KENETIX™

CONNECTOR DEVELOPMENT GUIDE

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

Authentication

In the platform Designer, users can connect an account to a Connector, set up their FLO, and then let it run. The platform handles all the hard work of authentication, and securely stores any pertinent authentication information (like credentials, API keys, or access keys obtained from an OAuth exchange) so it can be used whenever the FLO runs.

The platform supports 4 kinds of authentication: Basic, OAuth 1.0, OAuth 2.0, and Custom (often used for Key/Secret). In Custom authentication, you also may choose to define an authentication object that takes in custom parameters that you can then send over as part of your request to the API.

To set up authentication, go the Authentication section on the sidebar, and select the authentication type your API requires. Each form has slight variations, but all authentication types come in the basic form:

```
{
"type": ""
"authparams" : {
"parameterName": {
"type": "",
"displayname" : ""
}
}
}
```

Fields:

- "type": The type of authentication (will be automatically populated)
- "authparams": The set of parameters you would like to show up in the authentication visual elements of your Connector.
- "parameterName": An individual authentication parameter that will be on your Connector. This name will not be surfaced onto the Connector itself, but will be the key you use to refer to the value throughout your methods. You may have as many of these as you need.
- "type": The type of this parameter. Can be string, or password (hides the inputted value on the UI)
- "displayName": The name that will be shown to the user on your Connector in the platform's Designer.

Although the platform will handle the storage of authentication information, many APIs require you to use this authentication information to sign requests. To do this, all authentication parameters can be referred to in other methods using Mustache, in the form: "{{auth.parameterName}}".

Setting Up Basic Authentication

Use basic authentication when all you need to grant access to the API is a username and password, sent over in the headers or body of the request.

Template:

```
{
"type": "basic",
"authparams": {
"username": {
"type": "string",
"displayname": "Username"
},
"password": {
"type": "password",
"displayname": "Password"
}
}
}
```

Fields:

- "type": must be set to 'basic' for basic authentication
- "authparams": an object that contains the authentication parameters users will enter. For Basic auth, these are usually "username" and "password"
- "type": can be 'string' or 'password.' If type is 'password,' the input will be hidden as the user types it in
- "displayname": the name that will display above the authentication parameter in the UI

Example:

```
{
"type": "basic",
"authparams": {
"username": {
"type": "string",
"displayname": "Username"
},
"password": {
"type": "password",
"displayname": "Password"
},
"instance_url": {
"type": "string",
"displayname": "Instance URL (w/o https)"
}
}
}
```

Later on, you can reference these parameters using the Mustache templates {{auth.username}} and {{auth.password}}.

Setting Up OAuth 1.0

Template:

```
{
"type": "oauth",
"version": "1.0",
"request_url": "",
"access_url": "",
"consumer_key": "",
"consumer_secret": "",
"callback": "{server}/app/oauth/(YOUR APP)/authorize",
"signature_method": "HMAC-SHA1",
"nonce_size": null,
"redirect": ""
}
```

Fields:

- "type": must be set to 'oauth'
- "version": must be set to 1.0 for OAuth 1
- "request_url": the external URL where the token request will be made
- "access_url": the external URL where the request token will be exchanged for the access token
- "consumer_key": the access key obtained from the application
- "consumer_secret": the access secret obtained from the application
- "callback": the callback URL. Substitute the Connector's unique filename (the one you created when you first saved your Connector) for "(YOUR APP)"
- "signature_method": must be 'HMAC-SHA1'
- "nonce_size": must be null
- "redirect": the redirect URL (may also be called the authorize URL).

Optional Fields:

- "authparams": an object that contains the authentication parameters users will enter. Only include authparams in the OAuth object if you require users to enter additional data before you can run the OAuth exchange, such as an instance URL. Each parameter defined in this object can be referenced as {{auth.(parameter key)}} using Mustache.
- "type": can be 'string' or 'password.' If type is 'password,' the input will be hidden as the user types it in
- "displayname": the name that will display above the authentication parameter in the UI

The access token that results from the OAuth exchange can be referenced from inside each method and hashed into HTTP calls using the Mustache template {{auth.access_token}}. After you have set up your authentication, all requests to the API will be automatically signed for you according to OAuth 1.0 protocol.

OAuth 2.0 Redirect URL

Some applications require you to set acceptable callback or redirect URLs when you register your application. You must add this redirect URL to your application to be able to run OAuth 1.0 from the Connector Builder, Beta Environment, and Production Environment:

https://api.the platform.com:443/app/oauth/(FILENAME)/callback

Substitute the Connector's unique filename (the one you created when you first saved your Connector) for "FILENAME". Note, if you are developing in the platform's Beta environment, please replace "api" with "betaapi" in the above URL.

Setting Up OAuth 2.0

For information about OAuth 2.0 types, please check out this article. We currently allow integration with one part, two part and refresh-able OAuth 2.0 exchanges. We default our template for Authorization Code grants (a two part OAuth exchange, with a refresh option).

