# KENETIX™

*BUILDING YOUR FIRST CONNECTOR*

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 21, 2017*

# Table of Contents

## Building Your First Connector, Part 1

This guide will be walking through "An API of Ice and Fire", a Game of Thrones API by Joakim Skoog. This RESTful API is great to use for an introductory connector to build because it is a well-documented, simple API that enables us to explore foundational connector concepts:

- Building connector UI with both collections and single entities
- Building a connector "Action" that handles a single entity, with collection and singleton fields
- Testing connector methods in the console
- Deploying and testing connectors in the main Kenetix platform

We should always design our connector before building it. We want to make sure we always know what we're building towards before we start development. "Designing" a connector means that we have to take into account:

- What methods we would like to build
- The inputs and outputs for those methods

Although this API offers several routes we can grab data from, we're only going to focus on building one method in this first walkthrough. We will build "Get Character" which returns a single character and all of their information from a character ID. API documentation here.

Now that we know what method we would like to build, let's figure out our inputs and outputs for it.

**Inputs**

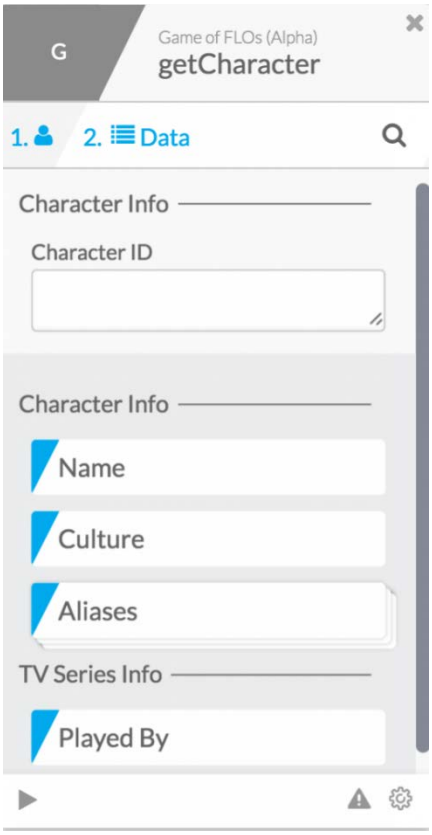
The API requires a character ID only for this route. So, we only need a single input.

**Outputs**

We get back a lot of information from this request, which can be viewed here. For this demo, we're going to output this information (with headings added for readability on the method card):

- Character Info
- Name
- Culture
- Aliases (collection of strings)
- TV Series Info
- Played By

Here's what we'd like the card to end up looking like.




Now we know what we are building towards, let's get started building it.

Once there, make sure you have "New Connector" selected in the top left menu.

## Connector Authentication

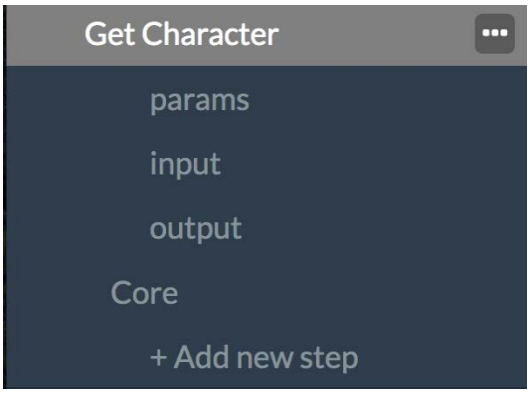
Normally when starting a connector, we would start by filling in the Authentication section of our connector (top of the left menu). However, we are building against an open API, so there is no authentication information needed for this connector.

## Creating Our Method

Now, we'll start on the "Get Character" method. To start, click **Actions**, and then **Add Method**. This method will be an Action, because this API interaction will only ever need to take place after a FLO is triggered. If we had wanted to build a method that would trigger FLO, we would have done these exact same steps with the Events list.

After you've named your method and (optionally) added your description, you will see a bunch of information pop up on the left menu and in the editor. This means that the connector builder has generated the template of your method for you. Now we've just got to fill that template in with what we need.

If you look at our new method on the left menu, you can see two sections: **params/input/output** and **Core**.




**Params**, **input**, and **output** are all places where you define the UI elements of your method's card. The **Core** section is responsible for all data processing.

## UI

Let's define the UI for our method's input. Click **input**, underneath the name of your method. You should see this:

```
{
   "extensible": false,
   "attributes": [
     {
       "name": "",
       "attributes": [
         {
           "name": "",
           "type": "string"
         }
       ]
     }
   ]
 }
```

As you can see, there are a couple of layers to our input object. The top-level attributes represents each group of inputs, as designated by a header. Inside of this is another layer of attributes, which designate the items inside of each group of inputs. If you look back at the card we designed, we wanted a "Character ID" input that was underneath a header called Character Info. So, we want to name our top item "Character Info" and name an attribute inside of it "Character ID."

This will look like this:

```
{
  "extensible": false,
  "attributes": [
    {
      "name": "Character Info",
      "attributes": [
         {
            "name": "Character ID",
            "type": "string"
         }
      ]
    }
  ]
}
```

Nicely done! We've completed our first bit of UI. Now, onto the output section of our card.

