JobRunner - Control & User Interface

<Job>

The overall document tag.*Attributes:*

- **title** Specifies the job title. If not specified will use the file name. This appears on window and dialog titles.
- **autoclose** If YES closes the window on completion.
- **checkallfieldsdefined** If YES then errors are generated if {} and : substitutions use non-existent variables; otherwise no substitution takes place but no error is generated.
- **continuetransaction** Use if a job is run in an external transaction to prevent automatic rollback. For example, a job embedded in a Woodpecker wizard. See Transaction Control for more information.

Be careful copying old jobs, whilst “<job>” instead of “<Job>” will work in some cases, many things including libraries will not.

<Parameters>

Starts a block of <Parameter> child tags. Parameter values are stored as global variables. Note that a <Parameters> block does not execute children other than <Parameter> and <ParametersCancel>. The global variables created are available to all subsequently executed tags regardless of nesting.

If you want access to a parameter label later in the job, eg in an error message, use a variable to hold the label.

Example

```
<Job name="label test" title="label test" autoclose="YES">
  <SetVariable name="weekprompt" value="Select the week"/>
  <Parameters>
    <Parameter name="ppweek" type="N" value="1">{weekprompt}
    </Parameter>
  </Parameters>
  <MessageDialog text="{weekprompt}"/>
</Job>
```

<Parameter>

Each parameter within a <Parameter> parent tag. The tag text is the descriptive prompt. If missing the name attribute is used. *Attributes*

- **name** The variable name. Defaults to the tag text with spaces removed.
- **type** String, Date or Number. The default is String. See Field Types.
- **value** Default value. Date parameters may use 'relative' dates: 0 for today, 1/-1 for beginning of last month, 31 for end of current month 31/12 for end of current year etc. May be a Select statement. Note that when using a select statement, any parameters used in the statement must use the {} style substitution (rather than :) otherwise an error: " -335 Parameter datatype is unknown" will be received. *When using a {} substitution remember to quote strings and protect against embedded quotes or control characters.*
- **required** YES forces completion. Dates are always required.
- **combochoices** Either "select description,value from ..." or "description1:value1;description2:value2;...".
- **password** YES causes characters typed to be hidden.
- **multiselectsql** SQL statement in form`select cast(name as varchar(30)) as description, divisionid as id from division order by description'. Description and id must be used. Must be 30 and 20 varchars. Parameter is returned in form - `field IN (list of id's selected)'
- **multiselectfield** Name for field in returned IN statement. Example - tempdesk.divisionid. Both multi select attributes must be filled and the parameter is then required.

See Publishing Web Services for information about retrieving http parameters and headers when publish a web service.

<ParametersCancel>

If one of these appears as a child of a <Parameters> tag it will be executed if Cancel is pressed. If not, pressing Cancel causes the job to be cancelled.

<SetVariableBlob>

Sets a variable to a Base64 encoded blob. Gets the blob from the blobstore or a file. Sets LastSetVariableBlobSize to the non-encoded size. Use either blobid and blobclass OR filename for the source. *Attributes*

- **name** The name of the variable
- **blobid** The id in the blobstore
- **blobclass** The class in the blobstore
- **filename** If loading from a file, not the blobstore

<SetVariable>

Sets the value of a variable from either a SQL expression or a string or arithmetic on x1, x2 and operator or by one or more <Write...> commands. If no variable of the specified name exists a global variable is created. x1, x2 perform integer arithmetic only. If you need to do float arithmetic or date manipulation use a SQL expression. If you just need string concatenation use the Value attribute with {} substitutions. If neither sql nor value nor x1,x2 attributes are supplied, tag text will be looked for

and an export will be opened into which the variable contents can be ‘written’ by descendents.
Attributes

