Post JSON

Post HTTP permits the convertion from AspireObject input into a JSON document and then transmits that to a remote server. This is intended for servers which recieve JSON as input, such as MongoDB and elastics earch.

Post JSON	
Factory Name	com.searchtechnologies.aspire:aspire-post-http
subType	default
Inputs	A standard job with an embedded AspireObject
Outputs	A JSON document posted to a remote server

- Configuration
- Example Configuration
- JSON Transformation Script

Configuration

This section lists all configuration parameters available to configure the Post JSON component.

Element	Type	Default	Description
postUrl	String	-	Url to post
postJsonTransform	String	-	Path to the groovy transformation file
okayResponse	String	-	Encoded JSON response

Example Configuration

ElasticSearch Simple Indexing

Single document indexing (without batching).

```
<component name="PostElasticsearch" subType="default" factoryName="aspire-post-http">
        <postUrl>http://localhost:9200/testindex/testtype/</postUrl>
        <postJsonTransform>config/json/aspireToElasticsearch.groovy</postJsonTransform>
        <okayResponse><![CDATA[{"ok":true}]]></okayResponse>
</component>
```

ElasticSearch Bulk Indexing

Elasticsearch bulk indexing (batching).

```
<component name="PostElasticsearch" subType="default" factoryName="aspire-post-http">
        <postUrl>http://localhost:9200/_bulk</postUrl>
        <postJsonTransform>config/json/aspireToElasticsearchBulk.groovy</postJsonTransform>
        <okayResponse><![CDATA[{"ok":true}]]></okayResponse>
</component>
```

JSON Transformation Script

The purpose of the JSON Tranformation Script is to transform the input AspireObject into the appropriate JSON to be pushed to the remote server. As part of this transformation you can:

- Select fields
- Map field names
- Map values
- Create new JSON structure

In effect, the JSON transformation script allows you to restructure your AspireObject into any JSON structure at all, so that the JSON you send to the remote server can be exactly what you want it to be.

Getting Started

Here is a simple builder script for creating a piece of fixed JSON output:

```
builder.$object() {
  title doc.displayTitle;
  body doc.content;
  submitTime (new Date());
}
```

Produces the following JSON:

```
{
   "title":"Welcome To The World Wide Widget Company",
   "body":"The WWW company is a world-wide seller of widgets...",
   "submitTime":"2012-11-15T17:20:03+0000"
}
```

As you can see, the builder script is a template representation for the resulting JSON structure. It can contain nested JSON objects, lists, and name /value pairs.

The JSON Transformation Script is itself a Groovy Script which uses a special "Aspire JSON Builder" to create the JSON. See Groovy Builders for more information about Groovy Builders.

In Groovy the semi-colons are optional, as long as all the fields are on separate lines. The following produces the same output:

```
builder.$object() {
  title doc.displayTitle
  body doc.content
  submitTime (new Date())
}
```

Familiarize yourself with Groovy strings

Groovy strings with *double-quotes* allow you to embed Groovy variables into strings. Go here to learn more about Groovy strings.

For example:

```
builder.$object() {
    yearField "Year $doc.year"
}
```

Produces:

```
{"yearField":"Year 1990"}
```

You can even embed Groovy expressions into your strings. Note the use of the ${} notation below:

```
builder.$object() {
     yearField "Years $doc.year-${doc.year+9}"
}
```

Produces:

```
{"yearField":"Years 1990-1999"}
```

JSON Objects: $object

JSON Objects are hash maps of name/value pairs. These can be represented in several ways within the JSON Transformation Script.

Simple Objects

The $object variable is available for creating objects as needed.

```
builder.$object();
```

Produces:

```
 {}
```

Name:Value Pairs

Name value pairs are specified by simply putting the name and the value, one after the other, on a separate line:

```
builder.$object() {
  name 'Barack Obama'
  age 51
};
```

Produces:

```
 {"age":51,"name":"Barack Obama"}
```

Note that these are actually converted by Groovy into function() calls. This means that the above script could be written like this:

```
builder.$object() {
  name('Barack Obama');
  age(51);
};
```

And it produces the exact same output:

```
 {"age":51,"name":"Barack Obama"}
```

A top-level object with a single name/value pair can be created in a number of ways:

```
 builder.document();
```

Produces:

```
 {"document":null}
```

And:

```
 builder.document('Hello world!');
```

Produces:

```
 {"document":"Hello world!"}
```

Using Variables for Names

You can use Groovy strings with embedded $variables to make a variable name. For example, suppose you have a document which contains the following fields:

```
 "def doc = [year:1990, title:'Research Reports', titleType:'display'];\n" +
```

With the above document the following script:

```
builder.$object() {
    yearField "Years $doc.year-${doc.year+9}"
    "title-$doc.titleType" doc.title     // << THIS NAME HAS AN EMBEDDED VARIABLE
}
```

Will produce the following JSON:

```
{"yearField":"Years 1990-1999","title-display":"Research Reports"}
```

Use curly braces to create nested objects

Objects which are nested inside of other objects can be created with curly braces. Note that there needs to be at least one nested name/value pair for this to work.

For example:

```
builder.$object() {
   family {
     father {
        name 'Barack Obama'
        age 51
     }
     mother {
        name 'Michelle Obama'
        age 48
     }
   }
}
```

In this way, the JSON Transformation Script mimics the structure of the JSON you want to produce.

