Introduction to the Solution Engine ....................................................................................................................5
  The Action Sequence .........................................................................................................................................9
  A More Complicated Example ...........................................................................................................................10
  Are There Tools To Help? ..................................................................................................................................11

Executing a Solution ............................................................................................................................................12
  URL ..................................................................................................................................................................12
  Web Service Call .............................................................................................................................................12
  Java Call ..........................................................................................................................................................13

Parameters ...........................................................................................................................................................13

Components ..........................................................................................................................................................13

Integrating your own reports ...............................................................................................................................14
  Overview ...........................................................................................................................................................14
  Introduction ......................................................................................................................................................14
  General Assumptions and Conventions ............................................................................................................14
  Integrating BIRT Reports ................................................................................................................................14
    Eclipse BIRT Report Definitions ....................................................................................................................14
    Creating the Action Sequence ........................................................................................................................16
    JDBC Driver Setup .......................................................................................................................................18
    Verifying BIRT Integration into Pentaho Platform .........................................................................................18
  Integrating JasperReports ..................................................................................................................................20
    JasperReports Report Definitions ...................................................................................................................20
    Creating the Action Sequence ........................................................................................................................21
    JDBC Driver Setup .......................................................................................................................................24
    Verifying JasperReports Integration into Pentaho Platform .........................................................................25

Charting ..............................................................................................................................................................26
  ChartComponent ..............................................................................................................................................26
Category: DatasetComponent ................................................................. 29

Scheduler ...................................................................................................................... 30
  Actions .......................................................................................................................... 30
  Triggers ......................................................................................................................... 34

Misfires .......................................................................................................................... 36

Datasources ................................................................................................................. 39
  SQLLookupRule ......................................................................................................... 41
  XQueryLookupRule .................................................................................................... 42

Appendix B .................................................................................................................. 47
  Action Sequence Data Types ..................................................................................... 47
  Action Sequence Resource Types .............................................................................. 49

Appendix C .................................................................................................................. 50
  Component Definitions ............................................................................................... 50
    HelloWorldComponent ......................................................................................... 50
      component-name: org.pentaho.component.HelloWorldComponent.java .......... 50
    EmailComponent ................................................................................................. 51
      component-name: org.pentaho.component.EmailComponent.java ................... 51
    JavascriptRule ...................................................................................................... 52
      component-name: org.pentaho.component.JavascriptRule.java ......................... 52
    SQLLookupRule ................................................................................................. 53
      component-name: org.pentaho.component.SQLLookupRule.java ..................... 53
    PrintComponent ................................................................................................. 54
      component-name: org.pentaho.components.PrintComponent ......................... 54
    JobSchedulerComponent ....................................................................................... 55
      component-name: org.pentaho.component.JobSchedulerComponent.java ....... 55
    SchedulerAdminComponent ................................................................................. 55
      component-name: org.pentaho.component.SchedulerAdminComponent.java .... 55
    BIRTReportComponent ......................................................................................... 55
      component-name: org.pentaho.birt.BIRTReportComponent.java .................. 55
    JasperReportsComponent ....................................................................................... 55
      component-name: org.pentaho.jasper.JasperReportsComponent.java ............. 55
    ContentRepositoryCleaner ..................................................................................... 55
      component-name: org.pentaho.component.ContentRepositoryCleaner.java ....... 55
Solution Oriented Approach

Introduction

In our many years helping customers create reports and analytical systems, we have encountered a similar situation many times. The scenario is always different, but the basic need is always the same: a report is delivered or a particular situation is encountered in the data and something specific needs to happen - a decision must be made, causes must be discovered, or a process must be started. In these cases the information presentation, analysis, and delivery (BI) is a part of a larger process. This process exists to solve the business problem: it is the solution.

To clarify:

- Often the solution to a business problem is a process that includes Business Intelligence.
- Therefore: the Business Intelligence, by itself, is not the solution to the problem.
- If Business Intelligence is part of the process, the Business Intelligence tools are, unavoidably, also part of the process.
- A Business Intelligence tool that does not understand processes, or how to be part of one, will be hard to integrate into the solution.

The Pentaho BI Platform is the first process-centric and solution-oriented Business Intelligence platform.

Sure we can throw in a little bold text and make it look grand but how do we back up a statement like that when other BI providers are claiming to be adding process-centric features?

Workflow At The Core

The Pentaho BI Platform uses a workflow engine to execute activities. This workflow engine uses a standard language, XML Process Definition Language (XPDL), to execute the activities within the system. For example: you want to run a query to find out which departments are over budget, run a budget report for each of those departments, and finally email each report to the department manager. This is defined as XPDL and executes as a business process. This is how all activities are run within the platform.

- The platform understands the nature of processes because everything in it is one.
- The processes are defined in a standard process definition language that is externally viewable, editable, and customizable.
- There is no hidden business logic.

The platform is built on processes and process definitions.

Service-Oriented Architecture (SOA)

This is rapidly becoming a meaningless term with every application that accepts URLs claiming to have a SOA. When you design a system with a workflow engine as the conductor and director every activity in the system, every step of each process, every bubble in your process diagram must be implemented as a standalone, re-usable component that can be directed to execute the activity required. This is not just an SOA, this is a Service-Implemented Architecture (SIA). Every activity in every process can be a web service because all activities only ever execute as services. They know no other invocation. The three rules to web services success are: invocation, invocation, invocation.

Services are the building blocks of automated business processes.
Process Integration

Every process and activity in the Pentaho BI Platform executes as a service. If you want to call a process or activity defined in the platform from a process executing in another system, you can. It’s easy.

Every activity in the system understands how to be part of another process.

Rules, Rules, Rules

The definition of the platform processes are externally defined, but what about the rules that govern the workflow? XPDL has built in support for complex routing control, and we have added support for multiple rules engines so business logic can be integrated easily into the processes. Multiple rules engines are supported and required because it is unlikely the logic for every decision in every process can be defined easily by only one rules engine.

For example, the business rule to determine the credit analyst for a customer might be best described three different ways in three implementations:

- A simple piece of script: if (customerNo < 3000 ) return ‘Bob’ else return ‘Mary’
- A complex chaining algorithm that bases the decision on the customer’s current sales pipeline, service level agreement, lifetime value, payment history, industry segment, and location
- A call to a database or ERP system to lookup the analyst for the customer

If the credit analyst for each customer is stored as a record in an ERP system, trying to maintain the rule in a different system will be a redundant effort with additional cost, additional risk, and no added value.

Flexible business rules are a critical part of automated business processes.

Business Intelligence / Business Process Boundary

The line between business intelligence and business processes is flexible in the Pentaho BI Platform. This is because the line between business intelligence and business processes is blurry and should be up to you. If you have a BI platform that clearly defines the boundaries between it and your other systems, you probably have an application silo that is hard to integrate the way you need it to. It is your processes, your data, and your software.

- The engine executing processes within the platform is a full-featured workflow engine
- The Pentaho BI Platform includes multiple rules engines
- The Pentaho BI Platform activities are easily integrated into other processes
- The roadmap includes manual and data entry tasks by users

Case Study

Problem: When an employee shows up for work at a healthcare organization with an expired license either
- It is noticed and a more costly agency worker must replace the employee until their license is renewed
- It is not noticed in which case a patient safety hazard and a liability risk occurs

Business goals: increase patient safety, reduce liability of unlicensed employees, and reduce money spent on agency staff covering for unlicensed employees.

Current process: Each manager maintains a list of license expirations for their department.

Proposed ‘solution’: Scheduled execution of a report from a central database that lists, by department, licenses held by each employee, and the expiration date of their current licenses.

Solution 1: Give them what they ask for

Create a 50 page report and deliver it to each department once a month.

Resulting Business Process:
- Running of report is not audited. If it does not happen when expected how long before someone realizes?
Managers in each department are required to read the report and filter the information. Reports get lost, managers take vacations, and dates get misread.

When managers spot upcoming license expirations they leave a note in the employee's mail box. Notes get lost or placed in wrong boxes.

Employees try to schedule preparation, application and certification time. Schedule conflicts arise, preparation suffers.

Employees fail certification with no time for further preparation or certification before license expiration.

This solution is incomplete because it only automates the information delivery, it does not help the real process that has to occur. The business goal is reached at best as a by-product of the reporting solution.

**Solution 2: Give them what they need**

Create business rules that determine the lead time required for adequate preparation for each type of license and escalation paths for problem cases.

Run an audited report every day or week that lists those employees within their lead time. For each employee, initiate a defined license renewal process:

1. Deliver the information electronically to both manager and employee
2. Require electronic acknowledgement from both
3. Direct employee to schedule preparation time
4. Direct manager to approve schedule
5. Require employee to enter certification test date
6. Escalate warnings if insufficient re-test time has been allowed
7. Require manager to validate new license
8. Deliver notifications on certification failure to manager and scheduling application.

- Provide on-line, real-time reporting on the license renewal process
- Produce audit reports of monthly and quarterly performance

This solution solves the business problem.

**Conclusion**

In order to deliver this solution you need reporting and analysis tools that:

- Support the business rules needed
- Audit report execution and delivery of information
- Integrate seamlessly with a workflow system

You also need a workflow / business process engine that:

- Handles time-based escalations
- Audits execution of activities within the process
- Integrates with reporting and analysis tools

You also need to provide real-time and historical process performance reports

This is the Pentaho BI Platform.

The Pentaho BI Platform is uniquely process-centric and solution-oriented.

- It is process-centric because it is built ground-up to be process-based.
- It is solution-oriented because the solution for many business problems is a process, and the platform includes all the major components required to implement process-based solutions.

The rest of this document describes how the engines and components of the Pentaho BI Platform can be used to create solutions to your Business Intelligence needs.
Building Solutions

The previous section described our Solution Oriented approach and its benefits. The rest of this document provides a technical description of the Solution Engine and the documents required to build solutions in the Pentaho BI Platform.

Audience

This section is intended for people interested in building solutions and creating content and is also valuable for anyone who needs to interface with or develop portions of the Pentaho BI Platform. Prior to reading this document, you should have read and understood the “Pentaho Technical White Paper” which is can be downloaded from SourceForge - http://sourceforge.net/projects/pentaho.

This document has examples and references to files distributed with the Pentaho BI Platform Pre-Configured Install (PCI). It is recommended that the PCI be downloaded and installed. It is available at SourceForge http://sourceforge.net/projects/pentaho.

Introduction to the Solution Engine

The Solution Engine is the focal point for activity within the Pentaho BI Platform. It “sits” between the outside world – Web Client, Services, System Monitor etc. and the Component Layer. See Figure 1 – Architecture Diagram. Requests to do work come into the solution engine and are routed to the appropriate component or components for execution. The following terms will be used in the discussion of the Solution Engine:

- **Solution** - A Solution consists of a collection of documents that collectively define the processes and activities that are the system’s part in implementing a solution to a business problem. These documents include Action Sequence Definitions, workflow process definitions, report definitions, images, rules, queries etc.

- **Solution Repository** - The location where solution definitions and the metadata they rely on is stored and maintained. Requests made to the platform to have actions executed rely on the action being defined in the Solution Repository.

- **Solution Engine** - The engine that retrieves the definition of an action from the Solution Repository and directs its execution.