- **name** The name of the variable
- **sql** A SQL expression (with or without the word ‘select’). The value is the first column of the first row. Variables and parameters can be substituted into the SQL expression - use :VariableName to ensure *properly typed substitution* - so that (for example) embedded quotes to not generate errors.
- **database** For external databases. Omit for default.
- **queryname** Allows fast re-use of a query (see name attribute in <SQLQuery>)
- **value** The explicit value to set. Ignored if SQL is used.
- **ensureendswith** e.g. Set to \ to ensure that a path has a trailing backslash before appending the filename. Only works in combination with the value attribute.
- **htmlwrappedandsignedby** (IQX only) Set to a StaffID and the value text will be inserted into the HTML wrap structure with the appropriate signature. If you set this attribute to “NOBODY” the text will be wrapped but have no signature.
- **extractfilenamepart** If value contains a file pathname this will extract specified part of it: path (excludes filename, includes trailing \), filename (excludes path), extension (includes leading .).
- **type** Defaults to String. Ignored if SQL is used (type is taken from the SQL expression). See Field Types for the supported types.
- **x1** first expression. Ignored if SQL is used or value filled.
- **x2** second expression
- **operator** +-*/ defaults to +
- **watchlist** If YES and job hits a breakpoint then value will be displayed in watchlist in debugger. Items can also be added to/removed from watchlist by right-clicking on selected value while debugger stopped.
- **guid** If YES a GUID (without {}) is generated. Everything except name is ignored
- **regex** used to either extract a matched expression from or apply a substitution to the value (wrapping the regex in slashes is optional, allowing i, s and m options after closing /). If the **replacement** attribute is not specified the value assigned will be the first capturing round-bracketed group, if any, otherwise the whole matched expression. To make a group non-capturing do (?: …)
- **replacement** - replaces occurrences of regex. $0 will insert the matched expression into the replacement text, $1, $2 etc. each round-bracketed group Example This example shows the use of **regex** & **replacement** to repair a malformed xml file which has no top level tag and multiple format declarations

```
<Job autoclose="YES">
    <ImportFile filename="c:\temp\Originalfile.xml">
        <Read name="OriginalFile"/>
        <SetVariable name="NewFile" value="{OriginalFile}"
regex="(?s)(&lt;\?xml version=&quot;1\.0&quot;
encoding=&quot;UTF-8&quot;\?&gt;)" replacement=""/>
        <ExportFile filename="c:\temp\NewFile.xml">
            <Write>&lt;?xml version=&quot;1.0&quot;
encoding=&quot;UTF-8&quot;?&gt;</Write>
            <WriteXMLTag tag="Documents">
                <Write>{NewFile}</Write>
```

```
            </WriteXMLTag>
        </ExportFile>
    </ImportFile>
</Job>
```

Using SetVariable to interact with Woodpecker forms

SetVariable can be used to set values on a Woodpecker view. The name attribute is used to indicate the required field eg:

```
<SetVariable name="WPK_Q_Postcode" value="{PostCode}"/>
```

If the field is not on the master query of a form, the view name should be specified eg:

```
<SetVariable name="WPK_Q_ViewID.ItemID" value="whatever"/>
```

WPK Form switch values can also be set in this way eg:

```
<SetVariable name="WPK_F_SwitchName" value="whatever"/>
```

<If>

Tag children only executed if the comparison evaluates to True. Unlike IfSQL If does not use the database engine to perform the comparison. *Attributes*

- **x1** 1st expression
- **x2** 2nd expression
- **comparison**
  - **=** - the default
  - **<>** - or !=
  - **<**
  - **⇐**
  - **>**
  - **>=**
  - **@** - x1 contained in x2
  - **!@** - x1 *not* contained in x2
  - **%** - starts with
  - **!%** - does not start with
  - **matchregex** - x2 must be a regular expression - wrapping in slashes is optional, allowing i, s and m options after closing /
    Characters regarded as special by XML (< > “ ' &) must be *escaped*. This can be done by using JobRunner substitutions where available (*{LessThan} {GreaterThan}*

{DoubleQuote} {SingleQuote} {Ampersand}) or XML control codes (*< > " ' &*)

- **type** Defaults to String. See Field Types for details. **Warning:** If you are doing less than/greater than comparisons with numbers make sure you set type="N" or you will get unwanted results. **Field Types are case sensitive**.
- **ignorecase** YES or NO, strings only, defaults to NO
- **trim** YES or NO, strings only, trims both strings before comparing, defaults to NO

Examples

If MyValue less than or equal to 27:

```
<If x1="{MyValue}" x2="27" comparison="{LessThan}=" type="Numeric">....</If>
```

If MyValue greater than or equal to 16:

```
<If x1="{MyValue}" x2="16" comparison="&gt;=" type="Numeric">....</If>
```

If MyValue (trimmed) is empty (note comparison of "=" and type of "String" come from the defaults:

```
<If x1="{MyValue}" x2="" trim="YES">....</If>
```

If a value matches one of a list:

```
<If x1="{LastHTTPStatus}" x2="200,201" comparison="@">
```

using a REGEX comparison:

```
<Job title="Email Address Checker" autoclose="YES">
    <Parameters>
        <Parameter name="EM">Enter email address</Parameter>
    </Parameters>
    <If x1="{EM}" x2="[a-zA-Z0-9.!#$%{Ampersand}'*+/=?^_`{|}~-]+@[a-zA-Z0-9-
]+(?:\.[a-zA-Z0-9-]+)*$" comparison="matchregex">
        <MessageDialog>Email Address OK</MessageDialog>
    </If>
    <Else>
        <MessageDialog>Email Address fails check</MessageDialog>
    </Else>
</Job>
```

<IfSQL>

Tag children only executed if the SQL condition evaluates to True. *Attributes*

- **condition** A logical expression which can be used in a SQLAnywhere 'if' expression. Parameters are handled in the same way as in other SQL tags.

