CHAPTER 29

Template Language

29.1 Purpose

A template language is used to combine data values with text, in order to produce an email or a report. Each BPM application has a different set of UDA variables distinct to that application. In order to send email about the application to users, we need to combine the text formatting of the email with the data values of the UDAs.

To provide a robust, well tested and proven template language, we turned to the **Chunk Templates**[1] open source Java library. This library focuses exclusively on doing what a template language is to do, and it does not require you to learn a lot of new capability. Also, it is not depend on a lot of other libraries or languages so it is easy to integrate.

Chunk offers a straight forward and familiar syntax. You can, of course, place data values into the text but you can also use IF-THEN constructs, looping constructs, and you have a lot of options for formatting the data to look correct. Chunk offers two syntaxes, and we will also prefer and use exclusively the new syntax.

DXP will always provide a JSON structure to the template engine. The exact same JSON structures that are defined in the Agile Adapter REST API are used again for templates. This simplifies the documentation of what you need to know. The exact same process instance JSON object is used when generating an email with the process instance.

The most common use for templates at this time is for email messages. You can send an email from DXP using the Send Process Email rest command (see 22 Send Process Email on page 130). This command will read a template file, combine it with process data, and send it as email.

29.2 The Basics

To support the examples, consider the following abbreviated example process instance object:

```
{
  "pd": {
    "id": 5568,
    "name": "DeliveryProcess"
  },
  "pi": {
    "dueDate": 1490975220545,
    "id": 55992,
    "initiator": "kswenson@us.fujitsu.com",
    "name": "Giants Day Victory Celeb",
    "owners": ["johnb", "alfredh"],
    "priority": 8,
    "uda": {
      "CustomerName": "Staples Office Supply",
      "goldClub": true,
      "Salesman": "Ivan Contreras",
      "Items": [
```

[1] Find them at `http://www.x5software.com/chunk/wiki/`

```
          {  "name":"hat",        "price":85.00 },
          {  "name":"car",        "price":34995.00 },
          {  "name":"happiness",  "price":null }
      ]
    }
  }
}
```

You place values into text to form an email message. Here is an example.

```
<html><body>
<h1>Carmic Suppliers</h1>
3831 Nowhere Road, Heavenly Valley, California, 90666

{!--comments can be done like this and are omitted from output --}
Dear {$pi.uda.CustomerName|html},

Here is a confirmation for your order for {$pi.name|html}
which is due on {$pi.dueDate|date("mmm dd, yyyy")}.

<table>
{% loop in $pi.uda.Items as $anItem %}
<tr><td>{$anItem.name|html}</td><td>{$anItem.price}</td></tr>
{% endloop %}
</table>

{% if ($pi.uda.goldClub) %}
You have achieved Gold Club status with us.
{% endif %}

yours sincerely,
<h3>{$pi.uda.Salesman|onempty(Your Salesman)|html}</h3>

</body></html>
```

This is not a complete listing but an example of the most common elements. Here is what you can notice. The simple tags to embed data start with open-brace dollar sign and then the path to the value. This looks very much like a javascript path expression. Since the information you want is usually in the process instance, these generally start with ‘pi’. Since most of the data you want to embed is UDA values, you will see most of them start with ‘pi.uda’.

The string values that are placed are sent to an HTML filter. This will make sure that any special characters are properly encoded to be in HTML, and so that everything in the CustomerName or Salesman name are properly interpreted as string value, and not markup. There are other types of filters, such as the example for the data where the format for displaying the date is specified.

If you have an array of elements, you can loop over them. in this case, there is a table, and it creates a row for every element in the list. The loop syntax allow you to specify the list you want to iterate, and a variable name that can be used within the loop to address the current item. Then, within the loop that items members can be embedded in the output. There are more options, such as a special section that is included only if the list is empty, and another special section that is included on every element except the last so you can put things between items.

The conditional statement on the bottom outputs a statement or not depending on a boolean value. In this case, a goldClub boolean value can be true or false. If it is true, then the line between the if and endif will be included in the output. There is also an ‘else’ tag which works as you would expect.

29.3 Token Types

29.3.1 Data Token

The basic form of a data substitution token is brace, dollar sign, variable reference expression, and closing brace.

```
{$a}
{$a.b}
{$a.b.c}
{$a|html}
{$a.b|date}
{$a.b|date(MMM dd, yyyy)}
{$a.b|onempty(No Value Provided)}
{$a.b|onempty(Unknown)|lc|html}
```

The first is a simple variable expression, while the second and third will be expressions that identify sub-elements of higher level record. In these first three examples the data value from the variable will be placed into the output without any modification.

The fourth example makes use of a HTML filter. The vertical bar character separates the variable reference expression from the filter name. The effect of a filter is that the data before the bar is 'piped' to the filter which processes it and outputs the result. Filters may be chained together with the output of one filter being processed by the next.

The HTML filter is critically important if you are producing HTML. Those data values had characters that are special in HTML (such as "&" or "<") then including them in the output would cause HTML syntax problems. The HTML filter will scan all the characters, and then properly encode any characters that need special encoding. So a "<" would be converted to "<" the proper encoding for a less-than symbol such that it is not confused with the beginning of a tag. The properly formatted HTML will present to the viewer on screen exactly what was in the data value, without any of the characters being mistaken for markup control characters.

The fifth and sixth examples are date formatting filters, the first uses the default date format, and the sixth specifies the date format that you want to see. The date filter relies on the data expressed before it to be an long epoch value with the number of milliseconds since Jan 1, 1970 in it.

The seventh (onempty) is a powerful filter that checks that the variable has a value in it. If so, it outputs the value, but if there is no value, it outputs what you place in the parameter. This is very useful if you don't want to leave a field blank, and want to fill it in with something that indicates that no value had been provided.