- **Component** - The component layer provides a standard interface between the solution engine and the application that executes business logic. A component may contain all of the code required to perform a task or may just be an interface to another application or system. Data and instructions to the component are provided via an Action Definition.

- **Action Definition** - An XML definition specifying the parameters, resources and settings required for the execution of a task within a single component. The Action Definition defines which component to call, what data to pass into and receive from the component and any component specific information required. An action definition is not a stand alone document; it is a part of an Action Sequence Definition.

- **Action Sequence Definition** - An XML document that defines the interaction between one or more Action Definitions. It defines the smallest complete task that the Solution Engine can perform. When the Solution Engine is told to execute - it is given an Action Sequence document to execute. The execution of the Action Sequence can be completed autonomously or may execute as part of another Action Sequence. Action Sequence Definitions are stored in the Solution Repository.

- **Runtime Context** - Action Sequences are transformed from XML by the solution engine into objects that are interpreted by the Runtime Context. The Runtime Context maintains a contract
between the Solution Engine and the Action Sequence and enforces a contract between the Action Sequence and the components.

The architecture diagram below shows how the Solution Engine, Solution Repository, and components fit into the architecture. The Solutions, Action Definitions and Action Sequence Definitions are stored in the Solution Repository. The Solution Engine creates Runtime Contexts every time a request is received.

**Figure 1 - Pentaho Architecture Diagram**
Solutions in More Detail
A solution is not a single document; it's a collection of documents. It's a logical grouping of Action Sequence Definitions and the resources they require. The grouping is maintained by the Solution Repository. You can see the structure of the solution repository by navigating to the pentaho-solution directory in the top level PCI install directory. The default location is: /preconfigured-install/pentaho-solutions.

In the pentaho-solutions directory, there are 2 predefined solutions test and samples. Figure 2 shows the HelloWorld.xaction Action Sequence Document selected in the samples solution.

**Figure 2 - Solution Repository**

![Solution Repository Diagram]

Notice that there are subdirectories under the solution test. Action Sequence documents and resources may be located anywhere below the solution directory (in this case - test) and with any level of nesting of folders. This provides a way to logically group content. Within the test solution, Action Sequences are grouped by component type; email, reporting, workflow etc. For a production system the grouping may be by department or role or whatever structure makes sense for that solution.

To locate an Action Sequence in the repository, use the three part address: solution id, path and action sequence name. In the case of the HelloWorld Action Sequence the address is: samples, getting-started, HelloWorld.xaction.

The two solutions test and samples are available with the PCI or can be downloaded from SourceForge. The test solution is used to verify that the different components in the system are set up and function correctly. The samples solution has... well, samples. The samples are a good starting point for learning and building new Action Sequences. The Action Sequences referenced in this document are located in samples, getting-started.
The Action Sequence

The Action Sequence is an XML document that defines the smallest complete task that the solution engine can perform. But why an action sequence and not an Action you may ask. Way back in Pentaho history (June I believe) we had action documents that performed a single action. It turns out, however, that in many cases there needs to be two or three things done within an action, like data conversion or message formatting. The Action Sequence can contain multiple component calls and also has the ability to loop.

This is a listing of the HelloWorld.xaction Action Sequence Document.

```
<action-sequence>
  <name>HelloWorld.xaction</name>
  <title>Hello World Action Sequence</title>
  <version>1</version>
  <logging-level>DEBUG</logging-level>

  <documentation>
    <author>Joe Pentaho</author>
    <description>The most basic Action Sequence Document</description>
    <help>Hello World demonstrates the most basic Action Sequence document.
    It uses the Hello World Component.</help>
  </documentation>

  <inputs/>
  <outputs/>
  <resources/>

  <actions>
    <action-definition>
      <action-name>Hello World Action</action-name>
      <action-inputs/>
      <action-outputs/>
      <logging-level>DEBUG</logging-level>
      <component-name>org.pentaho.component.HelloWorldComponent</component-name>
      <component-definition>
        <quote>(2B || !2B) That is the question</quote>
      </component-definition>
    </action-definition>
  </actions>
</action-sequence>
```

Some of the XML Nodes are self explanatory like author and description. In this section, we will cover the most important nodes, for a complete explanation of the Action Sequence – See Appendix A

The Action Sequence has 2 major parts, the main document and the actions node. The main document defines the Action Sequences interface with the outside world. It specifies the inputs and resources that are required by the components referenced by the document and it defines the outputs that will be available when all referenced components have executed. This example has no inputs, no outputs and uses no resources. The actions node contains the Action Definitions that execute.

There is a one-to-one correspondence of Action Definitions and components. The HelloWorld example has one action Definition, it is named “Hello World Action”, takes no inputs and generates no outputs. The component that will be loaded to respond to the request is the Java class org.pentaho.component.HelloWorldComponent.

When the component is loaded, the XML in component-definition is passed in as are the available input parameters.

The HelloWorld Component is not very exciting, it basically just outputs “Hello World. “ and appends whatever is in the quote node to the console.
A More Complicated Example
This is a listing of the Example1.xaction Action Sequence Document.

<action-sequence>
  <name>Example1.xaction</name>
  <inputs>
    <region type="string">
      <default-value>Central</default-value>
      <sources>
        <request>REGION</request>
        <session>aRegion</session>
      </sources>
    </region>
    <from type="string">
      <default-value>joepentaho@pentaho.org</default-value>
    </from>
    <subject type="string">
      <default-value>Pentaho Example1</default-value>
    </subject>
    <message-plain type="string">
      <default-value>This is an email from the Pentaho BI Platform - Example1</default-value>
    </message-plain>
  </inputs>
  <outputs/>
  <resources/>
  <actions>
    <action-definition>
      <action-inputs>
        <region type="string"/>
      </action-inputs>
      <action-outputs>
        <rule-result type="string"/>
      </action-outputs>
      <component-name>org.pentaho.component.JavascriptRule</component-name>
      <component-definition>
        <script> <![CDATA[
          function getManagersEmail( theRegion ) {
            if ( "Central".equals( theRegion ) ) {
              return( "joe.pentaho@pentaho.org" );
            }
            return( "jdixon@pentaho.org" );
          }
          getManagersEmail( region );
        ]]> </script>
      </component-definition>
    </action-definition>
  </actions>
</action-sequence>
In this example, the Action Sequence has 4 inputs: region, from, subject and message-plain. For region, the type is defined as a string; it has a default value of “Central” and may come from one of two sources; request and session. When the RuntimeContext resolves the region input at runtime, it will first look in the request (most likely an http request.) If it doesn’t find it in the request, it will look in the session (most likely the http session.) If it is not available in the session, the default value will be used. The order that the sources are specified in the XML document is the order that they will be searched. The default is always used as a last resort.

The other inputs only specify a default value. This is analogous to hard coding the parameters to a constant value. Since the output of this Action Sequence is an email, no out parameters will be set.

There are 2 action-definition nodes for this sequence. The first defines a JavaScript rule and requires a region parameter; it will create a new parameter called “rule-result”. This new parameter will be made available to other action-definition nodes in the sequence.

Without getting too deep into the workings of the JavaScript rule, the script defined in the component-definition will be executed and will set the value of “rule-result” to the appropriate email address based on the value of region.

When the first action-definition completes, the second will execute. It defines an interaction with the Email component. The email component requires 4 action-inputs: to, from, subject and message-plain. You may have noticed that the action-inputs: from, subject and message-plain are specified in the inputs section of the Action Sequence header. The RuntimeContext will take the values from there and hand them to the Email Component just as it handed region to the JavaScript Component. The source of the “to” action-input isn’t directly defined. It is indirectly defined with the “mapping” attribute. This attribute is telling the RuntimeContext to use the value from “rule-result” that was generated by the JavaScript rule and use it as the components “to” input.

Are There Tools To Help?

If it seems complicated, there is an easier way. The Pentaho Design Studio has been released, and it helps generate and validate the solutions documents as they are built. Download the latest Beta version and user guide at http://sourceforge.net/project/showfiles.php?group_id=140317.

To recap the relationships; the inputs, outputs and resources in the Action Sequence header define a contract in the form of an XML schema between the Action Sequence and the outside world. The action-inputs define a contract between the component and the Action Sequence. The mapping attribute of the action-inputs allow outputs from one component that have different names to be used as inputs to another component.

Specifying the input/output relationships and their data types allows the system to validate an Action Sequence or set of Action Sequences without actually executing the components. A complete solution can
be validated and “locked down” to prevent modification of the Action Sequence documents and eliminate errors due to “broken links” between these documents.

**Executing a Solution**

Action Sequence documents define the tasks that need to be executed, the order in which they will be performed, the input data required and the output data that will be available when they complete. They are stored in the solution repository and are located by specifying solution id, path and action sequence name.

When an Action Sequence is executed, the Solution Engine creates a Runtime Context which provides the execution environment. The Runtime Context contains the input parameters, coordinates access to resources, stores new data and content generated and maintains the state of the Action Sequence execution. As actions are completed, their outputs and generated content are stored in the Runtime Context. At completion of the Action Sequence execution, the data portion of the Runtime Context is persisted in the Runtime Repository.

When a new Runtime Context is created, it is given a unique instance id. The instance id is the key that identifies this Runtime Context when it is persisted. The instance id is also used for auditing and can be tracked back to parent instance that created it. If you are wondering what happens to all those completed instances that have been saved to the runtime repository, don’t worry, I’m sure those Pentaho people have some kind of clean up scheme.

Action Sequences don’t have to be given a new Runtime Context each time they are executed. They can be given a previous instance id which makes all the data and content from previous Action Sequence Document executions available. This is where the contract between Action Sequence documents becomes important. As long as an output from a previous Action Sequence document has been specified, it will be made available as an input to a later Action Sequence. This ability to string Action Sequence documents together via the same Runtime Context becomes quite powerful when it is used within a workflow engine.

There are several ways to execute a solution; via URL, Java Code or a Web Service call.

**URL**

The samples that come with the preconfigured install are launched via URL using the ViewAction (org.pentaho.ui.servlet.ViewAction) servlet. The following URL will launch the HelloWorld Action Sequence:

```
http://localhost:8080/pentaho/ViewAction?&solution=samples&path=getting-started&action=HelloWorld.xaction
```

The result returned depends on the Action Sequence Document. You may get a report to view, a text message or just “Action Successful.” The following parameters can be entered on the URL:

- `solution, path, action` - The location of the Action Sequence document to load.
- `instance_id` - The instance Id of a previous Runtime Context
- `debug` - set to “true” in order to have debug information written to the execution log.

**Web Service Call**

In the “Settings and Services” group of the samples that come with the preconfigured install is a Web Service Example. It is still a URL call, this time to the servlet ServiceAction (org.pentaho.ui.servlet.HttpWebService). The following URL will launch the HelloWorld Action Sequence:

```
```

In this case, the result returned is an XML SOAP Response. The following parameters can be entered on the URL:

- `solution, path, action` - The location of the Action Sequence document to load.
• **instance_id** - The instance Id of a previous Runtime Context
• **debug** - set to “true” in order to have debug information written to the execution log.

## Java Call

An Action Sequence can be executed directly from a Java application. For an example of how to do this, open the Java file “org.pentaho.test.RuntimeTest.java” and look at the JUnit test for HelloWorld.

