KENETIX™

MODULES REFERENCE

globalscape®
unleashing the power of data

GlobalSCAPE, Inc. (GSB)	
	Corporate Headquarters
Address:	4500 Lockhill-Selma Road, Suite 150, San Antonio, TX (USA) 78249
Sales:	(210) 308-8267
Sales (Toll Free):	(800) 290-5054
Technical Support:	(210) 366-3993

Web Support: http://www.globalscape.com/support/

June 22, 2017

Table of Contents

Introduction

You've set up your Authentication and the framework of an Event/Action, but now you actually need to connect your Connector to an API. You can do this using Kenetix's pre-built library of declarative building blocks, known as ***modules***. Basically, modules lay out a set of steps that your Connector should execute at runtime, determining how user data is turned into API requests, and how API data is in turn rendered into a user-friendly format.

In the "Core" section of your method (or, in the case of webhook events, the "Start" and "Stop" sections) you can declare a chain of modules that will handle everything from HTTP requests to JSON parsing and data manipulation. Modules perform in the order they appear in a method.

All modules will generate a template when selected. That template takes this general form:

```
{
"brick": "module_class.module_name",
"id": "id_name",
"inputs": {
"input1": {
"_availableTypes" : [
type1,
type2
],
"_type": "type1",
"_array": true/false,
"_value": ""
},
"input2": {
"_availableTypes" : [
type1,
type2
],
"_type": "type1",
"_array": true/false,
"_value": ""
}
},
"outputs": {
"output1" : {
"_type": "",
"_array": true/false
}
}
}
```

Fields:

- "brick": This is the internal name for modules, and represents the class and function that you are choosing to use in the given module. This will be populated after selecting a module of your choice.
- "id": Is an optional parameter. It can either remain the calculated ID, or can be set to any value unique to the method the module exists within. When set, it can be used by other modules to refer to the output of that module.To reference this specific module in other modules, you can use the Mustache template {{[idName].[any of its output keys]}}
- "inputs": Represents the set data that will used inside of the module to complete its given process. For more information, see "Module Inputs".

- "outputs": Represents the set of data that has been processed by the module's function and is available to be used in other modules within the same method or accompanying Metadata method.

Module Inputs

"Inputs" represents the structure of data being passed into the module, and how those values will be referenced throughout the module. It may contain any number of inputs, but modules will only read a certain set of predefined inputs, as denoted in the individual module documentation. An "inputs" object will always load a template in this form:

```
"inputs": {
"input1": {
"_availableTypes" : [
type1,
type2
],
"_type": "type1",
"_array": true/false,
"_value": ""
},
"input2" : {
"_availableTypes" : [
type1,
type2
],
"_type": "type1",
"_array": true/false,
"_value": ""
}
}
```

In this format, the values of each key besides "_value" will auto-populate with the default values for the given module. The keys that are available are:

- "_availableTypes": The list of types this input object is allowed to be with the given module. You may change "_type" to any of the values given in this array. Changing this array will not affect your Connector in any way. If this field is absent, that means that the only type this input could be is the default type.
- "_type": The type of the object that will be input into this module's underlying function. This will auto-populate to the default type for that input. You may change the "_type" value to match any of the types listed in the "_availableTypes" array, depending on what type of object you would like input into the module.
- "_array": Whether or not this input is a collection. When marked true, this object will become a collection of the already denoted type from the "_type" key.
- "_value": The value that the input object will have. You will need to change "_value" to set the value for that input. This is the field you will be using the most throughout your Connector development experience.
- (Optional) "_defaultValue": The default value of the input object. May be set to any value as long as its type is consistent with that input's "_type" value.
- Depending on your input object's "_type" value, you may set its value in many different ways.