Our "output" section is a bit more complicated. There are more outputs, as well as support for collection fields ("aliases") and multiple headers ("Character Info" & "TV Series Info").

Let's fill in and add the names of all the headers and inputs, besides aliases using what we learned before.

You should end up with something like this:

```
{
  "extensible": false,
  "attributes": [
    {
      "name": "Character Info",
      "attributes": [
        {
          "name": "Name",
          "type": "string"
        },
        {
          "name": "Culture",
          "type": "string"
        }
      ]
    },
    {
      "name": "TV Series Info",
      "attributes": [
        {
          "name": "Played By",
          "type": "string"
        }
      ]
    }
  ]
}
```

**Note**: if we had fields with different types, we could change those types in the "type" field.

Now that we have the basics done in our "output" section, we can add the outputs that act as collections, the aliases field.

To add "alliances," which is a simple collection of strings, add another item inside of "Character Info," the same as you would with a "normal" output. Then, add a field called "collection" and set it true. This tells the UI to treat the output like a collection.

You should end up with something like this:

```
{
  "extensible": false,
  "attributes": [
    {
      "name": "Character Info",
      "attributes": [
        {
          "name": "Name",
          "type": "string"
        },
        {
          "name": "Culture",
          "type": "string"
        },
        {
          "name": "Aliases",
          "type": "string",
          "collection": true
        }
      ]
    },
    {
      "name": "TV Series Info",
      "attributes": [
        {
          "name": "Played By",
          "type": "string"
        }
      ]
    }
  ]
}
```

Ta da! We've finished all the UI for this method.

## Core

Now that we have the UI of our method done, let's get started on actually building out the functionality of our connector. A connector wouldn't do any good if all it did was render a card in the Kenetix FLO tool, we need it to also grab data from our service, and then format that data in a way that will align with the UI we just built.

Inside the Kenetix connector builder, functionality is determined through a chain of "modules." Each module begins as JSON, and is compiled into Javascript. Inside the original JSON definition, we set the inputs into that underlying Javascript function to define how we would like it to run. After the underlying Javascript function is run, we receive an output that represents the results of that function.

For example, a JSON.Parse module would require the developer to define the stringified objects they would like parsed as an input, and can expect back a parse JSON object as an output.

Let's take a look at what modules we'll need to use to complete this method's functionality. We'll need modules that can do the following:

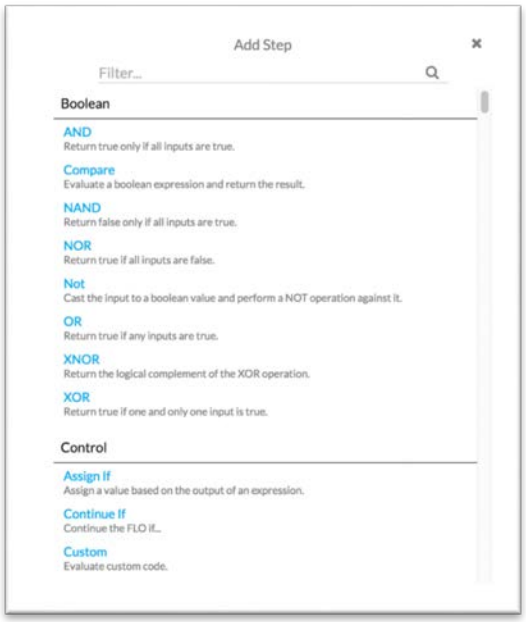
1. We'll need to grab the data from our API, with an HTTP "GET" operation
2. Then we'll need to take that data, and then shape it into a JSON object with the structure our method is expecting as output

**Note**: If, while building these modules, you come across this error while saving …

```
{
 "message": "action method must return an object",
 "type": "INVALID_CLOSE_TYPE",
 "channelMethod": "myMethodNameHere",
 "parent": []
}
```

… this means that your Action method needs to have its last output be a single object, as the engine needs to know how to tie the data to our current UI framework. So, if the last module in your method has more than one output (like most HTTP bricks) or has an output that is a collection, you will see this error pop-up. To fix it, just add in a module that ends with a single, non-collection output (like Object.Construct).

Inside of your method, under the subtitle "Core" click **+ Add Step**. This will bring up a menu of all the available modules:




Since we know we'll be interacting with an HTTP endpoint, go to the **HTTP** section. Here, we'll see a number of options, but we're only interested in doing a GET operation, so, click **HTTP Get**.