## Parameters

There are three types of parameters that Action Sequence documents understand; inputs, outputs and resources. Inputs and outputs are variables of a specific data type like string or property-map (see Appendix B for valid data types.) Resources are similar to inputs except they specify a mime type and path. A default value cannot be specified for resources. Typically resources represent large amounts of data like report definitions or images (see Appendix B for valid resource types.)

Parameters can come from four sources; runtime, request, session and default. Runtime parameters are parameters that are stored in the Runtime Context. Remember, the Runtime Context stores the inputs and outputs from previous instances and makes them available to future executions of the same runtime instance id. Request parameters are the name-value pairs specified on a URL. Session parameters are variables that are stored in the session (usually an HTTP session.) Default values are specified in the Action Sequence document and are used as a last resort.

Here is an example if the inputs section of an Action Sequence document:

```xml
<inputs>
  <region type="string">
    <default-value>Central</default-value>
    <sources>
      <request>REGION</request>
      <runtime>aRegion</runtime>
    </sources>
  </region>
</inputs>
```

This example indicates that the Action Sequence document requires a parameter named “region” (case sensitive.) When executed, the Runtime Context will first look to see if there was a parameter named “REGION” in the request. If the Action Sequence was launched from a URL, and there was a parameter “REGION=xxx” specified, than this value (xxx) will be substituted for the “region” input. If it doesn’t find the parameter in the request, it will look in its own runtime data for a parameter named “aRegion.” If it doesn’t find it in the Runtime Context Data, the value “Central” will be used. The Runtime Context always looks in the sources in the order in which they are specified and takes the default last. If no default was specified, then the Action Sequence would throw an error and return.

There are two implicit parameters **instance-id** and **solution-id** that are always available and do not need to be specified as inputs or outputs. They are the... well I’m sure you can guess what they are.

## Components

The Action Sequence document is the definition, the Runtime Context provides an execution environment and the Components are the business logic. A Component performs a single function, a group of related functions or is a bridge between the Pentaho BI Platform and an external application. The BIRT reporting component is an example of a component that interfaces the platform to the BIRT reporting engine.

There are two major functions that a Component gets called to do - validate and execute. Validate is called to verify that the inputs are valid and all resources required to execute are available. Execute actually performs the action.
The action-definition node in the Action Sequence document defines how the component should function. It is the place to define and map inputs, outputs and any other metadata or settings the component will require when it is executed. The Java class that implements the component is specified by its fully qualified class name in the component-name node. This class will be dynamically loaded at runtime.

The action-inputs define the parameters that will be available to the component when it executes. Some components have required inputs that must be available or runtime error will occur. Some inputs may be specified but are optional. The action-outputs define what parameters will be saved in the Runtime Context and be made available to other Components when that component finishes executing.

The component-definition node provides a place for any Component specific options or parameters. See Appendix C for the Component specific input, output and definition requirements.

Integrating your own reports

Overview

Introduction

So now you've built your own reports using Eclipse BIRT or JasperReports that use your own data, and you're asking yourself - how do I integrate these reports into the Pentaho BI Platform? That's what this how-to guide is for. It assumes familiarity with your reporting tool of choice (be it JasperReports or Eclipse BIRT), and it assumes that you have the following "in hand" so to speak:

- A working report XML document (.rptdesign for Eclipse BIRT or a .jrxml for JasperReports). By working, I mean that you were able to test the report outside of the Pentaho BI Platform, and it is working properly.
- The JDBC driver .jar file for the database you are using.

General Assumptions and Conventions

This how-to document assumes the following conditions are present:

- The Pentaho pre-configured install has been installed, configured, and is operational.
- All slashes in this document will be presented using forward slashes (/). On Windows platforms, forward slashes should be changed to backward-slashes (\) unless specifically stated.
- The Pentaho pre-configured install directory will be referenced in this document as {PCI}. Where you see specific paths, they will be referenced starting from the pre-configured install root like this: {PCI}/jboss/server/default/deploy. Thus, if you have installed the pre-configured install on your Windows C: drive in a folder called pentaho-pci, then '{PCI}/' means 'C: \pentaho-pci'.
- The reader has read and understands the above section on The Action Sequence, Appendix A-Action Sequence XML in Detail, and Appendix B with respect to action sequences, their format, and content.

Integrating BIRT Reports

Eclipse BIRT Report Definitions

Eclipse BIRT report files (report.rptdesign) are simply XML documents with a funny extension. The extension allows the Eclipse IDE to recognize the file as a BIRT report. The Pentaho BIRTReportComponent (org.pentaho.birt.BIRTReportComponent) is capable of executing these reports, and producing output as HTML, PDF, FO, and FOP.
Eclipse BIRT parameters are defined in the report definition as Scalar Parameters. In BIRT a parameter must be added to a Data Set. This is done when creating a Data Set or while editing an existing Data Set. There is a “Parameters” section of the Data Set editor dialog which is where you will define your parameters. When defining the parameter the “Default Value” must follow the convention `params["SOME_PARAMETER"]`. The “SOME_PARAMETER” must match the name of a report parameter as defined in BIRT’s Data Explorer “Report Parameters” section.

For the example above, with a report parameter named “REGION”, the report parameter has been defined as shown below.
Creating the Action Sequence

After verifying that the report works in BIRT correctly we can safely drop it into a Pentaho solution. Copy the BIRT report into {PCI}/pentaho-solutions/samples/reporting. What we need to do now is create a Pentaho action sequence XML document and save it in {PCI}/pentaho-solutions/samples/reporting. This document is made up of several sections: documentation, inputs, outputs, resources and action definitions. Note that the name of the action sequence document must match the <name> XML tag in it.

- **Documentation:** This section allows you to include author information, a description, an icon to represent the action sequence, a help URL and a result-type. For this example, we specify “report” as the result-type.

```
<documentation>
  <author>Michael D'Amour</author>
  <description>BIRT Report with Parameters</description>
  <icon>reporting.png</icon>
  <help>https://example.com</help>
  <result-type>report</result-type>
</documentation>
```
• **Inputs:** There are two inputs, output-type and REGION. Notice the REGION input is the same name as the report parameter from BIRT. This is important. The output-type in the example below is "html", other acceptable output-types are pdf, fo and fop.

```xml
<inputs>
  <output-type type="string">
    <default-value>html</default-value>
    <sources>
      <request>type</request>
    </sources>
  </output-type>
  <REGION type="string">
    <default-value></default-value>
    <sources>
      <request>REGION</request>
    </sources>
  </REGION>
</inputs>
```

• **Resources:** In this section we define a “report-definition” which points to the BIRT .rptdesign XML file.

```xml
<resources>
  <report-definition>
    <solution-file>
      <location>BIRT-report.rptdesign</location>
      <mime-type>text/xml</mime-type>
    </solution-file>
  </report-definition>
</resources>
```

• **Actions:** For this example we have only one action-definition. The component-name specifies the Java class that handles the action. For BIRT reports use name "org.pentaho.birt.BIRTReportComponent". There are two inputs to this action, the output-type and the REGION. These inputs are defined in the inputs section of the action-sequence document.

```xml
<actions>
  <action-definition>
    <action-inputs>
      <output-type type="string"/>
      <REGION type="string"/>
    </action-inputs>
    <component-name>org.pentaho.birt.BIRTReportComponent</component-name>
    <action-type>report</action-type>
    <component-definition/>
  </action-definition>
</actions>
```

The entire action sequence document used for the example is shown below.

```xml
<action-sequence>
  <name>report.xaction</name>
  <version>1</version>
  <title>Parameterized BIRT Report</title>
  <documentation>
    <author>Michael D'Amour</author>
    <description>BIRT Report with Parameters</description>
    <icon>reporting.png</icon>
  </documentation>
</action-sequence>
```
JDBC Driver Setup

The BIRT report contains JDBC connection information that will be used by BIRT to generate the reports. Since we are generating the reports inside the Pentaho framework we must put the JDBC driver for the database we are using in {PCI}/jboss/server/default/lib.

Verifying BIRT Integration into Pentaho Platform

At this point the report should be plugged in and ready for use. Point your web browser to your PCI (typically http://localhost:8080). Navigate to the report that you created under the Reporting Examples group.
At this point you are prompted for the parameter.

Success! Here we see the generated report in HTML.

<table>
<thead>
<tr>
<th>DEPARTMENT</th>
<th>TITLE</th>
<th>ACTUAL</th>
<th>BUDGET</th>
<th>VARIANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sales</td>
<td>District Manager</td>
<td>$682,625.00</td>
<td>$617,250.00</td>
<td>$65,375.00</td>
</tr>
<tr>
<td>Sales</td>
<td>Senior Sales Rep</td>
<td>$497,223.00</td>
<td>$484,820.00</td>
<td>$12,403.00</td>
</tr>
<tr>
<td>Sales</td>
<td>Sales Rep</td>
<td>$675,975.00</td>
<td>$612,500.00</td>
<td>$63,475.00</td>
</tr>
<tr>
<td>Sales</td>
<td>Account Executive</td>
<td>$409,975.00</td>
<td>$422,500.00</td>
<td>$12,525.00</td>
</tr>
</tbody>
</table>

Region: Central
Integrating JasperReports

JasperReports Report Definitions

JasperReports report files (report.jrxml) are also just XML documents with a funny extension. In JasperReports though, the .jrxml file is considered the “source code” for your report. Before you can execute the report, you must compile the file into a file that has a .jasper extension. The Pentaho JasperReportsComponent (org.pentaho.jasper.JasperReportsComponent) is designed to do that “compilation” for you, so all you’ll need to get started is the .jrxml file. The Pentaho JasperReportsComponent is able to execute these reports, and produce output as either HTML or PDF.

JasperReports parameters are defined in the report definition. If you’re using iReport to create your JasperReports, you create parameters in your Object Library then reference the parameter in your report query.

The report above is using a parameter named “STATE”. In order for the user to be prompted for the value of the STATE parameter from within the Pentaho solution you must select “Is for prompting” when creating the parameter, as shown below.
Creating the Action Sequence

We can now drop the JasperReport into a Pentaho solution. Copy the jrxml file into {PCI}/pentaho-solutions/samples/reporting. What we need to do now is create a Pentaho action sequence XML document and save it in {PCI}/pentaho-solutions/samples/reporting. This document is made up of several sections: documentation, inputs, outputs, resources and action definitions. Note that the name of the action sequence document must match the <name> XML tag in it.