If the input's "_type" value IS NOT "flo", you may set the input object's value to any of the following forms, permitting the value you use is the same type as your input's "_type" value:

- A Mustache reference, i.e. {{objectName}} or {{moduleID.[outputFieldHere]}}. In this case, the input value is the exact JSON of objectName, and is not stringified or escaped.
- A string containing a reference, i.e. "Bearer {{auth.access_token}}". In this case, the input type is a string and the value of auth.access_token should be a string (or it will be converted). Under the hood, this performs a string concatenation.
- A string without Mustache, i.e. "application/json". In this case, the input type is string.
- An object. Any object is simply treated as an input of type "object". The object can contain keys with values that have any valid JSON. Both keys and values are parsed for Mustache references.
- An array. The array can contain any valid JSON as its values. Mustache references can be included in any value of the array.
- A number. Parsed literally.
- A boolean. Parsed literally.
- null. Parsed literally.

If the input's "_type" IS "flo", you may set "value" to any of the following forms:

- A single module, contained in the "_value" object. You may omit the "_availableTypes" and "_defaultValue" keys for that module's inputs. This will look like so:

  ```
  "flo": {
  "_type": "flo",
  "_value": {
  "brick": "object.construct",
  "inputs": {
  "input1" : {
  "_type" : "object",
  "_array" : true,
  "_value" : "{{reference_here}}"
  }
  },
  "outputs": {
  "output": {
  "_type" : "object",
  "_array" : "false"
  }
  }
  }
  }
  ```

- An array of modules, contained in the "_value" object.
- A string that refers to the ID of a method, e.g. "methodNameHere".

Notes:

- When accessing paths of a key like "prevData" or "input", which both represent objects passed into the current module from a previous module, an object.get is performed on the referenced key unless the output can be found on that exact module. For example, if prevData is an http.get method, it will expose body, statusCode, and headers as outputs. Thus, you can reference {{prevData.body}} and this will be able to grab the exact reference. Using {{prevData.body.ok}} will produce an object.get on prevData.body.
- Using Mustache with object keys results in another method being used to set the key generated by Mustache. Generally speaking you'll produce faster FLOs if you know the keys of your hardcoded object in advance.

Module Outputs

"Outputs" represents the values that are produced after running a given module. All keys inside of the outputs schema are available in any other module in the same method by using the Mustache template {{moduleID.[outputName]}}.

Outputs take a similar form as inputs, with a few changes. The values in the "outputs" object are now meant to act as a reference so that you can know the shape of what comes out of your module. Now, there are no "_value" or "_defaultValue" keys, although you may add these fields and change them, if you would like.

Outputs look like this:

```
"outputs": {
"output1": {
"_type": "",
"_array": true/false
},
"output2" : {
"_type": "",
"_array": true/false
}
}
```

The remaining fields are:

- "_type": The default type of the output object. You often will not need to change this, however, there are a few important cases where you would need to change the "_type" value.
- For example: The "Object.get" module and the "JSON.Scope" module will both output a value from an object, but the "_type" value will default to "object", in which case you may want to change that output's "_type" to match the type you are expecting from the object you are pulling from.
- "_array": Whether or not the output object is a collection.
- If you are developing an Action method, your last module cannot output a collection, because all Actions expect an "object" type. In this case, follow up the module that outputs a collection with a JSON.Render brick or an Object.Construct to create the object you need.

Note:

The type and format of an output object is crucial to Connector development, because the expected format of the "Output" section of a method must match whatever the output of the last module in that method is. If your Event/Action method is expecting an object with a string field called "X", your last module must output an object with a string field called "X".

Passing Data with Mustache

Kenetix uses Mustache templates to hash data, so that it may be passed around modules within a method. Using Mustache, you can reference the parameters and inputs of your Event/Action method from any module within that method. Additionally, you may pass any data from the output schema of any module into any other module in the same Event/Action method.

For example, if you're building an Action card that creates a new resource (e.g., building a new task), you'll probably need to take information that the user gives you and then send it to the API.