Once you do, it'll bring up a JSON template like this:

```
{
  "brick": "http.get",
  "id": "HJ6XE",
  "inputs": {
    "url": {
      "_availableTypes": [
        "string"
      ],
      "_type": "string",
      "_array": false,
      "_value": null
    },
    "filterEmpty": {
      "_availableTypes": [
        "boolean"
      ],
      "_type": "boolean",
      "_array": false,
      "_value": true
    },
    "ssl": {
      "cert": {
        "_availableTypes": [
          "string"
        ],
        "_type": "string",
        "_array": false,
        "_value": null
      },
      "key": {
        "_availableTypes": [
          "string"
        ],
        "_type": "string",
        "_array": false,
        "_value": null
      },
      "ca": {
        "_availableTypes": [
          "string"
        ],
        "_type": "string",
        "_array": false,
        "_value": null
      },
      "passphrase": {
        "_availableTypes": [
          "string"
        ],
        "_type": "string",
        "_array": false,
        "_value": null
      }
    },
    "query": {
      "_availableTypes": [
        "object",
        "string"
      ],
      "_type": "object",
      "_array": false,
```

```
      "_value": null
    },
    "headers": {
      "_type": "object",
      "_array": false,
      "_value": null
    }
  },
  "outputs": {
    "statusCode": {
      "_type": "number",
      "_array": false
    },
    "body": {
      "_type": "object",
      "_array": false
     }
   }
}
```

That blob is JSON definition for our HTTP.Get function. For now, we can get rid of all the items in the "inputs" section that we don't need, remembering that each of these items represents a parameter into its underlying function. For such a simple API, this means we can get rid of everything but "url," since that's the only thing we need to customize for this module. Documentation for each module can be found beneath the Modules category on the sidebar.

This pares us down to just this:

```
{
  "brick": "http.get",
  "id": "HJ6XE",
  "inputs": {
    "url": {
      "_availableTypes": [
        "string"
      ],
      "_type": "string",
      "_array": false,
      "_value": null
    }
  },
  "outputs": {
    "statusCode": {
      "_type": "number",
      "_array": false
    },
    "body": {
      "_type": "object",
      "_array": false
    }
  }
}
```

This means that the only data we have to pass in is the URL, and then we can get back the statusCode and body of that API's response.

So, let's fill in our "url" value. To do this, grab the general URL that we will need, "http://anapioficeandfire.com/api/characters/<characterID>", and stick it in the "_value" key for that input.

We're almost done! Now we just need to generalize the character ID to be based off of what the user enters into our method's card. We can access this using Mustache to reference our "input" object, which contains all values entered by the user. In this case, at {{input.Character Info.Character ID}}, due to the structure we defined for our inputs.

The resulting module:

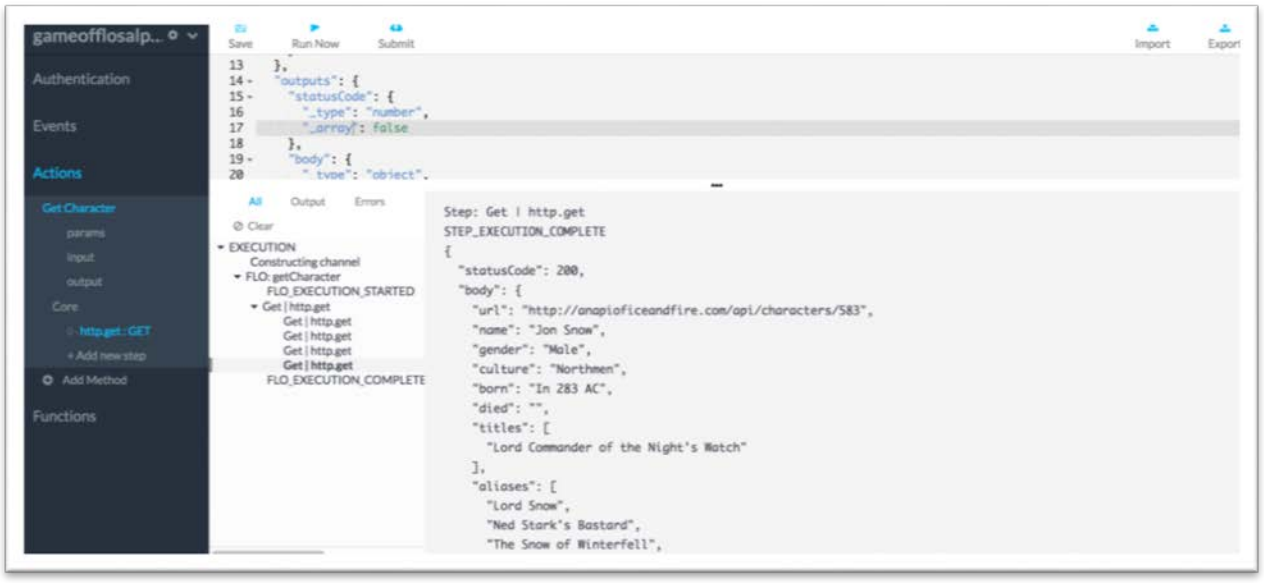
```
{
  "brick": "http.get",
  "id": "HJ6XE",
  "inputs": {
    "url": {
      "_availableTypes": [
        "string"
      ],
      "_type": "string",
      "_array": false,
      "_value": "http://anapioficeandfire.com/api/characters/{{input.Character
Info.Character ID}}"
    }
  },
  "outputs": {
    "statusCode": {
      "_type": "number",
      "_array": false
     },
    "body": {
      "_type": "object",
      "_array": false
    }
  }
}
```

Let's give this module a whirl to see what we can expect when this actually runs. In the toolbar, click **Run Now**. A dialog appears that allows you to simulate a single run of your connector's method.