The above example produces:

```
{
  "family": {
      "mother": {
          "age": 48,
          "name": "Michelle Obama"
      },
      "father": {
          "age": 51,
          "name": "Barack Obama"
      }
  }
}
```

"$" is output for content with no name

It's possible to create a name/value pair with no name. When this happens, "$" will be used for the name.

For example:

```
builder.$object('Hello world!');
```

Produces:

```
{"$":"Hello world!"}
```

Note that this can also happen when "$object" is used inside of other objects and lists. For example:

```
builder.quotations() {
  $object "Hello world!"
}
```

Produces:

```
{"quotations":{"$":"Hello world!"}}
```

Use "$" to add content to objects

'$' can also be used inside your transformation script where it represents simple content (as opposed to a name/value pair).

For example, it can be used to represent a nested value, as follows:

```
builder.factorial() {
  'five' {
    def j = 1;
    for(int i = 1 ; i <= 5 ; i++) j *= i;
    $ j;
  }
}
```

Produces:

```
{"factorial":{"five":120}}
```

JSON Lists

Creating Lists with $list

The $list directive can be used to create lists.

To create a simple top-level list:

```
builder.$list();
```

Produces:

```
[]
```

A simple top-level list with a nested string:

```
builder.$list('Hello world!');
```

Produces:

```
["Hello world!"]
```

$list inside of objects

$list can be used inside of objects. For example:

```
builder.answer() {
  $list 42
}
```

Produces:

```
{"answer":[42]}
```

Another way to produce the same result:

```
builder.$object() {
  answer {$list 42 }
}
```

Produces:

```
{"answer":[42]}
```

Use "$" to add content to lists

If you want to add simple content to a list, use the "$" to indicate this. For example:

```
builder.fibonacci() {
  $list {
    def j0 = 0, j1 = 1;
    for(int i = 0 ; i < 10 ; i++) {
       $ j1
       def jtmp = j1;
       j1 = j0+j1;
       j0 = jtmp;
    }
  }
}
```

Produces:

```
{"fibonacci":[1,1,2,3,5,8,13,21,34,55]}
```

Creating List of Objects

The above structures can be combined to create lists of objects.

First, what happens when you specify name/value pairs inside of a list?

```
builder.ordinals() {
  def ord = [1:"first", 2:"second", 3:"third"];
  $list {
    for(int i = 1 ; i <= 3 ; i++) {  // NOTE:  PROBABLY NOT WHAT YOU WANT
      ordinal (ord[i])
      arabic  i
    }
  }
}
```

Produces:

```
 {"ordinals":[{"ordinal":"first"},{"arabic":1},{"ordinal":"second"},{"arabic":2},{"ordinal":"third"},
{"arabic":3}]}
```

Notice that every name/value pair is a separate JSON object. This is probably not what you want.

The problem here is that there is no way for the builder to know how to group name/value pairs together unless you specify it explicitly with a "$object" directive:

```
builder.ordinals() {
  def ord = [1:"first", 2:"second", 3:"third"];
  $list {
    for(int i = 1 ; i <= 3 ; i++) {
       $object {          // << NESTED $object TO GROUP NAME:VALUE PAIRS TOGETHER
         ordinal (ord[i])
         arabic  i
       }
    }
  }
}
```

This produces something more like what you might have been expecting:

```
{"ordinals":[{"arabic":1,"ordinal":"first"},{"arabic":2,"ordinal":"second"},{"arabic":3,"ordinal":"third"}]}
```

Lists of Lists

These are also perfectly possible using the $list nomenclature:

```
builder.ordinals() {
  def ord = [1:"first", 2:"second", 3:"third"];
  $list {
    for(int i = 1 ; i <= 3 ; i++) {
       $list {
         $ (ord[i])
         $ i
       }
    }
  }
}
```

Produces:

```
{"ordinals":[["first",1],["second",2],["third",3]]}
```

Multiple Values for a Name -> An Automatic List

If you specify multiple values for the same name, the name will be automatically converted into a list.

For example:

```
builder.properties() {
  server 'http://server1.com'
  server 'http://server2.com'
  server 'http://server3.com'
  user 'george'
  password '1234'
}
```

Produces:

```
{
  "properties": {
      "server": [
          "http://server1.com",
          "http://server2.com",
          "http://server3.com"
      ],
      "password": "1234",
      "user": "george"
  }
}
```

You can do the same thing with a nested loop:

```
builder.properties() {
  for(int i = 1 ; i <= 3 ; i++) {
    server "http://server${i}.com"
  }
  user 'george'
  password '1234'
}
```

This produces the exact same output as above. Notice the use of ${i} for an embedded variable inside of a JSON string. This can be very helpful when producing JSON output.

Multiple JSON objects

By using builder.flush() you can specify the transformer to have multiple separated JSON objects for each transform. This is used for elasticsearch bulk indexing.

```
builder.index() {
  '_type' "test"
  '_index' "posthttp"
}
builder.flush()
builder.$object() {
  url doc.fetchUrl;
  body doc.content;
  submitTime (new Date());
}
```

Produces:

```
{"index":{"_type":"test","_index":"posthttp"}}
{"url":"http://www.mydomain.com/", "body":"This is the content of my web page", "submitTime":"2012-11-15T17:
20:03+0000"}
```

The flush method is always required if you want separated JSON objects. It also adds each JSON object in a new line "\n".