When you set up your inputs you may have included a field for the user to put in the name of their task:

```
{
"name": "Task",
"attributes": [
{
"name" : "Task Name",
"type": "string"
}
]
}
```

The resulting input object will look something like this:

```
{
"Task: {
"Task Name": ""
}
}
```

Later, you'll want to hash this data into the body of your API request. This is where Mustache comes in handy:

```
"body": {
"task_name": "{{input.Task.Task Name}}"
}
```

At runtime, the Mustache will be swapped out for the data that the user enters or drags over from another card.

Universal Mustache Tags

Inside of Kenetix, there are several Mustache tags that can be used anywhere within a method. These will come in handy as you develop your Connectors:

- {{moduleID.[fieldName]}}, This is how most data will be passed around. Modules and their data can be referenced directly by any other module inside of the Action/Event method this way. Additionally, Metadata methods referenced by the module (with moduleID) can access that module's output data using this Mustache reference. This is done by using the desired module's ID and attaching the desired output field via a path.

- {{params.[parameterName]}}, where "parameterName" is the name of your parameter object in the Parameters section of your method. Use this tag to refer to the value of any parameters that the user enters.
- {{input.HeaderName.FieldName}}, where HeaderName is the "name" value in a header object in your "input" section, and FieldName is the "name" value of a field you would like to reference within that header object.
    - Use this tag to refer to any data that the users enters as an input.
    - You can also use it to reference inputs inside of Helper Functions that have been explicitly declared from the calling module or schema.
- {{prevData.[outputField]}. The object "prevData" represents the module directly preceding the module that this reference is used inside of. For example, if you use {{prevData.outputFieldName}} in the second module inside of a method's Core section, it will grab the outputFieldName value from the first module in that method's Core section.

Note: All Mustache references are case-sensitive!

Conditional Templates

There are lots of cases where you may want to only include mustache templates if certain conditions are met. The most common is when you are updating a resource. If the user doesn't enter data for a certain field, it will be passed to the modules as an empty string. However, you don't want to overwrite data by sending an empty string to the API.

For example, if we're updating a contact, the input section of the card might look something like this…

```
{
"name": "Contact",
"attributes": [
{
"name": "Contact Name",
"type": "string"
},
{
"name": "Contact Email",
"type": "string"
}
]
}
```

...and the body of the API request looks like this:

```
"body": {
"contact_name": "{{input.Contact.Contact Name}}",
"contact_email": "{{input.Contact.Contact Email}}"
}
```

However, the user may not want to update all of these fields when they actually run the card. When the FLO actually runs, the resulting input object built by the card might look like this:

```
{
"Contact": {
"Contact Name": ""
"Contact Email": "demo@Kenetix.com"
}
}
```

Then, if the empty data is mustached into the body of the call to the API, the API would be sent an empty string:

```
"body": {
"contact_name": "",
"contact_email": "demo@Kenetix.com"
}
```

This would update the email like the user wants, but overwrite the existing contact_name with an empty string. Instead, you can use Mustache's built-in logic to check if the fields exist before templating them into the API request.

To help with this, Mustache can evaluate an expression with the # character. If the expression evaluates to true, then Mustache will include the value between the Mustache references (one opening with the # character, the other closing with the / character). You may also put a Mustache reference in between the two conditional templates. You can use the ^ character in place of the # character to run a function that will only include the value between the brackets if the expression evaluates to false.

Example:

```
"body": {
"{{#input.Contact.Contact Name}}contact_name{{/input.Contact.Contact Name}}":
"{{input.Contact.Contact Name}}",
"{{#input.Contact.Contact Email}}contact_email{{/input.Contact.Contact Email}}":
"{{input.Contact.Contact Email}}"
}
```

This time, when the user doesn't enter a value for a field, Mustache will check to see if the field exists before including it in the body of the request to the API. Now, the request to the API will look something like this:

```
"body": {
"": "",
"contact_email": "demo.Kenetix.com"
}
```

The contact_name field isn't included, since the user didn't enter a value. That way, the email will be updated, and the existing contact name will be left alone.