In the "inputs" section, you'll see this:

```
{
  "Character Info": {
    "Character ID": ""
  }
}
```

This should look familiar, as this is the structure of the data we built in the UI section. The blank ("") value represents what would be passed into our method at runtime. Let's test with an actual character ID, like a real user would. Insert 583 (or any character ID of your choosing) into those quotes, and click **Submit**.




Now, you'll see that the bottom panel has filled in some logs for you about our HTTP.Get operation. You'll see four logs for each module when you run your method. The first two represent what the underlying function is taking in, and the second pair represent what is coming back out of that underlying function. As you can see, we get back a "statusCode" output, and then the service response in the "body" output. This is where all the data for our method can be found.

Here's the response log you should get (assuming you used 583 as your character ID):

```
{
  "statusCode": 200,
  "body": {
    "url": "http://anapioficeandfire.com/api/characters/583",
    "name": "Jon Snow",
    "gender": "Male",
    "culture": "Northmen",
    "born": "In 283 AC",
    "died": "",
    "titles": [
      "Lord Commander of the Night's Watch"
    ],
   "aliases": [
     "Lord Snow",
     "Ned Stark's Bastard",
     "The Snow of Winterfell",
     "The Crow-Come-Over",
     "The 998th Lord Commander of the Night's Watch",
```

```
    "The Bastard of Winterfell",
    "The Black Bastard of the Wall",
    "Lord Crow"
  ],
  "father": "",
  "mother": "",
  "spouse": "",
  "allegiances": [
    "http://anapioficeandfire.com/api/houses/362"
  ],
  "books": [
   "http://anapioficeandfire.com/api/books/5"
  ],
  "povBooks": [
    "http://anapioficeandfire.com/api/books/1",
    "http://anapioficeandfire.com/api/books/2",
    "http://anapioficeandfire.com/api/books/3",
    "http://anapioficeandfire.com/api/books/8"
  ],
  "tvSeries": [
    "Season 1",
    "Season 2",
    "Season 3",
    "Season 4",
    "Season 5",
    "Season 6"
  ],
  "playedBy": [
    "Kit Harington"
  ]
 }
}
```

As you can see, we'll want to use the following data points to generate this method's outputs:

- (Character Info)
- (Name)- body.name
- (Culture)- body.culture
- (Aliases) - body.aliases
- (TV Series Info)
- (Played By) - body.playedBy.0

Now we need to map this data to the UI of our method. To do this, we'll need to output a JSON object that matches the structure of our UI, with data filled in for our various fields. To do this, let's use the "Object.Construct" module. Add this underneath our HTTP.Get module.

It will generate a new JSON object for you, like this:

```
{
  "brick": "object.construct",
  "id": "H1Pzr",
  "inputs": {},
  "outputs": {
    "output": {
      "_type": "object",
      "_array": false
    }
  }
}
```

For this module, we want to create a JSON schema inside out "inputs" section, and our new JSON object will come out in that given structure. So, let's define the basic structure of our outputs in JSON schema format. Like so:

```
{
  "brick": "object.construct",
  "id": "H1Pzr",
  "inputs": {
    "Character Info" : {
      "Name" : "",
      "Culture" : " ",
      "Aliases" : ""
    },
    "TV Series Info" : {
      "Played By" : ""
    }
  },
  "outputs": {
    "output": {
      "_type": "object",
      "_array": false
    }
  }
}
```

Now we have the structure complete, but we still need to map the data to this structure. But the data is sourced from our **HTTP. Get** module! How do we access it?

You can access any outputs of any previous modules by using Mustache in this format: "{{ModuleIDHere.outputNameHere}}". To find your module ID, look at the top of the module for the "id" key. Its value is your module's ID. You may also change this value to whatever suits your needs.

For clarity's sake, let's navigate to our HTTP.Get module and change its ID, since we'll need to reference its outputs. I'll change mine from "B1wq-" to "GET". Like this:

**Original:**

```
"brick": "http.get",
"id": "B1wq-"...
```

**After:**

```
"brick": "http.get",
"id": "GET"...
```

Now that we've adjusted our method a bit, let's add those references to our final Object.Construct. Navigate back to the Object.Construct module, and add in those Mustache references in the proper spots. You should end up with this:

```
{
  "brick": "object.construct",
  "id": "H1Pzr",
  "inputs": {
    "Character Info": {
      "Name": "{{GET.body.name}}",
      "Culture": "{{GET.body.culture}}",
      "Aliases": "{{GET.body.aliases}}"
    },
    "TV Series Info": {
      "Played By": "{{GET.body.playedBy.0}}"
    }
  },
  "outputs": {
    "output": {
      "_type": "object",
      "_array": false
    }
  }
}
```