- **Documentation**: This section allows you to include author information, a description, an icon to represent the action sequence, a help URL and a result-type. For this example, we specify “report” as the result-type.

```xml
<documentation>
  <author>Angelo Rodriguez</author>
  <description>Example of how to add a new Jasper Report to a solution</description>
  <icon>reporting.png</icon>
  <help></help>
  <result-type>report</result-type>
</documentation>
```

- **Inputs**: There are two inputs, output-type and STATE. Notice the STATE input is the same name as the report parameter being used by the JasperReport. This is important. The output-type in the example below is “html”. Optionally “pdf” could have been specified.

```xml
<inputs>
  <output-type type="string">
    <default-value>html</default-value>
    <sources>
      <request type="type"/>
    </sources>
  </output-type>
  <STATE type="string">
    <default-value></default-value>
    <sources>
      <request type="type"/>
    </sources>
  </STATE>
</inputs>
```
Resources; In this section we define a “report-definition” which points to the JasperReport .jrxml XML file.

```xml
<resources>
  <report-definition>
    <solution-file>
      <location>report.jrxml</location>
      <mime-type>text/xml</mime-type>
    </solution-file>
  </report-definition>
</resources>
```

Actions; For this example we have only one action-definition. The component-name specifies the Java class that handles the action. For JasperReports use “org.pentaho.jasper.JasperReportsComponent”. There are two inputs to this action, the output-type and the STATE. These inputs are defined in the inputs section of the action-sequence document.

```xml
<actions>
  <action-definition>
    <action-inputs>
      <output-type type="string"/>
      <STATE type="string"/>
    </action-inputs>
    <action-outputs>
      <component-name>org.pentaho.jasper.JasperReportsComponent</component-name>
      <action-type>report</action-type>
    </component-definition>
  </action-definition>
</actions>
```
The entire action sequence document used for the example is shown below.

```xml
<action-sequence>
  <name>report.xaction</name>
  <version>1</version>
  <title>Sample MySql Report</title>
  <documentation>
    <author>Angelo Rodriguez</author>
    <description>Example of how to add a new Jasper Report to a solution</description>
    <icon>reporting.png</icon>
    <help></help>
    <result-type>report</result-type>
  </documentation>
  <inputs>
    <output-type type="string">
      <default-value>html</default-value>
      <sources>
        <request>type</request>
      </sources>
    </output-type>
    <STATE type="string">
      <default-value></default-value>
      <sources>
        <request>STATE</request>
      </sources>
    </STATE>
  </inputs>
  <outputs>
    <outputs>
      <resources>
        <report-definition>
          <solution-file>
            <location>report.jrxml</location>
            <mime-type>text/xml</mime-type>
          </solution-file>
        </report-definition>
      </resources>
      <actions>
        <action-definition>
          <action-inputs>
            <output-type type="string"/>
            <STATE type="string"/>
          </action-inputs>
          <action-outputs>
            <component-name>org.pentaho.jasper.JasperReportsComponent</component-name>
            <action-type>report</action-type>
            <component-definition>
              <jndiUrl>MySqlDS</jndiUrl>
            </component-definition>
          </action-outputs>
        </action-definition>
      </actions>
    </outputs>
  </outputs>
</action-sequence>
```
JDBC Driver Setup

Since we are generating the reports inside the Pentaho framework we must put the JDBC driver for the database we are using in {PCI}/jboss/server/default/lib.

Unlike BIRT report definitions, .jrxml files do not contain the database connection information for the report. This information needs to be specified in the action sequence. As shown above, the database location can be defined within the action definitions as follows:

```xml
<component-definition>
    <driver>com.mysql.jdbc.Driver</driver>
    <connection>jdbc:mysql://localhost:3306/sampledata</connection>
    <user-id>jim</user-id>
    <password>password</password>
</component-definition>
```

Alternatively the database can be identified using a JNDI name as follows:

```xml
<component-definition>
    <jndi>MySqlDS</jndi>
</component-definition>
```

If you're going to use JNDI to identify the report database you'll need to configure JBoss to map the JNDI name to your database as follows:

- Create an xxxxx-ds.xml file for your database type in {PCI}/jboss/server/default/deploy. For this example we'll create a mysql-ds.xml file with the following content.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
    <local-tx-datasource>
        <jndi-name>MySqlDS</jndi-name>
        <connection-url>jdbc:mysql://localhost:3306/sampledata</connection-url>
        <driver-class>com.mysql.jdbc.Driver</driver-class>
        <user-name>jim</user-name>
        <password>password</password>
        <exception-sorter-class-name>
            org.jboss.resource.adapter.jdbc.vendor.MySQLExceptionSorter
        </exception-sorter-class-name>
        <metadata>
            <type-mapping>mySQL</type-mapping>
        </metadata>
    </local-tx-datasource>
</datasources>
```

- Add a resource reference to {PCI}/jboss/server/default/deploy/pentaho.war/WEB-INF/web.xml.

```xml
<resource-ref>
    <description>MySql Connection</description>
    <res-ref-name>jdbc/MySqlDS</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

- Add a resource reference to {PCI}/jboss/server/default/deploy/pentaho.war/WEB-INF/jboss-web.xml.

```xml
<resource-ref>
    <res-ref-name>jdbc/MySqlDS</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <jndi-name>java:/MySqlDS</jndi-name>
</resource-ref>
```
Verifying JasperReports Integration into Pentaho Platform

Restart your Pentaho server by running `{PCI}/stop_pentaho then {PCI}/start_pentaho`. At this point the report should be plugged in and ready for use. Point your web browser to your PCI (typically http://localhost:8080). Navigate to the report that you created under the Reporting Examples group.

At this point you are prompted for the parameter.

At last we’ve reached our goal! Here we see the generated report in HTML. Isn’t it a beauty?
Charting

The Pentaho BI platform currently employs JFreeChart as its charting engine. The implementation of the engine currently includes UI components for the following charts: Dial, Pie, Pie Grid, Bar Chart, Line, and Area. Each chart component can create at least 1 type of chart. Several of the chart components can create many chart types. Certain charts (where it makes sense) can be rendered as stacked and three dimensional. The platform charts render themselves as XML and then the XML is transformed to HTML via the use of an .xslt transformation.

ChartComponent

The ChartComponent is a UI component that can create a variety of charts including Bar, Line, Pie, Pie Grid, and Area. Where applicable there are several options that can be applied. The following action sequence uses the output from an SQLLookupRule as the input for a ChartComponent.

```xml
<name>Chart.xaction</name>
<title>Default Title</title>
<version>1</version>
<logging-level>DEBUG</logging-level>

<documentation>
  <author>Brian C Hagan</author>
  <description>Default Description</description>
  <icon>JFree-quadrant-budget-hsql.png</icon>
  <help>Help</help>
  <result-type>rule</result-type>
</documentation>

<inputs>
  <chart-type type="string">
    <default-value>.png</default-value>
    <sources>
      <request>type</request>
    </sources>
  </chart-type>
</inputs>

<actions>
  <action-definition>
    <action-outputs>
      <result-set type="list" />
    </action-outputs>
    <component-name>org.pentaho.component.SQLLookupRule</component-name>
    <action-type>rule</action-type>
    <component-definition>
      <source>sql</source>
      <live>true</live>
      <jndi>SampleData</jndi>
      <query>
        <![CDATA[select QUADRANT_ACTUALS.REGION, QUADRANT_ACTUALS.DEPARTMENT,
                     QUADRANT_ACTUALS.POSITIONTITLE, QUADRANT_ACTUALS.ACTUAL, QUADRANT_ACTUALS.BUDGET
                     from QUADRANT_ACTUALS order by QUADRANT_ACTUALS.REGION,
                     QUADRANT_ACTUALS.DEPARTMENT]]}>
      </query>
    </component-definition>
  </action-definition>
</actions>
```
<by-row/> Required node that describes how the chart data is aggregated. Values are TRUE or FALSE.

<chart-attributes/> Required by the ChartComponent, this node contains all chart attributes nodes. This node is not used by the CategoryDatasetComponent (described below).

<chart/> The chart root node. This item is a mandatory singleton node. This node is not used by the ChartComponent.
<chart-type/> Optional singleton node that contains the chart type in the node text. When used with the CategoryDatasetComponent, if the chart-type is not set in the xml definition then it must be set on the CategoryDataset Component directly in code (servlet or JSP).

<title/> Optional singleton node that contains the requested title in the node text. Valid chart type strings are “PieChart”, “PieGrid”, “BarChart”, “LineChart” and “AreaChart”

<title-position/> Optional node that describes the position of the chart title. Valid positions are TOP, BOTTOM, LEFT, and RIGHT.

<title-font/> Optional singleton node that describes the chart font. Font attributes are included as child nodes. Child nodes include, <font-family>, <size>, <is-bold>, and <is-italic>.

<subtitle/> Optional singleton node that contains the requested subtitle in the node text.

<range-title/> Optional node that describes the chart range (usually the y-axis).

<chart-background/> Optional singleton node that describes the background type. Valid types include color and image. If type="color", designate the color with 6 byte hexadecimal notation. When type="image", the value is a filepath (relative to the solution directory) of an image file to use as the chart background image. Note: This image replaces the background of the chart itself and NOT the plot area. So if you set the image here you will probably see your image under the axis labels and scales and not in the plot area.

<plot-background/> Optional singleton node that describes the plot type. Valid types include color and image. If type="color", designate the color with 6 byte hexadecimal notation. When type="image", the value is a filepath (relative to the solution directory) of an image file to use as the plot background image.. Note: This image replaces the background of the plot area only.

<orientation/> Optional singleton whose text value can be either “Horizontal”, “Vertical”. Defaults to “Vertical”.

<height/> Optional singleton whose text value is an integer that represents the height of the chart.

<width/> Optional singleton whose text value is an integer that represents the width of the chart.

<is-3D/> Optional singleton whose text value can be either “true” or “false”. Defaults to “false”. If true the charting engine does it best to render a 3-D view of the chart.

<is-stacked/> Optional singleton whose text value can be either “true” or “false”. Defaults to “false”. If true the charting engine will create a stacked version of this chart type (if possible).

<urlTemplate/> Optional singleton whose text value is used as a template to create a drill link map for the image. This is not used with the ChartComponent.

<paramName/> Optional singleton whose test values is the parameter name of the innermost query variable. If this name occurs in the urlTemplate, it will be replaced with the correct item. This is not used with the ChartComponent.

<paramName2/> Optional singleton whose test values is the parameter name of the outermost query variable. If this name occurs in the urlTemplate, it will be replaced with the correct item. This is not used with the ChartComponent.

<color-palette/> Singleton that contains a list of <color/> nodes that make up the series palette.

<color/> Optional multiple nodes that contain a 6 byte hexadecimal notation to be used as an entry in the series palette.
The CategoryDatasetComponent is a UI component that can create a variety of charts including Bar, Line, Pie, Pie Grid, and Area. Where applicable there are several options that can be applied. The creation of the chart is commonly performed by a JSP or indirectly by creating the appropriate portlet object. See the ChartComponent for explanations of each node.

```xml
<chart>
  <chart-type>BarChart</chart-type>
  <title>Sample Chart</title>
  <subtitle>a simple sample</subtitle>
  <chart-background type="color">#FFFFFF</chart-background>
  <chart-background-image>test\charts\ChartBackground.jpg</chart-background-image>
  <plot-background-color>#FFFFFF</plot-background-color>
  <plot-background-image>test\charts\ChartBackground.jpg</plot-background-image>
  <orientation>Horizontal</orientation>
  <height>550</height>
  <width>650</width>
  <is-3D>true</is-3D>
  <is-stacked>true</is-stacked>
  <urlTemplate><![CDATA[/pentaho/Pivot?solution=samples&path=analysis&action=query1.xaction&department=.[{DEPARTMENT}]&measures=.[{MEASURES}]]]]></urlTemplate>
  <paramName>MEASURES</paramName>
  <paramName2>DEPARTMENT</paramName2>
  <color-palette>
    <color>#336699</color>
    <color>#99CCFF</color>
    <color>#999933</color>
    <color>#666699</color>
    <color>#CC9933</color>
    <color>#006666</color>
    <color>#3399FF</color>
    <color>#993300</color>
    <color>#CCCC99</color>
    <color>#666666</color>
    <color>#FFCC66</color>
    <color>#6699CC</color>
    <color>#663366</color>
    <color>#9999CC</color>
    <color>#CCCCCC</color>
    <color>#669999</color>
    <color>#CCCC66</color>
    <color>#CC6600</color>
    <color>#9999FF</color>
    <color>#0066CC</color>
    <color>#99CCCC</color>
    <color>#999999</color>
    <color>#FF9900</color>
    <color>#999966</color>
    <color>#66CCCC</color>
    <color>#339966</color>
    <color>#CCCC33</color>
  </color-palette>
</chart>
```
Scheduler

The Pentaho BI platform currently employs Quartz as its scheduler. The implementation is a singleton that is JDBC persisted. It is fault tolerant and fault recoverable. Scheduled misfires are handled according to a set of predefined rules.