- **database** For external databases. Omit for default.
- **queryname** If this condition will be re-used multiple times, specify a unique name for it, and it will be prepared and retained which will improve performance.

<IfFileExists>

Tag children only executed if the specified file exists. *Attributes:*

- **filename**

<IfFileDoesNotExist>

Tag children only executed if the specified file does not exist. *Attributes:*

- **filename**

<IfFileReadOnly>

Tag children only executed if the specified file has the Read Only attribute set. *Attributes:*

- **filename**

<IfAnyRows>

Tag children only executed if the nearest ancestor Query has remaining rows.

<IfNoRows>

Tag children only executed if the nearest ancestor Query has NO remaining rows.

<ForEachRow>

Tag children executed for each remaining row in the nearest ancestor Query.

If there are no rows returned by the ancestor Query, then the tag children are NOT executed. It is not necessary therefore to wrap <ForEachRow> tags in an <IfAnyRows> tag UNLESS some other activity is required before or after the <ForEachRow> children are executed.

<NextRow>

Forces query to move to the next row. E.g. to ignore the first row or alternate rows.

<IfOKDialog>

An Ok/Cancel dialog. Tag children only executed if Ok clicked. *Attributes:*

- **text** The text of the dialog.

<IfCancelDialog>

An Ok/Cancel dialog. Tag children only executed if Cancel clicked. *Attributes:*

- **text** The text of the dialog.

<IfYesDialog>

A Yes/No dialog. Tag children only executed if Yes clicked. *Attributes:*

- **text**

<IfNoDialog>

As above but tag children only executed if No clicked. *Attributes:*

- **text**

<Else>

Tag children only executed if preceding <If...> tag evaluated false. Must be sibling not child of the If... tag. Some other tags like <ExportFile> execute an <Else> if the cancel button is pressed. Some like <HTTPRequest> do so on failure.

<MessageDialog>

An Ok dialog. *Attributes:*

- **text**
- **clipboard** if YES text is copied to clipboard. Default NO

Example

```
<MessageDialog>Display this message</MessageDialog>
```

Or

```
<MessageDialog text="Display this message"/>
```

Using a <SetVariable> value in a <MessageDialog>:

```
<MessageDialog>Display the SetVariable {myvalue}</MessageDialog>
```

<Message>

Updates the message on the background of the window using the tag text or the text attribute. If the messagelogfile attribute has been specified in an ancestor tag the text will also be appended to the specified file, prefixed by date and time see Logging. *Attributes:*

- **text**

Sometimes the message text is not updated immediately, especially if the next command is long running eg a query. In such cases it can be helpful to follow the <Message> command with a <Wait> command with a very short interval eg <Wait seconds="0.1"/>

<Title>

Updates the job title. This appears on window and dialog titles. *Attributes:*

- **text**

<Finish>

Immediate termination of job. If text attribute not specified final background message will say 'Finished'. *Attributes:*

- **text**

<Cancel>

Like <Finish> but final background message reads 'Cancelled' instead of 'Finished'.

<Loop>

Keep executing child tags until a <Break> tag is encountered.

<Break>

Break out of the nearest enclosing <Loop>. If not in a loop behaves like <Finish>.

<Block>

Normally an exception will cause the Job to terminate with a message. If you wish to handle exceptions place the protected code as descendants of a <Block> and the exception will just terminate the block (and execute the immediately following <Catch> if there is one).

<Catch>

Exception handling code. Should be the next sibling of a <Block>. The exception text is available in the Exception pseudo-field.

<Throw>

Allows you to generate an exception. *Attributes*

- **text** The exception text

<Local>

Allows the declaration and initialisation of local variables, as attributes of the tag. Descendant tags have access to these variables, and they hide any variables of the same name with external or global scope. Attributes of Local tags are accessed WITHOUT the usual $ prefix, and formatting attributes may NOT be set as attributes of a Local tag. Attributes: The local variables and their initial values.