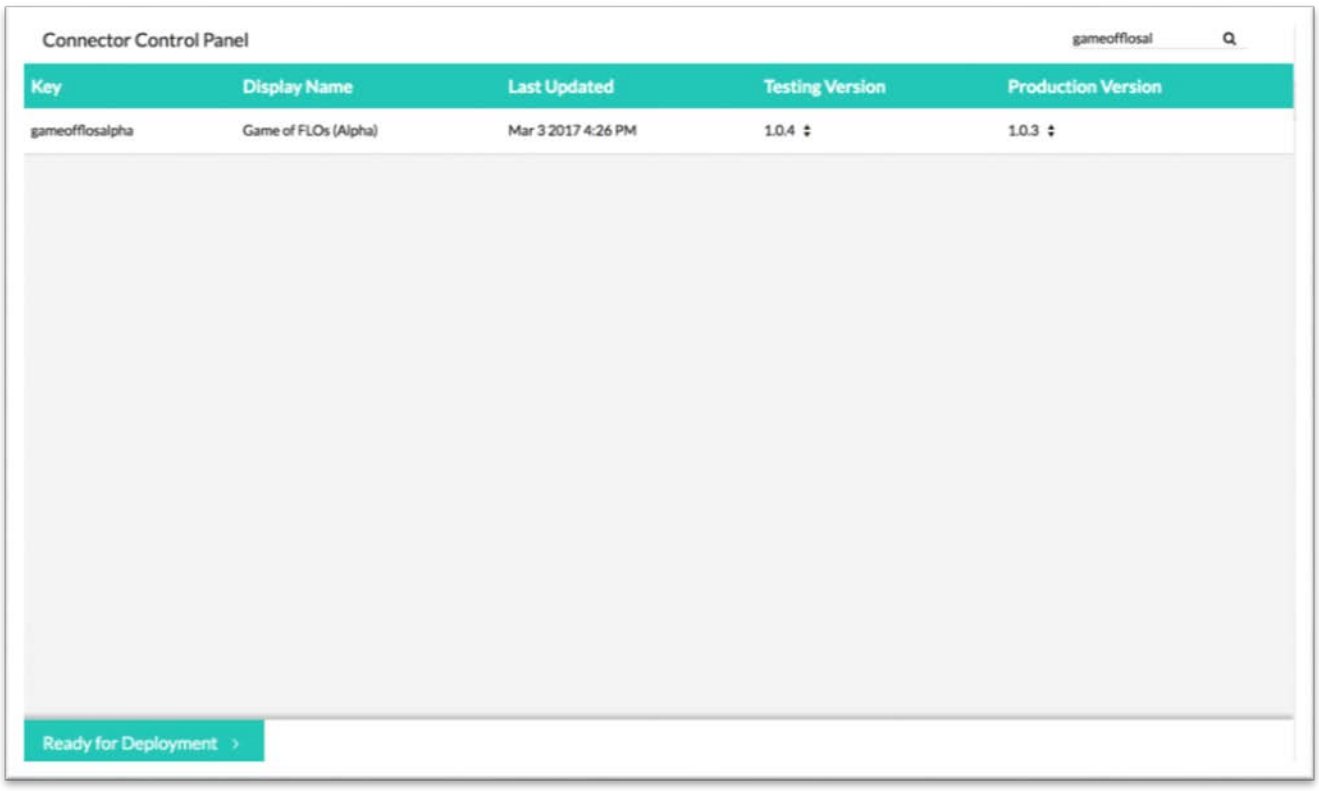
Give it a final test (with **Run Now**) to make sure the data makes it all the way through our modules. You should end up with this, if you use the character ID "583":

```
Step: Construct | object.construct
 STEP_EXECUTION_COMPLETE
 {
  "output": {
    "Character Info": {
      "Name": "Jon Snow",
      "Culture": "Northmen",
      "Aliases": [
        "Lord Snow",
        "Ned Stark's Bastard",
        "The Snow of Winterfell",
        "The Crow-Come-Over",
        "The 998th Lord Commander of the Night's Watch",
        "The Bastard of Winterfell",
        "The Black Bastard of the Wall",
        "Lord Crow"
      ]
    },
    "TV Series Info": {
      "Played By": "Kit Harington"
    }
  }
}
```

We're done! You just built a custom API integration into the Kenetix platform. Now let's see it in a FLO.