Access to the scheduler is through the org.pentaho.component.JobSchedulerComponent by implementing a solution document. Samples can be found in test/scheduler/. There are currently four different actions available to the job scheduler. “startJob”, “suspendJob”, “resumeJob”, and “deleteJob”.

Actions

- **startJob** - Creates a job and a trigger and then registers the job and trigger for execution with the scheduler. In the case of a “Simple Trigger”, actual job execution occurs when the trigger condition is met and occurs at the defined repeat interval until the number of defined repeat cycles has occurred. In the case of “Cron Trigger” the firing of the job occurs according to the rules set forth in the cron expression string (see below). The job itself is a solution document that is to be performed. This allows the scheduling of any other existing solution such as printing and email. In the event of a fault such as power failure, system crash, etc., after the scheduler restarts it will apply the misfire rules to any trigger that has misfired. The values of the “jobName”, “triggerType” (and the trigger types associated inputs), “solution”, “path”, and “action” need to be defined in the solution document.

```xml
<action-sequence>
  <name>SchedulerTest_new_job.xaction</name>
  <version>1</version>
  <title>Schedules a task</title>
  <logging-level>debug</logging-level>
  <documentation>
    <author>William E. Seyler</author>
    <description>..</description>
    <help>This is just a test...</help>
  </documentation>
  <inputs>
    <jobAction type="string">
      <default-value>startJob</default-value>
    </jobAction>
    <solution type="string">
      <default-value>test</default-value>
    </solution>
    <path type="string">
      <default-value>email</default-value>
    </path>
    <action type="string">
      <default-value>text_only_email.xaction</default-value>
    </action>
    <jobName type="string">
      <default-value>MyJob</default-value>
    </jobName>
    <triggerType type="string">
      <default-value>simple</default-value>
    </triggerType>
    <triggerName type="string">
      <default-value>MyTrigger</default-value>
    </triggerName>
    <repeatInterval type="string">
      <default-value>10</default-value>
    </repeatInterval>
  </inputs>
</action-sequence>
```
<repeatCount type="string">
  <default-value>1</default-value>
</repeatCount>

<misfirePolicy type="string">
  <default-value>INSTRUCTION_DELETE_TRIGGER</default-value>
</misfirePolicy>

<jobAction type="string"/>
<solution type="string"/>
<path type="string"/>
<action type="string"/>
<jobName type="string"/>
<triggerType type="string"/>
<triggerName type="string"/>
<repeatInterval type="string"/>
<repeatCount type="string"/>
<misfirePolicy type="string"/>

<component-name>org.pentaho.component.JobSchedulerComponent</component-name>
<action-type>schedule</action-type>

<actions>
  <action-definition>
    <action-inputs>
      <jobAction type="string"/>
      <solution type="string"/>
      <path type="string"/>
      <action type="string"/>
      <jobName type="string"/>
      <triggerType type="string"/>
      <triggerName type="string"/>
      <repeatInterval type="string"/>
      <repeatCount type="string"/>
      <misfirePolicy type="string"/>
    </action-inputs>
    <component-name>org.pentaho.component.JobSchedulerComponent</component-name>
    <action-type>schedule</action-type>
    <component-definition/>
  </action-definition>
</actions>

<resources/>

<actions>
  <action-definition>
    <action-inputs>
      <jobAction type="string"/>
      <solution type="string"/>
      <path type="string"/>
      <action type="string"/>
      <jobName type="string"/>
      <triggerType type="string"/>
      <triggerName type="string"/>
      <repeatInterval type="string"/>
      <repeatCount type="string"/>
      <misfirePolicy type="string"/>
    </action-inputs>
    <component-name>org.pentaho.component.JobSchedulerComponent</component-name>
    <action-type>schedule</action-type>
    <component-definition/>
  </action-definition>
</actions>

<resources/>

<actions>
...

Figure X.X -- Sample StartJob Action Sequence

- **suspendJob** - Pauses a specified running job. Once the job is paused the only way to start it again is with a resume job. Once the job is resumed it will apply the misfire rules if required. The only input for this action is the “jobName” that is to be suspended.

<action-sequence>
  <name>SchedulerTest_pause_job.xaction</name>
  <version>1</version>
  <title>Pauses a scheduled job</title>
  <logging-level>debug</logging-level>
  <documentation>
    <author>William E. Seyler</author>
    <description>..</description>
    <help>This is just a test...</help>
  </documentation>
  <inputs>
    <jobAction type="string">
      <default-value>suspendJob</default-value>
    </jobAction>
    <jobName type="string">
      <default-value>MyJob</default-value>
    </jobName>
  </inputs>
</actions>

<outputs/>
<resources/>
**resumeJob** - Resumes a specified suspended job. The only input for this action is the "jobName" to resume.

```
<action-definition>
  <action-inputs>
    <jobAction type="string"/>
    <jobName type="string"/>
  </action-inputs>
  <component-name>org.pentaho.component.JobSchedulerComponent</component-name>
  <action-type>schedule</action-type>
  <component-definition/>
</action-definition>
```

Figure X.X – Sample SuspendJob Action Sequence

- **deleteJob** - Deletes a specified job. The job is deleted immediately. However if a job is currently executing in a scheduler thread then it will continue to execute. However, no new instances of the job will be scheduled. The only input for this action is a "jobName".

```
<action-definition>
  <action-inputs>
    <jobAction type="string"/>
    <jobName type="string"/>
  </action-inputs>
  <component-name>org.pentaho.component.JobSchedulerComponent</component-name>
  <action-type>schedule</action-type>
  <component-definition/>
</action-definition>
```

Figure X.X – Sample deleteJob Action Sequence
<name>SchedulerTest_delete_job.xaction</name>
  <version>1</version>
  <title>Deletes a Job</title>
  <logging-level>debug</logging-level>
  <documentation>
    <author>William E. Seyler</author>
    <description>..</description>
    <help>This is just a test...</help>
  </documentation>
  <inputs>
    <jobAction type="string">
      <default-value>deleteJob</default-value>
    </jobAction>
    <jobName type="string">
      <default-value>MyJob</default-value>
    </jobName>
  </inputs>
  <outputs/>
  <resources/>
  <actions>
    <action-definition>
      <action-inputs>
        <jobAction type="string"/>
        <jobName type="string"/>
      </action-inputs>
      <component-name>org.pentaho.component.JobSchedulerComponent</component-name>
      <action-type>schedule</action-type>
    </action-definition>
  </actions>
</action-sequence>

Figure X.X - Sample deleteJob Action Sequence
**Triggers**

The Pentaho BI Platform currently supports two different types of triggers.

- **Simple Trigger** – selected in the “triggerType” node as “simple”. The simple trigger allows a task to be scheduled at a specified regular interval for a specified number of repetitions. The inputs to the simple trigger are the integer values “repeatInterval” (in seconds) and “repeatCount”. The trigger will begin firing immediately and continue at the “repeatInterval” for “repeatCount” number of cycles. See Figure X.X -- Sample StartJob Action Sequence for an example.

- **Cron Trigger** – selected in the “triggerType” node as “cron”. This trigger uses unix style cron task definitions. The cron trigger takes a “cronString” that represents the trigger definition much like an entry into crontab. The following describes the format of the cron expression string.

```xml
<action-sequence>
  <name>SchedulerTest_new_cron_job.xaction</name>
  <version>1</version>
  <title>Schedules a task</title>
  <logging-level>debug</logging-level>
  <documentation>
    <author>William E. Seyler</author>
    <description>..</description>
    <help>This is just a test...</help>
  </documentation>
  <inputs>
    <jobAction type="string">
      <default-value>startJob</default-value>
    </jobAction>
    <solution type="string">
      <default-value>test</default-value>
    </solution>
    <path type="string">
      <default-value>email</default-value>
    </path>
    <action type="string">
      <default-value>text_only_email.xaction</default-value>
    </action>
    <jobName type="string">
      <default-value>MyJob</default-value>
    </jobName>
    <triggerType type="string">
      <default-value>cron</default-value>
    </triggerType>
    <triggerName type="string">
      <default-value>MyTrigger</default-value>
    </triggerName>
    <cronString type="string">
      <default-value>0 40 16 * * ?</default-value>
    </cronString>
    <misfirePolicy type="string">
      <default-value>MISFIRE_INSTRUCTION_SMART_POLICY</default-value>
    </misfirePolicy>
  </inputs>
  <outputs/>
  <resources/>
</action-sequence>
```
// NOTE the following is from the javadoc for org.quartz.cronTrigger

A "cronString" is a string comprised of 6 or 7 fields separated by white space. The 6 mandatory and 1 optional fields are as follows:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Allowed Values</th>
<th>Allowed Special Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seconds</td>
<td>0-59</td>
<td>, - * /</td>
</tr>
<tr>
<td>Minutes</td>
<td>0-59</td>
<td>, - * /</td>
</tr>
<tr>
<td>Hours</td>
<td>0-23</td>
<td>, - * /</td>
</tr>
<tr>
<td>Day-of-month</td>
<td>1-31</td>
<td>, - * ? / L W C</td>
</tr>
<tr>
<td>Month</td>
<td>1-12 or JAN-DEC</td>
<td>, - * /</td>
</tr>
<tr>
<td>Day-of-Week</td>
<td>1-7 or SUN-SAT</td>
<td>, - * ? / L C #</td>
</tr>
<tr>
<td>Year (Optional)</td>
<td>empty, 1970-2099</td>
<td>, - * /</td>
</tr>
</tbody>
</table>

The '*' character is used to specify all values. For example, "*" in the minute field means "every minute".

The '?' character is allowed for the day-of-month and day-of-week fields. It is used to specify 'no specific value'. This is useful when you need to specify something in one of the two fields, but not the other. See the examples below for clarification.

The '-' character is used to specify ranges. For example "10-12" in the hour field means "the hours 10, 11 and 12".

The ',' character is used to specify additional values. For example "MON,WED,FRI" in the day-of-week field means "the days Monday, Wednesday, and Friday".

The '/' character is used to specify increments. For example "0/15" in the seconds field means "the seconds 0, 15, 30, and 45". And "5/15" in the seconds field means "the seconds 5, 20, 35, and 50". You can also specify '/' after the '*' character - in this case '*' is equivalent to having '0' before the '/'.