Example

```
<Job>
  <SetVariable name="MyFirstVariable" value="Hello"/>
  <SetVariable name="MyNextVariable" value="World"/>
  <MessageDialog>Before Local Block: {MyFirstVariable}
{MyNextVariable}</MessageDialog>
  <Local MyFirstVariable="Goodbye" MyNextVariable="World">
    <MessageDialog>Inside Local Block: {MyFirstVariable}
{MyNextVariable}</MessageDialog>
  </Local>
  <MessageDialog>After Local Block: {MyFirstVariable}
```

```
{MyNextVariable}</MessageDialog>
</Job>
```

The screenshots show how the variables revert to the original value outside of the <Local> block

<Call>

Call a sub-routine. The tag text is the name of the routine – which should be a tag within /Job/Library. Attributes of Call are available as local variables within the called routine WITHOUT the usual $ prefix, and formatting attributes may NOT be set as attributes of a Call tag. The attributes of the CALLED tag are not used, but may be set with explanatory comments in the values, as a convenient way of documenting the routine. To pass values out of the routine use global variables or variables with a scope external to the routine (see Local). If temporary variables are used within the routine it is highly desirable to use a Local block to prevent unintentionally over-writing external variables.

Sub-routines may be in external files. Place <Include>filename.xml</Include> in /Job/Library. The procedures should be placed within /Job/Library of the included file. See example under <Include>.

Attributes The sub-routine arguments and their values.

New feature: instead of placing the procedure name in the tag text you can use a **procedure** or **proc** attribute to make it more readable.

<Library>

Container for sub-routines. Each the code for each sub-routine is contained in a tag with the name of the sub-routine within the Library tag. Sub-routines can reference each other. See Call for details of how to call Library sub-routines.

<Include>

Library sub-routines can be held in an external file. Place <Include>filename.xml</Include> in /Job/Library. The procedures should be placed within /Job/Library tag of the included file.

Example

Calling job:

```
<Job>
  <Parameters>
```

```
    <Parameter name="Operand" type="Numeric"/>
  </Parameters>
  <Call Operand="4">SquareRoot</Call>
  <MessageDialog>The square root of {Operand} is {Result}</MessageDialog>
  <Library>
    <Include>MathsFunctions.xml</Include>
  </Library>
</Job>
```

Library job (MathsFunctions.xml):

```
<Job>
  <Library>
    <SquareRoot>
      <SetVariable name="Result" sql="SQRT(:Operand)" type="Numeric"/>
    </SquareRoot>
  </Library>
</Job>
```

The outermost tag should be **J**ob not job. Although older jobs may be found with the lower case “j”, many JobRunner commands such as <Library> and <Include> will fail in such a job

<OpenForm>

Open an application form of the given id. Static forms are hard coded in the application, non-static forms are defined in Woodpecker. If a # version of the form exists that will be opened in preference. A single form switch can be passed as the tag text. Multiple form switches require a succession of Writeln tags. *Attributes*

- **formid** The id of the form
- **static** YES or NO

Note that static forms will not open if another instance with the same *formid* already exists.

To open an internal browser window that loads pages from the LocalWebPage table use formid INTERNALBROWSER. First parameter is the URL and the second is the form id, followed by the required window title, separated by a dollar symbol.

Example

```
<Job title="Show XYZ Login" autoclose="Yes">
  <SetVariable name="CurrentStaff" sql="UserStaffID" />
  <OpenForm formid="INTERNALBROWSER" static="YES">
    <Writeln>
<![CDATA[http://localhost:{HTTPServerPort}/iqx/page/xyz-login.html?StaffID={
```

```
CurrentStaff}&amp;RunListener=1&amp;Port={HTTPServerPort}]]>
    </Writeln>
    <Writeln>
       <![CDATA[XYZ$Xyz Login]]>
    </Writeln>
  </OpenForm>
</Job>
```

<CloseForm>

Close a Woodpecker form. *Attributes*

- **wpkformid** The form id
- **primarykey** If blank it will close the first instance of the wpkformid it finds
- **withoutsaving** YES or NO. Defaults to YES

<Wait>

A timed delay for external processing. *Attributes:*

- **seconds** Accepts reals e.g. 1, 1.0, 0.5. .5.

<PersistFields>

Saves the current row field values of the immediate parent query as variables of the same names. This means that they persist after the query is closed. This may simplify some jobs by reducing nesting. Note that variables are appended to the end of the field stack unlike fields and locals which are 'pushed' and 'popped' i.e. inserted at the beginning of the field stack and removed again on close of the query or local block. This behaviour can be observed in the debugger. Note that it will usually replace global variables but if you have local variables or library procedure arguments of the same name you may not get what you expect. Use sparingly and with caution.

Back to Technical Help Section list
Back to JobRunner user guide