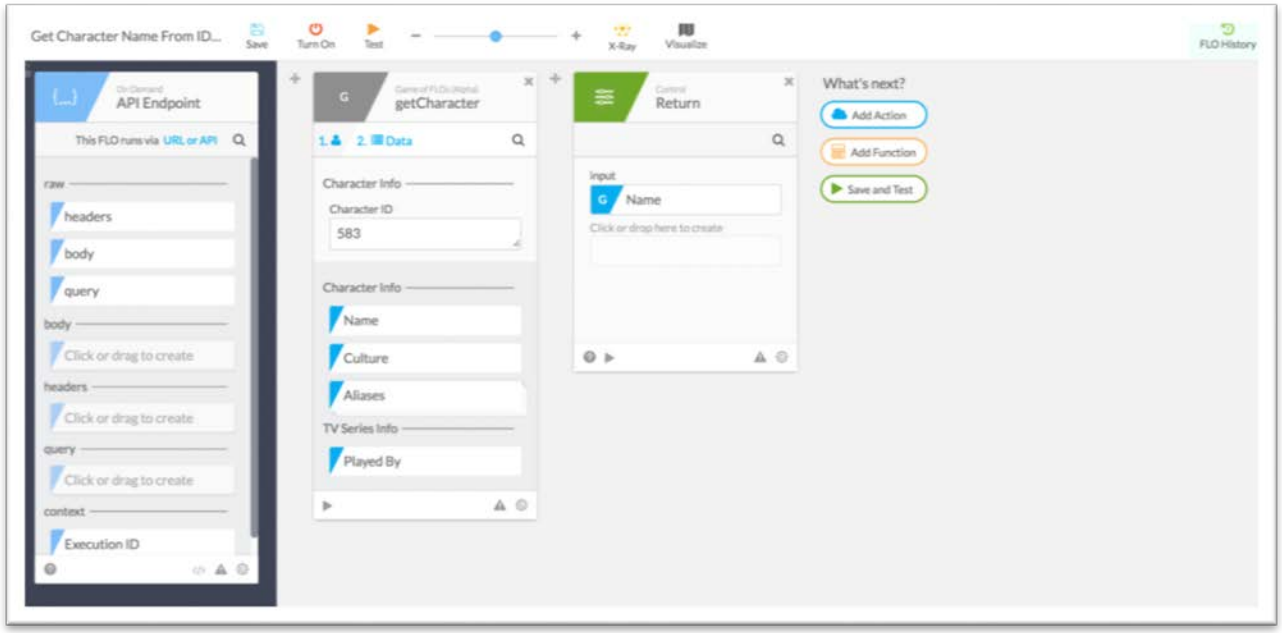
To get our connector into the core Kenetix platform, we first need to save and submit it. Click **Save** in the toolbar, then click **Submit**. This will prompt you to submit this connector file as a version of your connector. Please use proper semantic versioning. ***Incrementing the major version of a connector will break FLOs operating with your connector.***

Navigate to the Control Panel in the upper navigation bar where you should see a list of connectors that your organization has built. Here is where we deploy and manage the connector versions that are available in the main Kenetix platform.




You should see stuff about "Testing Version" and "Production Version" but for now we won't worry about this. Click "Testing Version" by your connector name, and select the version you just submitted from that dropdown. Once selected, click **deploy**, and wait for your connector to deploy to the Kenetix FLO building environment.

Navigate to the main platform (available via either URL or the "Designer" navigation menu item), and start building a FLO! Try adding your brand new connector as a step in a FLO.




Congrats! You've built a connector!

In the next guide, we'll be building off of this API further, and adding a new method that outputs collections of items. Thanks for following along!

## Building Your First Connector, Part 2

In the first part of this tutorial, we started building a connector that integrated with a Game of Thrones API, an API of Ice and Fire by Joaquim Skoog. In that previous walkthrough we went over:

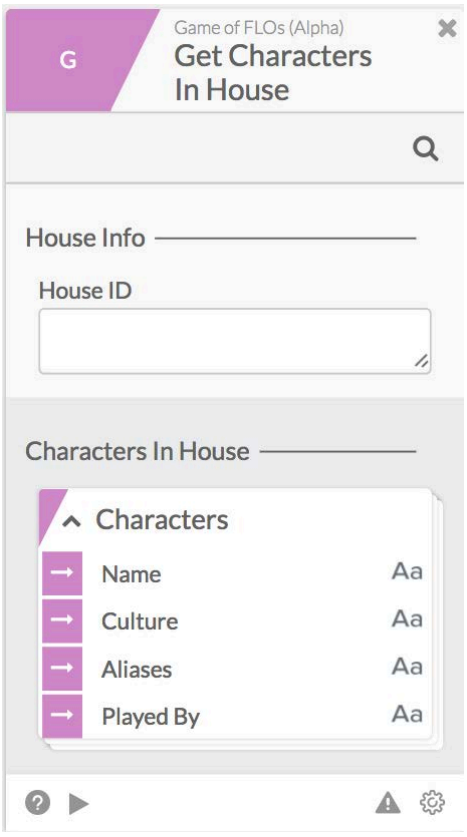
- Building a basic connector method's UI
- Creating actual functionality in your connector method, including module basics and more
- Testing connector methods
- Saving, submitting and deploying a connector

Specifically, we went through a tutorial about how to build a method in our connector that takes in a character ID and returns information about that character.

This time, we'll be building off of the connector we built earlier and adding integration with another route from the API of Ice and Fire. We'll now add these skills to our arsenal of connector building techniques:

- Returning a collection from a card
- Mapping over a list, to return a collection in the format we'd like
- Using helper functions in a connector
- Using "variant"

This route allows you to look up a Game of Thrones house, and get back some useful information, including a list of URLs for all characters in that house. We'll then process each one of those URLs, and return a collection of characters that belong to that house. We'll call it "Get Characters in House."




Just like before, we'll start by designing the way users will interact with this connector while building FLOs.

For this specific route, we'll only need to take in the house ID. Then, we'll return a collection of characters, with each character providing the following information:

- Name
- Culture
- Aliases
- Played By

The card we'll be aiming for will look like this in a FLO.