The 'L' character is used to specify the day-of-month and day-of-week fields. This character is short-hand for "last", but it has different meaning in each of the two fields. For example, the value "L" in the day-of-month field means "the last day of the month" - day 31 for January, day 28 for February on non-leap years. If used in the day-of-week field by itself, it simply means "7" or "SAT". But if used in the day-of-week field after another value, it means "the last xxx day of the month" - for example "6L" means "the last friday of the month". When using the 'L' option, it is important not to specify lists, or ranges of values, as you'll get confusing results.
The 'W' character is allowed for the day-of-month field. This character is used to specify the weekday (Monday-Friday) nearest the given day. As an example, if you were to specify "15W" as the value for the day-of-month field, the meaning is: "the nearest weekday to the 15th of the month". So if the 15th is a Saturday, the trigger will fire on Friday the 14th. If the 15th is a Sunday, the trigger will fire on Monday the 16th. If the 15th is a Tuesday, then it will fire on Tuesday the 15th. However if you specify "1W" as the value for day-of-month, and the 1st is a Saturday, the trigger will fire on Monday the 3rd, as it will not 'jump' over the boundary of a month's days. The 'W' character can only be specified when the day-of-month is a single day, not a range or list of days.

The 'L' and 'W' characters can also be combined for the day-of-month expression to yield 'LW', which translates to "last weekday of the month".

The '#' character is allowed for the day-of-week field. This character is used to specify "the nth" XXX day of the month. For example, the value of "6#3" in the day-of-week field means the third Friday of the month (day 6 = Friday and "#3" = the 3rd one in the month). Other examples: "2#1" = the first Monday of the month and "4#5" = the fifth Wednesday of the month. Note that if you specify "#5" and there is not 5 of the given day-of-week in the month, then no firing will occur that month.

The 'C' character is allowed for the day-of-month and day-of-week fields. This character is short-hand for "calendar". This means values are calculated against the associated calendar, if any. If no calendar is associated, then it is equivalent to having an all-inclusive calendar. A value of "5C" in the day-of-month field means "the first day included by the calendar on or after the 5th". A value of "1C" in the day-of-week field means "the first day included by the calendar on or after sunday".

The legal characters and the names of months and days of the week are not case sensitive.

Here are some full examples:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;0 0 0 0 * ?&quot;</td>
<td>Fire at 12pm (noon) every day</td>
</tr>
<tr>
<td>&quot;0 15 10 ? * *&quot;</td>
<td>Fire at 10:15am every day</td>
</tr>
<tr>
<td>&quot;0 15 10 * * ?&quot;</td>
<td>Fire at 10:15am every day</td>
</tr>
<tr>
<td>&quot;0 15 10 * ? *&quot;</td>
<td>Fire at 10:15am every day</td>
</tr>
<tr>
<td>&quot;0 15 10 * ? 2005&quot;</td>
<td>Fire at 10:15am every day during the year 2005</td>
</tr>
<tr>
<td>&quot;0 * 14 * * ?&quot;</td>
<td>Fire every minute starting at 2pm and ending at 2:59pm, every day</td>
</tr>
<tr>
<td>&quot;0 0/5 14 * * ?&quot;</td>
<td>Fire every 5 minutes starting at 2pm and ending at 2:55pm, every day</td>
</tr>
<tr>
<td>&quot;0 0/5 14,18 * * ?&quot;</td>
<td>Fire every 5 minutes starting at 2pm and ending at 2:55pm, AND fire every 5 minutes starting at 6pm and ending at 6:55pm, every day</td>
</tr>
<tr>
<td>&quot;0 0-5 14 * * ?&quot;</td>
<td>Fire every minute starting at 2pm and ending at 2:05pm, every day</td>
</tr>
<tr>
<td>&quot;0 10 44 14 ? 3 WED&quot;</td>
<td>Fire at 2:10pm and at 2:44pm every Wednesday in the month of March</td>
</tr>
<tr>
<td>&quot;0 15 10 ? * MON-FRI&quot;</td>
<td>Fire at 10:15am every Monday, Tuesday, Wednesday, Thursday and Friday</td>
</tr>
<tr>
<td>&quot;0 15 10 15 * ?&quot;</td>
<td>Fire at 10:15am on the 15th day of every month</td>
</tr>
<tr>
<td>&quot;0 15 10 L * ?&quot;</td>
<td>Fire at 10:15am on the last day of every month</td>
</tr>
<tr>
<td>&quot;0 15 10 * 6L&quot;</td>
<td>Fire at 10:15am on the last Friday of every month</td>
</tr>
<tr>
<td>&quot;0 15 10 ? 6L 2002-2005&quot;</td>
<td>Fire at 10:15am on every last friday of every month during the years 2002, 2003, 2004 and 2005</td>
</tr>
<tr>
<td>&quot;0 15 10 ? 6#3&quot;</td>
<td>Fire at 10:15am on the third Friday of every month</td>
</tr>
</tbody>
</table>

**Misfires**

When a trigger misses its firing time due to reasons such as the Scheduler was paused or shut down, this is refereed to as a **misfire**. Misfires are determined by the triggers misfire instruction. There are trigger misfires types available for all trigger types and those that are specific to "simple" and "cron" trigger types.

All Triggers - These misfire instructions are applicable to any trigger.
**MISFIRE_INSTRUCTION_SMART_POLICY**: This misfire instruction is the default for all triggers created. Essentially this instructs the trigger to use a default policy dependent on the type of trigger that is created. For a "simple" trigger the rule is as follows:

- If the Repeat Count is 0 then the instruction will be interpreted as **MISFIRE_INSTRUCTION_FIRE_NOW**.
- If the Repeat Count is **REPEAT_INDEFINITELY**, then the instruction will be interpreted as **MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_REMAINING_COUNT**. **WARNING**: using **MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_REMAINING_COUNT** with a trigger that has a non-null end-time may cause the trigger to never fire again if the end-time arrived during the misfire time span.
- If the Repeat Count is 0, then the instruction will be interpreted as **MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_EXISTING_REPEAT_COUNT**.

**INSTRUCTION_RE_EXECUTE_JOB**: Instructs the Scheduler that the Trigger wants the JobDetail to re-execute immediately.

**INSTRUCTION_SET_TRIGGER_COMPLETE**: Instructs the Scheduler that the Trigger should be put in the COMPLETE state. It essentially skips the misfired trigger.

**INSTRUCTION_DELETE_TRIGGER**: Instructs the Scheduler that the Trigger wants itself deleted.

**INSTRUCTION_SET_TRIGGER_ERROR**: Instructs the Scheduler that the Trigger should be put in the error state.

Simple Triggers – These misfire instructions are applicable to only “simple” triggers.

**MISFIRE_INSTRUCTION_FIRE_NOW**: Instructs the Scheduler that upon a misfire situation, the trigger wants to be fired now by the Scheduler. **NOTE**: This instruction should typically only be used for 'one-shot' (non-repeating) Triggers. If it is used on a trigger with a repeat count > 0 then it is equivalent to the instruction **MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_REMAINING_REPEAT_COUNT**.

**MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_EXISTING_REPEAT_COUNT**: Instructs the Scheduler that upon a misfire situation, the trigger wants to be re-scheduled to with the repeat count left as-is. **NOTE**: Use of this instruction causes the trigger to 'forget' the start-time and repeat-count that it was originally setup with. **NOTE**: This instruction could cause the Trigger to go to the 'COMPLETE' state after firing 'now', if all the repeat-fire-times where missed.

**MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_REMAINING_REPEAT_COUNT**: Instructs the Scheduler that upon a misfire situation, the trigger wants to be re-scheduled to 'now' with the repeat count set to what it would be, if it had not missed any firings. **NOTE**: Use of this instruction causes the trigger to 'forget' the start-time and repeat-count that it was originally setup with this is only an issue if you for some reason wanted to be able to tell what the original values were at some later time. **NOTE**: This instruction could cause the trigger to go to the 'COMPLETE' state after firing 'now', if all the repeat-fire-times where missed.

**MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_REMAINING_COUNT**: Instructs the Scheduler that upon a misfire situation, the trigger wants to be re-scheduled to the next scheduled time after 'now' and with the repeat count set to what it would be, if it had not missed any firings. **NOTE/WARNING**: This instruction could cause the trigger to go directly to the 'COMPLETE' state if all fire-times where missed.

**MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_EXISTING_COUNT**: Instructs the Scheduler that upon a misfire situation, the trigger wants to be re-scheduled to the next scheduled time after 'now' and with the repeat count left unchanged. **NOTE**: Use of this instruction causes the trigger to 'forget' the repeat-count that it was originally setup with. **NOTE/WARNING**: This instruction could cause the trigger to go directly to the 'COMPLETE' state if all fire-times where missed.

CRON Triggers – these apply to the “cron” type triggers
**MISFIRE_INSTRUCTION_FIRE_ONCE_NOW**: Instructs the Scheduler that upon a misfire situation, trigger wants to be fired now by the Scheduler.

**MISFIRE_INSTRUCTION_DO NOTHING**: Instructs the Scheduler that upon a misfire, the trigger wants to have its next-fire-time updated to the next time in the schedule after the current time (taking into account any associated `<code>`{-link Calendar}`</code>), but it does not want to be fired now.
Datasources

The datasources components allow access a wide variety of datasource types. Each type of implementation allows the query of a specific datasource using a query string in that datasources natural syntax. For instance SQL data uses SQL syntax for the query. MDX data uses MDX syntax for the query and XML data uses XQuery as its query syntax. The datasources components include the following implementations:

- **MDXLookupRule**: For querying multidimensional data (current implementation is for mondrian)
- **SQLLookupRule**: For querying SQL based data.
- **XQueryLookupRule**: For performing XQuery selections against an XML document.

MDXLookupRule

The MDX lookup provides a facility to query multidimensional datasources using the the MDX query structure. A sample MDXLookupRule action sequence is shown below:

```xml
<action-sequence>
  <name>MDX_Datasource.xaction</name>
  <version>1</version>
  <title>%title</title>
  <logging-level>debug</logging-level>
  <documentation>
    <author>William Seyler</author>
    <description>%description</description>
    <help></help>
    <result-type>rule</result-type>
    <icon>MDX_Datasource.png</icon>
  </documentation>
  <outputs>
    <rule-result>
      <type>list</type>
    </rule-result>
  </outputs>
  <resources>
    <catalog>
      <url>
        <location>samples/reporting/SampleData.mondrian.xml</location>
        <mime-type>text/xml</mime-type>
      </url>
    </catalog>
  </resources>
  <actions>
    <action-definition>
      <connection>[jdbc:hsqldb:hsql://localhost:9001/sampledata]]>
      <location>mondrian</location>
      <user-id>sa</user-id>
      <password></password>
```

```xml
</component-name>
  </connection>
  </location>
  </user-id>
  </password>
```
The MDX query requires the following inputs.

**<connection> NOT REQUIRED** - This is a JDBC connect string for the desired datasource. If you need connectivity to a datasource other than hypersonic you must provide the JDBC drivers in a connection string see `<mdx-connection-string>` below for information on how to do this.

**NOTE**: Either this node or the `<mdx-connection-string>` must be supplied.

**<query> REQUIRED** - This is the MDX query string that defines the desired dataset. For more information about MDX query strings consult the mondrian documentation or visit: http://www.informit.com/articles/article.asp?p=29418&seqNum=3&rl=1

**<catalog> NOT REQUIRED** - The ROLAP catalog path. If the catalog starts with HTTP then this path is evaluated as an absolute URL and the system will attempt to load the catalog directly. If the catalog does not start with HTTP then the path is assumed to be from the solution root and a URL is constructed accordingly and passed to the datasource engine (mondrian).