The eighth example shows how filters can be chained together. In this case, the value is be replaced with "Unknown" if empty, then the result converted to lower case, and then the result of that encoded properly for HTML.

29.3.2 Comments

Comments start with an open-brace, exclamation mark and two hyphens. The close with two hyphens and a close-brace. All of the text, starting from the open-brace to the close-brace will be omitted from the output.

```
{!-- this is a comment --}
```

29.3.3 Data Filters

The template engine can be extended with more filters. The built-in filters and their usage are listed below.

`uc`	uppercase
`lc`	lowercase
`qs`	escape double and single quotes by adding a backslash before the quote character. Use this if you are outputting a value into a JavaScript section and you want to make sure that the result is a valid JS string literal.
`md5`	md5 hash
`sha1`	sha-1 hash
`base64`	base64 encode
`base64decode`	base64 decode
`url`	url encode. Use this if you are outputting a value into something that will be a URL
`urldecode`	lowercase
`html`	encode the characters in the string to be follow HTML syntax for those characters, specifically & < > " and '
`xml`	same as html
`defang`	removes HTML tag markers etc. to foil xss script-injection attacks
`translate`	translate via the localization engine
`type`	outputs STRING LIST OBJECT NULL CHUNK depending on the tag value type
`sprintf(%05.3f)`	applies sprintf formatting. Useful for formatting numbers
`selected(abc)`	outputs `selected="selected"` if the value of expression == abc
`checked(abc)`	outputs `checked="checked"` if the value of expression == abc
`indent(3)`	indents multi-line values by x spaces
`join(, )`	if the value resolves to an array, this will join the elements of the array with the specified separation value.
`get(0)`	if the value resolves to an array, this will select the specified element of the array
`alternate(even,odd)`	If the value before this is numeric, then if that value is even it outputs the first parameter, and if odd, it outputs the second parameter.
`s/[0-9]/#/g`	perl-style search and replace with regular expressions
`date(MM/dd, yyyy)`	formats a date using the supplied template according to Java DateFormatter rules

29.3.4 Conditional Expressions

Beyond putting data values into the text, the most commonly used command is the IF-THEN-ELSE structure. The commands look like the following.

```
{% if ($a.b) %}
This part included if a.b exists
{% else %}
This part included only if a.b does not exist
{% endif %}
```

The first block tests for the existence of a value. It will be considered true if a.b has a value that is not a boolean false value. A null will be considered false, but an empty string (“”) will be considered to be true. IF-THEN-ELSE blocks can be nested inside other IF-THEN-ELSE blocks.

```
{% if ($a.b == John) %}
This part included if a.b is equal to "John"
{% elseIf ($a.b =~ /Alex|Harry/) %}
This part if the regular expression is satisfied
{% elseIf ($a.b != Betty) %}
This part included if a.b is not "Betty"
{% elseIf ($a.b == $a.c) %}
You can compare a variable to another variable
{% elseIf ($a.b|lc == $a.c|lc) %}
You can use filters as well, in this case compare lower case versions
{% else %}
This part included only is nothing else before this matched
{% endif %}
```

The second block compares the variable to values. You can compare if values are equal (==) or not equal (!=) and you can use regular expressions (= and !). You can not compare if values are greater or less than each other. Also, you can not combine multiple expressions with logical-and or logical-or, but you should be able to accomplish the same thing by nesting conditional statements.

29.3.5 Looping

```
{% loop in $a.b as $x%}
This block repeated for every element.  Use {$x.c} to embed data
{% onEmpty %}
This block display ONLY if the array is empty
{% divider %}
This displayed only between items
{% endloop %}
```

This is the simple, basic loop. The variable a.b would hold a list of items. The variable x will be assigned each items for each pass through the text between the tags. Use x as a data insertion variable just like any other.

There are two special block you can define. ‘onEmpty’ is a block that will be included only if the list variable is empty. Don’t use the x variable because it will have no value if the list was empty. ‘divider’ is a block that is used between items. So if you have three items in the list, it will be output twice. There exist tokens for ‘endOnEmpty’ and ‘endDivider’ but there is no reason to use them since the blocks are not nestable within each other. Starting ‘divider’ ends the ‘onEmpty’ and vice versa.

```
{% loop in $team as $attr:$value %}
  the key is {$attr} and the corresponding value is {$value}
{% endloop %}
```

If you need to loop over a map (i.e. a regular JavaScript object with keys associated with values) this loop construct allows you to get both the key and value on the associative array.

29.3.6 Include

```
{% include headerSection %}
```

This will read, process, and include the specified file if it is found in the same folder that the current file is. This is particularly useful if all your emails have a common header or footer. The name you give here is without the extension, and the file is required to have an extension of “chtml” on the disk.

29.3.7 Debugging

```
<pre>
{$debugDump|html}
</pre>
```

This is a very useful debugging feature. It outputs the full current data context showing all the names of all the variables (tags) that are accessible. This allows you to verify the names of the tags that where given to the template processor at the time. Use this to make sure that you have spelled the names of UDA variables the same way the process definition does.

Because this is pre-formatted text, if you are generating HTML, you will want to preserve the indenting by using the <pre> tags around it, and be sure to encode the characters correctly for HTML.

29.3.8 Passthrough without processing

```
{% literal %}
This {text} will {% not %} be processed or $interpreted.
{% endliteral %}
```

Anytime you want to output something, and you do not need to substitute things in it, but the text might be confused as being template directives, you can surround the text with these tokens. All text between the start and the will be passed straight through to the output without processing. By design, the beginning and ending literal tags are included in the output. Thus you will probably want to hide these in HTML comments (if you are producing HTML).