Now that we know what we're aiming for, let's start building this connector.

## UI

If you've gone through the previous walkthrough, defining your inputs should feel fairly familiar at this point. Let's give it the heading "House Info" and name the field we're asking users to input "House ID."

You should end up with something like this:

```
{
   "extensible": false,
   "attributes": [
      {
         "name": "House Info",
         "attributes": [
             {
                 "name": "House ID",
                 "type": "string"
             }
          ]
       }
    ]
 }
```

Now, making our outputs will be a little different this time around. The foundational concepts will still be the same, but this time we'll be returning a collection of objects in our card.

In general, collections inside the platform are represented in the following way:

```
Heading----
      Collection Name
           - Field inside Individual Item
```

In our case, it will be:

```
Characters In House-----
      Characters
           - Name
           - Etc.
```

So, first, let's apply the knowledge we learned from last time and fill in our heading, and a single object field inside of that called "Characters." "Characters" should also contain the fields "Name," "Culture," "Aliases," and "Played By." Remember to assign the type "object" when defining an object with fields inside of it.

We'll end up with this:

```
{
  "extensible": false,
  "attributes": [
    {
      "name": "Characters In House",
      "attributes": [
        {
          "name": "Characters",
          "type" : "object",
          "attributes" : [
            {
              "name" : "Name",
              "type" : "string"
            },
            {
              "name" : "Culture",
              "type" : "string"
            },
```

```
            {
              "name" : "Aliases",
              "type" : "string",
              "collection" : true
            },
            {
              "name" : "Played By",
              "type" : "string"
            }
          ]
        }
      ]
    }
  ]
}
```

Turning this into a collection is actually very simple. All we need to do is add a "collection" field to the "Characters" object and set it to true, like so:

```
{
   "name": "Characters",
   "type" : "object",
   "collection" : true,
   "attributes" : [{...}]
}
```

We're now done defining our UI! Now onto the functionality.

## Core

NOTE: If, while building this method, you come across this error while saving …

```
{
   "message": "action method must return an object",
   "type": "INVALID_CLOSE_TYPE",
   "channelMethod": "myMethodNameHere",
   "parent": []
}
```

… this means that your Action method needs to have its last output be a single object, as the engine needs to know how to tie the data to our current UI framework.

So, if the last module in your method has more than one output (like most HTTP bricks) or has an output that is a collection, you will see this error pop-up. To fix it, just add in a module that ends with a single, non-collection output (like Object.Construct).

In this method, we're going to have to perform a couple of steps.

1. Grab the house information with the given house ID
2. Grab the "swornMembers" list of URLs from the above response, and for each item in that list:
    a. Grab the character information from the URL
    b. Format the information into a way that matches the expected UI structure for each item in our list
3. Then, format the final output to include the now-formatted list

At the end of it all, we'll want to be outputting information like this:

```
{
   "Characters In House": {
      "Characters": [
         {
            "Name": "Daenerys Targaryen",
            "Culture": "Valyrian",
            "Aliases": [
               "Dany",
               "Daenerys Stormborn",
               "The Unburnt",
               "Mother of Dragons",
               "Mother",
               "Mhysa",
               "The Silver Queen",
               "Silver Lady",
               "Dragonmother",
               "The Dragon Queen",
               "The Mad King's daughter"
            ],
            "Played By": "Emilia Clarke"
         }
      ]
   }
}
```

To actually accomplish this, we need to get our modules set up.

The first module we'll want to set up is an HTTP.Get module, that will make a request to the "/houses" route. This should feel familiar to the work we did in part 1, so if we just jump right to what this module will end up looking like, we should end up with:

```
{
   "brick": "http.get",
   "id": "GET HOUSE",
   "inputs": {
      "url": {
         "_availableTypes": [
            "string"
         ],
         "_type": "string",
         "_array": false,
         "_value": "http://anapioficeandfire.com/api/houses/{{input.House
Info.House ID}}"
      }
   },
   "outputs": {
      "statusCode": {
         "_type": "number",
         "_array": false
      },
      "body": {
          "_type": "object",
          "_array": false
      }
   }
}
```

After this, we will need to perform some operations on the list of URLs coming back from the API's response. After that we'll return that modified list to our final output. The best module for this is List.Map. This module takes in a list, performs a process on each item in that list, and replaces the original item in that list with the processed item.

On the module menu, click List.Map; this brick will be generated for you:

```
{
   "brick": "list.map",
   "id": "PROCESS ITEM",
   "item": "",
   "inputs": {
      "list": {
         "_availableTypes": [
             "*"
         ],
         "_type": "object",
         "_array": true,
         "_value": ""
      },
      "flo": {
         "_type": "flo",
         "_array": false,
         "_value": ""
      },
      "concurrency": {
         "_type": "number",
         "_array": false,
         "_value": 1
      }
   },
   "outputs": {
      "new list": {
         "_type": "object",
         "_array": true
      }
   }
}
```

Just like with the HTTP.Get module, each item in the "inputs" object in this module represents a way we can affect how this mapping operation runs. Let's fill in these values:

- "list" - this is the list we want to iterate over. As you can see, the default is a list of objects, but we're actually going to be iterating over a list of URLs- which are strings. Make sure to change the "_type" field on this input to "string". To set the value, use a Mustache reference on the "swornMembers" list from our HTTP.Get, like so: "{{httpModuleIDHere.body.swornMembers}}".
- "concurrency" - The number of items I would like to process at once. For our purposes, "1" is fine for now.