**NOTE**: Either this node or the `<mdx-connection-string>` must be supplied.

**<location> NOT REQUIRED** - Always “mondrian” for the current implementation.

**NOTE**: Either this node or the `<mdx-connection-string>` must be supplied.

**<user-id> NOT REQUIRED** - The JDBC user ID. If using other than hypersonic as the datasource then this may not evaluate correctly and you may need to use the `<mdx-connection-string/>` (see below) to set the user ID and password.

**<password> NOT REQUIRED** - The JDBC password If using other than hypersonic as the datasource then this may not evaluate correctly and you may need to use the `<mdx-connection-string/>` (see below) to set the user ID and password.

**<mdx-connection-string> NOT REQUIRED** - The mdx-connection-string can be used in lieu of all the above connection properties and in fact will override the above properties should any exist. This string is a semicolon separated named properties list that is parsed by mondrian to create the connection. Here is an example for a connection based on the mdx-connection-string:

```xml
<mdx-connection-string>
  <![CDATA[
    Provider=mondrian; jdbc=jdbc:odbc:MondrianFoodMart; Catalog=/WEB-INF/FoodMart.xml
  ]]> 
</mdx-connection-string>
```
NOTE: When using this format the user name and password could be passed in as part of the jdbc connection string. This is dependent on the driver being used. Driver names can also be passed in on this string. More information about this type of connection string consult the mondrian documentation.

NOTE: If `<mdx-connection-string>` is not defined then `<connection>` is required.

SQLLookupRule

The SQLLookupRule component is used to connect to JDBC compliant datasources. A sample action document using SQLLookupRule is show below:

```
<action-sequence>
    <name>SQL_Datasource.xaction</name>
    <version>1</version>
    <title>%title</title>
    <logging-level>debug</logging-level>
    <documentation>
        <author>James Dixon</author>
        <description>%description</description>
        <help></help>
        <result-type>rule</result-type>
        <icon>SQL_Datasource.png</icon>
    </documentation>
    <inputs>
        <dept type="string">
            <default-value>Product Development</default-value>
        </dept>
    </inputs>
    <outputs>
        <rule-result>
            <type>list</type>
        </rule-result>
    </outputs>
    <resources/>
    <actions>
        <action-definition>
            <action-inputs>
                <dept type="string"/>
            </action-inputs>
            <action-outputs>
                <rule-result type="list"/>
            </action-outputs>
            <component-name>
                org.pentaho.component.SQLLookupRule
            </component-name>
            <action-type>rule</action-type>
            <component-definition>
                <jndi>SampleData</jndi>
                <query>
                    <![CDATA[select REGION, MANAGER_NAME, EMAIL from DEPARTMENT_MANAGERS]]>
                </query>
            </component-definition>
        </action-definition>
    </actions>
</action-sequence>
```
<jndi> NOT REQUIRED - This tag is the name of a valid JNDI datasource.
NOTE: If you don't have a jndi datasource then you must use the <connection/> tag and supply a valid JDBC connection string.

<connection/> NOT REQUIRED - This tag is a valid JDBC connection string. When using this method to connect to a database you must ensure that the proper JDBC driver jar is available in the current classpath.
NOTE: You must either provide <connection> or <jndi>

<query> REQUIRED - This is the SQL query string that returns the requested data.

<driver> NOT REQUIRED - This is the driver name that the connection string requires. Not required for JNDI or if you are using the default Hypersonic Database.

<user-id> NOT REQUIRED - JDBC user ID. This may also be passed in as part of the <connection> string.

<password> NOT REQUIRED - JDBC password. This may also be passed in as part of the <connection> string.

<live> NOT REQUIRED - If set to true then no caching of the result set is performed. If missing or set to false then a cached dataset is returned. NOTE: When using a cached dataset the data changes made after the query won't be reflected in the result set.

XQueryLookupRule

The XQuery lookup rule allows XQuery to be performed against an XML document. The component will do its best at returning a result set with the proper headers. Below is an example of an XQueryLookupRule.

<action-sequence>

  <name>XQ_Datasource.xaction</name>
  <version>1</version>
  <title>%title</title>
  <logging-level>debug</logging-level>
  <documentation>
    <author>William Seyler</author>
    <description>%description</description>
    <help></help>
    <result-type>rule</result-type>
    <icon>XML_Datasource.png</icon>
  </documentation>

  <outputs>
    <rule-result>
      <type>list</type>
    </rule-result>
  </outputs>

  <resources>
    <document>
      <solution-file>
        <location>books.xml</location>
        <mime-type>text/xml</mime-type>
      </solution-file>
    </document>
  </resources>

  <actions>
<action-definition>
  <action-outputs>
    <rule-result type="list"/>
  </action-outputs>
  
  <component-name>
    org.pentaho.component.XQueryLookupRule
  </component-name>
  <action-type>rule</action-type>
  <component-definition>
    <query><![CDATA[/bookstore/book]]></query>
  </component-definition>
</action-definition>

<query> REQUIRED - The only currently supported tag. This is the full XQuery selection string. If no document is defined then the query string is evaluated as an absolute path. If a document is defined then the solution path is appended to the documents fully qualified path.
Appendix A – Action Sequence XML in Detail

XML Nodes marked as REQUIRED are only required if their parent node is being used. Attributes shown in square brackets [ ] are optional.

<action-sequence> REQUIRED - Top level node for the Action Sequence Document

<name> REQUIRED - The name of the Action Sequence, it must match the file name of the document.

<version> NOT USED - The version of this document

<title> NOT REQUIRED - Friendly name of the document. Used for display only

<logging-level> NOT REQUIRED - Sets the logging level for the entire Action Sequence. Valid values are: TRACE, DEBUG, INFO, WARN, ERROR and FATAL. If no logging level is set, ERROR will be used.

<documentation> NOT REQUIRED - Contains descriptive nodes used for generating documentation.

<author> - NOT REQUIRED - The author of this Action Sequence

<description> - NOT REQUIRED - Short (1-3 lines) description of the Action Sequence. This description is used by the solution navigation component to generate its display.

<help> - NOT REQUIRED - Long Description of the Action Sequence including instructions for it’s use by an end user.

<result-type> - NOT REQUIRED - Type of output this Action Sequence will generate. It is used by the solution navigation component to generate its display. Action Sequences without a result-type will not be displayed by the navigation component. Valid values are: Report, Process, Rule, View and None.

<icon> - NOT REQUIRED - Thumbnail image that the navigation component will use for generating its display. The path to the image is relative to the directory that the ActionSequence document is in. For example: Example1_image.png

<inputs> - NOT REQUIRED - Collection of input parameters.

<param-name type="data-type"> - NOT REQUIRED - param-name is the name of a parameter that the Action Sequence is expecting to be available at run time. The type attribute specifies the data type of this parameter. See below for valid data types.

<default-value> - NOT REQUIRED - Allows the input parameter to specify a default value if a value has not been supplied. If the default-value node is present but has no value specified, the user will be prompted for the value if possible.

<sources> - NOT REQUIRED - list of parameter providers in the order they should be queried to obtain a parameter. Valid values are request, session and runtime.

Note: if a param-name is set but default-value and sources are both not specified, a validation error will occur.

<outputs> - NOT REQUIRED - Collection of output parameters.
<param-name type="data-type"> - NOT REQUIRED - param-name is the name of a parameter that the Action Sequence is expecting will be set by the time all action-definitions have executed. The type attribute specifies the data type of this parameter. See below for valid data types.

<logging-level> NOT REQUIRED - Sets the logging level during this execution of the action-definition. Valid values are: TRACE, DEBUG, INFO, WARN, ERROR and FATAL. If no logging level is set, ERROR will be used.

<resources> - NOT REQUIRED - Collection of resource parameters.

<resource-name> - NOT REQUIRED - resource-name is the name of a resource that the Action Sequence is expecting to use. The type attribute specifies the data type of this parameter. See below for valid data types.

<resource-type> - REQUIRED - The name of the type of resource required. Valid values are: solution-file, file and url.

<location> - REQUIRED - The path to the resource. For a resource-type of "solution-file", the location is a pathname relative to the top level of the current solution. If the resource-type is "file" then the location is assumed to be the a fully qualified path. For resource-type of "url" the location is assumed to be a fully qualified URL.

<mime-type> - NOT REQUIRED - Gives a hint about the mime type of the resource.

<actions [loop-on="parameter-name"]> - REQUIRED - The actions node contains "action-definition" nodes and optionally more "actions" nodes.

The loop-on attribute is optional. When it is used, the nodes within "actions" will be executed multiple times. It is necessary to specify a parameter that is of type list (string-list or property-map-list) and the group of nodes that will be executed once for each element in the list. An input parameter will be generated with the same name as the loop-on attribute but it will have the value of one element in the list. For example: if a loop-on attribute named “department” is a string-list with department names, then a parameter named department will be available and be set to a different department name for each iteration.

<actions [loop-on="parameter-name"]> - NOT REQUIRED - Since a single level of looping is not very fun, actions nodes can be nested within actions nodes to any level desired - no matter how silly it may be to do so.

<action-definition> - REQUIRED (At least 1) - It defines one complete call to a component for execution of a task.

<action-inputs> - NOT REQUIRED - Collection of action-input parameters.

<input-name type="data-type" mapping="param"> - NOT REQUIRED - input-name is the name of a parameter that the Action Definition is expecting to be available at run time.

The type attribute specifies the data type of this parameter. See below for valid data types.
The mapping attribute allows this input to be mapped to an Action Sequence input or a previous action-definition output with a different name.

<action-outputs> - NOT REQUIRED - Collection of action-output parameters.

<output-name type="data-type"> - NOT REQUIRED - output-name is the name of a parameter that the Component will have set by the time it finishes executing. The type attribute specifies the data type of this parameter. See below for valid data types.

<component-name> - REQUIRED - The fully qualified java class name for the component to execute.

<component-definition> - REQUIRED - The component specific XML definition. See the documentation for the specific component for more information. This node may be empty but it must exist or a validation error will occur.
Appendix B

**Action Sequence Data Types**

The following data types are currently supported by the Pentaho BI Platform.

**string** - The standard stinky old Java String.

Example: This XML node defines a string with a default value of “Central.” The RuntimeContext will first look for an input parameter named “REGION” in the http request. It will then ask the session for an object named “aRegion.” If neither have a value it will create a string set to “Central”.

```
<region type="string">
  <default-value>Central</default-value>
  <sources>
    <request>REGION</request>
    <session>aRegion</session>
  </sources>
</region>
```

**long** - A Java Long Object.

Example: This XML node defines a long with a default value of 25.

```
<amount type="long">
  <default-value>25</default-value>
</amount>
```

**string-list** - A list of Java String Objects.

Example: This XML node defines a string-list with the name “to-address” with 4 entries. Items in the list are contained within <list-item> nodes.

```
<to-address type="string-list">
  <default-value type="string-list">
    <list-item>joe.pentaho@pentaho.org</list-item>
    <list-item>admin@pentaho.org</list-item>
    <list-item>sales@pentaho.org</list-item>
    <list-item>noxidj@pentaho.org</list-item>
  </default-value>
</to-address>
```

**property-map** - A property map of Java Strings.

Example: This XML node defines a property-map with the name “veggie-data” with 4 name value pairs. Items in the list are contained within <entry key="xxx"> nodes. Property maps are sometimes used to represent a single row of data from a database query. The keys map to column names and the value maps to that column’s data.

```
<veggie-data type="property-map ">
  <default-value type="property-map">
    <property-map>
      <entry key="name">carrot</entry>
      <entry key="color">orange</entry>
      <entry key="shape">cone</entry>
      <entry key="texture">bumpy</entry>
    </property-map>
  </default-value></veggie-data>
```
**property-map-list** - A list of property maps of Java Strings.