This leaves "flo." This input is the function, or sub-FLO, that we will use on every item in our list. The output of this FLO will replace the item it operated on. For example, if my original list was a collection of strings, and my sub-FLO returns an object, I will end up with a list of objects as my final output of this List.Map module.

To set this value, we'll first need to create a function/sub-FLO to run against. On the left menu, click **Functions**. Click **Add Method**, just like when we were adding an Action before. Go ahead and name this function; I will be naming mine "Get Character For List," since we will be grabbing specifics about each character hosted at a URL from our original list of URLs.

Let's fill in the "flo" input in our List.Map now that we have a method to put in there. Note, when you create a function, you will be creating it with a display name. However, this name is not the "identifying name," which we will need to use to set our "flo" input value. The identifying name will be the camel-cased, condensed name of your function. So, in this case, the identifying name would be "getCharacterForList," and that's what I would put as the "_value" for the "flo" input in this List.Map.

Now, we have our List.Map filled out. It should look like this:

```
{
   "brick": "list.map",
   "id": "PROCESS ITEM",
   "item": "",
   "inputs": {
      "list": {
         "_availableTypes": [
            "*"
         ],
         "_type": "string",
         "_array": true,
         "_value": "{{GET HOUSE.body.swornMembers}}"
      },
      "flo": {
         "_type": "flo",
         "_array": false,
         "_value": "getCharacterForList"
      },
      "concurrency": {
         "_type": "number",
         "_array": false,
         "_value": 1
      }
   },
   "outputs": {
      "new list": {
         "_type": "object",
         "_array": true
      }
   }
}
```

Now, we need to actually build out our helper function. This function will need to access the item in the list we're processing, grab the character information from that URL, and then output that new information.

To access the individual item in the list we're processing, we will need to add a "variant" flag to our helper method to signal to the engine that this helper function is intended to process a list. "Variant" flags live at the top level of a helper method and are set by clicking the name of your helper method, and then adding this right above "zebricks":


Now, we can refer to each individual item in our list by using "{{item}}" throughout this helper function.

So, in our helper function, we can now refer to the individual item, but now we need to create its actual functionality. First, we'll want to add an HTTP.Get module so we can call the URL we have from our list. Inside of that HTTP.Get module, we'll want to fill in the "url" value with "{{item}}", since each item in our parent list is an actual URL.

That will give us this:

```
{
   "brick": "http.get",
   "id": "GET CHARACTER",
   "inputs": {
      "url": {
         "_availableTypes": [
             "string"
         ],
         "_type": "string",
         "_array": false,
         "_value": "{{item}}"
      }
   },
   "outputs": {
       "statusCode": {
           "_type": "number",
           "_array": false
       },
       "body": {
           "_type": "object",
           "_array": false
       }
   }
}
```

Finally, we need to shape this data to match our desired output for each item in our list. This means that it needs to be shaped like this:

```
{
   "Name": "Daenerys Targaryen",
   "Culture": "Valyrian",
   "Aliases": [
      "Dany",
      "Daenerys Stormborn",
      "The Unburnt",
      "Mother of Dragons",
      "Mother",
      "Mhysa",
      "The Silver Queen",
      "Silver Lady",
      "Dragonmother",
      "The Dragon Queen",
      "The Mad King's daughter"
    ],
    "Played By": "Emilia Clarke"
}
```

So, just like in part 1, our Object.Construct will need to look like this:

```
{
   "brick": "object.construct",
   "id": "rkbVC",
   "inputs": {
      "Name": "{{GET CHARACTER.body.name}}",
      "Culture": "{{GET CHARACTER.body.culture}}",
      "Aliases": "{{GET CHARACTER.body.aliases}}",
      "Played By": "{{GET CHARACTER.body.playedBy.0}}"
   },
   "outputs": {
      "output": {
          "_type": "object",
          "_array": false
      }
   }
}
```

Our helper function is now done! Let's add the finishing touches to our "parent method" (Get Characters In House). All we need to do now is map this data into our UI's format. This means adding one last Object.Construct underneath our List.Map module. The structure of the object we're creating is going to be familiar, except this time we've already formatted the entire collection. This means that we fill in the heading and "Characters" field just like before, but we use a Mustache reference for the "Characters" value.

If we named our List.Map module "PROCESS ITEM," we would end up with this:

```
{
   "brick": "object.construct",
   "id": "FORMAT",
   "inputs": {
       "Characters In House": {
          "Characters": "{{PROCESS ITEM.new list}}"
       }
    },
    "outputs": {
        "output": {
            "_type": "object",
            "_array": false
        }
    }
}
```

And ta-da! We're done. List.Map is a crucial piece of working with connectors, and once you've understood one way to use it, the rest will come easily. Nicely done!