Example: This XML node defines a property-map with the name “fruit-data” with 3 property-map sets. Items in the list are contained within `<entry key="xxx">` nodes. Property map lists are sometimes used to store the result of a database query. Each property map in the list represents 1 row of data with the keys mapping to column names and the values mapping to data cells.

```xml
<fruit-data type="property-map-list">
  <default-value type="property-map-list">
    <property-map>
      <entry key="name">orange</entry>
      <entry key="color">orange</entry>
      <entry key="shape">sphere</entry>
      <entry key="texture">dimply</entry>
    </property-map>
    <property-map>
      <entry key="name">grapefruit</entry>
      <entry key="color">Yellow</entry>
      <entry key="shape">sphere</entry>
      <entry key="texture">dimply</entry>
    </property-map>
    <property-map>
      <entry key="name">cucumber</entry>
      <entry key="color">green</entry>
      <entry key="shape">ellipsoid</entry>
      <entry key="texture">smooth</entry>
    </property-map>
  </default-value>
</fruit-data>
```

**content** - Content is large chunk of data that is generated within a component. One example of content is a PDF file generated by the reporting component. You cannot specify a default value for content since it can be of any type and is represented internally as a byte stream.

Example: This XML defines a content node named ‘attachment’ that is expected to exist in the runtime context under the name “report-output”. In this example, the report component generated a document and stored it as ‘report-output’. The email component will embed this data as an attachment in an email.

```xml
<attachment type="content">
  <sources>
    <runtime>report-output</runtime>
  </sources>
</attachment>
```
Action Sequence Resource Types

The following resource types are currently supported by the Pentaho BI Platform.

**solution-file** - A file on the file system relative to the location of the current Action Sequence document.

```xml
<solution-file>
  <location>MyReport.rptdesign</location>
  <mime-type>text/xml</mime-type>
</solution-file>
```

**file** - An absolute path on the file system.

```xml
<file>
  <location>D:\samples\reporting\MyReport.rptdesign</location>
  <mime-type>text/xml</mime-type>
</file>
```

**url** - A URL.

```xml
<file>
  <location>http://www.myserver.com/logo.png</location>
  <mime-type>image/png</mime-type>
</file>
```
Appendix C

Component Definitions

The components in the Pentaho BI Platform can be thought of as building blocks with inputs, outputs and a definition. Parameters that will be made available to a component are defined in the ‘action-inputs’ node. The component does not have access to any parameters that have not been specified in the ‘action-inputs.’

So where do the action-inputs come from? The action-inputs can come from two places: parameters defined in the ActionSequence document’s ‘inputs’ and any parameters defined in a previous components ‘action-outputs’.

The ‘action-outputs’ define what parameters will be available in the runtime context when the component has finished executing. Other components that execute later on can use these outputs as inputs.

When the Action Sequence finishes executing, the runtime data will be persisted to the runtime repository and will only save the parameters defined in the ActionSequence document’s ‘outputs’ node.

Each component in the Pentaho BI Platform has unique and specific inputs and component definitions that must be correctly specified. They also have optional inputs and definitions. This section describes their interfaces.

**HelloWorldComponent**

HelloWorldComponent is a very simple component. It simply returns the text “Hello World” followed by the string between the “quote” tags.

- **component-name**: org.pentaho.component.HelloWorldComponent.java
- **action-inputs**: None
- **action-outputs**: None
- **component-definition**:

```xml
<quote>A message to display</quote>
```

**EXAMPLE**

```xml
<action-definition>
  <action-inputs/>
  <action-outputs/>
  <component-name>org.pentaho.component.HelloWorldComponent</component-name>
  <component-definition>
    <quote>Greetings from the Pentaho BI Platform.</quote>
  </component-definition>
</action-definition>
```
**EmailComponent**

EmailComponent is used to send text or HTML based emails that may contain attachments. If the data type of the `to` parameter is property-map, then the map is assumed to contain the name value pairs for `to`, `subject` and `from`. The parameter `attach` should contain the name of a parameter of type content that contains the attachment. `attach-name` is the file name that the attachment will appear to have when opened by the email client.

**component-name:** org.pentaho.component.EmailComponent.java

**action-inputs:**

REQUIRED
- `to` - string or property-map
- `subject` - string
- `message-plain` or `message-html` - string

OPTIONAL
- `from` - string *If not specified, the default from “email_config.xml” will be used*
- `cc` - string
- `bcc` - string
- `attach` - string
- `attach-name` - string

**action-outputs:** None

**component-definition:** None

**EXAMPLE**

```
<action-definition>
  <action-inputs>
    <to type="string"/>
    <from type="string"/>
    <subject type="string"/>
    <message-plain type="string"/>
  </action-inputs>
  <component-name>org.pentaho.component.EmailComponent</component-name>
  <component-definition/>
</action-definition>
```

For a complete Action Sequence example with HTML text and attachments, see samples/bursting/send-email.xaction from the solution test-solution
**JavascriptRule**

The JavascriptRule component will execute the Javascript defined in the component-definition script node. Parameters specified as inputs will be available to the script for use. The parameter **rule-result** will consist of a list of the values that have been returned by all the functions that have not been assigned to variables. In the example below, the function region is called but not assigned to a variable, it is the only function specified so upon completion of the component execution, rule-result will contain the value 'Central.'

**component-name:** org.pentaho.component.JavascriptRule.java

**action-inputs:**
- **OPTIONAL**
  - Any parameter specified will be available as a variable to the scripting engine.

**action-outputs:**
- **REQUIRED**
  - **rule-result** - object

**component-definition:**
- **script** the javascript to execute

**EXAMPLE**

```xml
<action-definition>
  <component-name>org.pentaho.component.JavascriptRule</component-name>

  <action-outputs>
    <rule-result type="string"/>
  </action-outputs>

  <action-type>rule</action-type>
  <component-definition>
    <script><![CDATA[
      function region() {
        return "Central";
      }
      region();
    ]]>    
  </script>

  </component-definition>
</action-definition>```
**SQLLookupRule**

The SQLLookupRule component will execute the SQL query defined in the component-definition query node. Parameters specified as inputs can be used to replace text in the query. The replacement string is in the form Brace open, name, close brace where name and the parameter name match. In the example, `{dept}` in the query will be replaced by the value specified by dept in the action-inputs.

**component-name:** org.pentaho.component.SQLLookupRule.java

**action-inputs:**

*OPTIONAL*

Any parameter specified can replace a text like `{name}` in the query where name matches the parameter name.

**action-outputs:**

*REQUIRED*

- rule-result - property-map-list

**component-definition:**

- jndi - If a JNDI database connection is provided it will be used in preference to the ‘connection’, ‘user-id’, ‘password’, and ‘driver’
- connection - If a valid JNDI connection is not provided this will be used to create a connection to the database
- user-id - If a valid JNDI connection is not provided this will be used to create a connection to the database
- password - If a valid JNDI connection is not provided this will be used to create a connection to the database
- driver - If a valid JNDI connection is not provided this will be used to create a connection to the database

**EXAMPLE**

```
<action-definition>
  <component-name>org.pentaho.component.SQLLookupRule</component-name>

  <action-inputs>
    <dept type="string"/>
  </action-inputs>

  <action-outputs>
    <rule-result type="string"/>
  </action-outputs>

  <component-definition>
    <jndi>sampledata</jndi>
    <connection>jdbc:hsqldb:hsql://localhost/sampledata</connection>
    <user-id>pentaho_user</user-id>
    <password>password</password>
    <driver>org.hsqldb.jdbcDriver</driver>
    <query><![CDATA[
      select distinct PositionTitle
      from quadrant_actuals
      where department='{'dep}'
      order by PositionTitle]]>
  </query>
</component-definition>
```

```
PrintComponent

PrintComponent is used to print reports and content to a named printer accessible from the computer hosting the solution engine. The content to print can be specified in one of two ways. (1) Specify the file as a `printFile` resource or component setting. (2) Have a previous action in the sequence output content to the parameter `report-output`. Currently, the `org.pentaho.jfree.JFreeComponent`, `org.pentaho.birt.BIRTComponent`, and `org.pentaho.jasper.JasperComponent` have the ability to generate report content as `report-output`. If no content to print is specified, the action sequence will fail.

**component-name:** org.pentaho.components.PrintComponent

**action-inputs:**

- **printFile** - string *If not specified, the report-output parameter is used.*
- **printerName** - string *If not specified, the default printer is used*
- **copies** - number

**action-outputs:**

- **last-printer-selected** - string *If not specified in the action-outputs and action-inputs, value is not set.*

**component-definition:** None

**EXAMPLE**

```xml
<action-definition>
  <action-inputs>
    <copies type="string"/>
    <orientation type="string"/>
    <printerName type="string"/>
  </action-inputs>
  <component-name>org.pentaho.component.PrintComponent</component-name>
  <action-type>print</action-type>
  <component-definition/>
</action-definition>
```

For a complete Action Sequence example with HTML text and attachments, see samples/bursting/send-email.xaction from the solution test-solution.
**JobSchedulerComponent**
See Scheduling Section Above.

**component-name**: org.pentaho.component.JobSchedulerComponent.java

**SchedulerAdminComponent**
See Scheduling Section Above.

**component-name**: org.pentaho.component.SchedulerAdminComponent.java

**BIRTReportComponent**
Please see Integrating BIRT Reports.

**component-name**: org.pentaho.birt.BIRTReportComponent.java

**JasperReportsComponent**
Please see Integrating JasperReports.

**component-name**: org.pentaho.jasper.JasperReportsComponent.java

**ContentRepositoryCleaner**
ContentRepositoryCleaner is used to remove stale items from the content repository. It takes only one input, a number that represents how long an item is allowed to reside in the content repository. ie. If the value 90 is used then items older than 90 days will be removed from the content repository.

**component-name**: org.pentaho.component.ContentRepositoryCleaner.java

**action-inputs**: REQUIRED
- **days_old** - The number of days an item should be retained in the Content repository.

**action-outputs**: 
- **delete_count** - The number of items deleted from the content repository.

**component-definition**: None

**EXAMPLE**

```xml
<action-definition>
  <action-inputs>
    <days_old type="string" />
  </action-inputs>

  <action-outputs>
    <delete_count type="string" />
  </action-outputs>
</action-definition>
```
<component-name>
  org.pentaho.component.ContentRepositoryCleaner
</component-name>
<action-type>rule</action-type>
<component-definition></component-definition>
</action-definition>

For a complete Action Sequence example with HTML text and attachments, see Advanced\Content\clean_repository.xaction from the solution test-solution