Template:

```
{
 "type": "oauth",
 "version": "2.0",
 "base_site": "",
 "authorize_path": "",
 "access_token_path": "",
 "access_token_name": "access_token",
 "refresh_token_name": "refresh_token",
 "client_id": "",
 "client_secret": "",
 "appParams": {
  "one": {
   "response_type" : "code"
   "state": true,
   "redirect_uri": "{server}/app/oauth/(YOUR APP)/authorize"
  },
  "two": {
   "grant_type" : "authorization_code"
   "redirect_uri": "{server}/app/oauth/(YOUR APP)/authorize"
  },
  "refresh": {
   "grant_type" : "refresh_token"
  }
 }
}
```

Fields:

- "type": must be set to 'oauth' for OAuth 2
- "version": must be set to 2.0 for OAuth 2
- "base_site": the base URL (may be omitted if the authorize_path and access_token path do not come from the same base site)
- "authorize_path": the URL (in addition to base_site) where users will be directed to give authorization
- "access_token_path": the URL (in addition to base_site) where the access token will be obtained
- "access_token_name": the key of the access token in the response
- "refresh_token_name": the key of the refresh token in the response
- "client_id": the client ID obtained from the application.Automatically appended to all steps of your authentication.
- "client_secret": the client secret obtained from the application. Automatically appended to all steps of your authentication.
- "appParams": an object representing the parameters you'll be sending in each step of your OAuth exchange.
- "one": an object that contains the query parameters required for the first exchange of your authentication process. The parameters included in the template are common parameters, but not all applications use the same fields.
- "state": a boolean that if true will include an unguessable state value in the OAuth exchange
- "redirect_uri": the URL where users will be sent after authorization. Substitute the Connector's unique filename (the one you created when you first saved your Connector) for "FILENAME", but leave "{server}"- as our engine will make that swap for you at runtime.
- "response_type": This parameter is specific to Authorization Code Grant OAuth 2.0
- "two": an object that contains the parameters required for the second step of your authentication except for "code", which is automatically passed to the second leg of the authentication exchange. The parameters included in the template are common parameters, but not all applications use the same fields.
- "redirect_uri": the URL where users will be sent after authorization. Substitute the Connector's unique filename (the one you created when you first saved your Connector) for "FILENAME"
- "grant_type" : This parameter is specific to Authorization Code Grant type of OAuth 2.0
- "refresh" (optional), an object that contains the parameters required to exchange a refresh token for a new auth_token if the token expires

Optional Fields:

- "authparams": an object that contains the authentication parameters users will enter. Only include authparams in the OAuth object if you require users to enter additional data before you can run the OAuth exchange, such as an instance URL. Each parameter defined in this object can be referenced as {{auth.parameterName}} using Mustache throughout the rest of your connector JSON.
- "type": can be 'string', 'password', or 'option'. If type is 'password,' the input will be hidden as the user types it in. If the type is 'option', you will need to provide an array of objects called "choices", where each item contains an "optionValue" and "displayName" field.
- "displayname": the name that will display above the authentication parameter in the UI
- The access token and refresh token that results from the OAuth exchange can be referenced from inside each method and hashed into HTTP calls using the Mustache reference {{auth.access_token}} or {{auth.refresh_token}}.

OAuth 2.0 Redirect URL

Some applications require you to set acceptable callback or redirect URLs when you register your application. You must add these redirect URLs to your application to be able to run OAuth 2.0 from the connector builder and from the production environment:

https://api.the platform.com:443/app/oauth/(FILENAME)/authorize

Substitute the Connector's unique filename (the one you created when you first saved your Connector) for "FILENAME". Note, if you are developing in the platform's Beta environment, make sure that you use "betaapi" instead of "api" in the above URL.

Setting Up Custom Authentication

Custom authentication allows you to define your own authentication parameters. Often, custom auth is used for key/secret authentication.

Template:

```
{
"type": "custom",
"authparams": {
"": {
"type": "string",
"displayname": ""
}
}
}
```

Fields:

- "type": must be set to "custom" for custom authentication
- "authparams": an object that contains the authentication parameters users will enter. For custom auth, you can define any authparams as you like

- "type": can be 'string' or 'password.' If type is 'password,' the input will be hidden as the user types it in
- "displayname": the name that will display above the authentication parameter in the UI

Example:

```
{
"type": "custom",
"authparams": {
"apikey": {
"type": "string",
"displayname": "Username"
},
"secret": {
"type": "password",
"displayname": "Password"
}
}
}
```

No matter how you define your authentication schema, you'll be able to reference your authentication parameters later on using Mustache in the form: "{{auth.parameterName}}"

Events and Actions

Once you've set up Authentication, you can start adding Events and Actions to your Connector. Each Event and Action your Connector supports represents a method that the platform user can use to interact with your API. Events are methods that monitor for a change in a system and are responsible for starting FLOs. Actions are methods that will run when the Event it's attached to is triggered, creating the actual functionality of a FLO.

All Events and Actions are composed of two parts:

- The schema that controls the parts of your Connector that users can see and manipulate on your card (params, input, and output)
- The underlying code that actually makes calls to the API and handles the response (modules in the Core section of your Connector).

Both the UI and functional parts of Events and Actions are supported by "Functions" which act as helper methods.

Actions

Actions have 4 sections:

- **Params**, where you declare any parameters that will appear on the card. Parameters do not accept any data from previous cards
- **Input**, where you declare any inputs that will appear on the card. Inputs can accept data that is dragged and dropped from an earlier card.
- **Output**, where you declare any outputs that will appear on the card
- **Core**, where you declare the steps (called Modules) that will make the call to the API to perform the Action and handle any data that the API returns.

In general, Actions take in data, use it to run a call to the API, then shape the resulting object into the declared output format.

Note: Actions do not return collections, just single objects. If the last module in your Action outputs a collections object (where "_array" = "true"), you will get a typecasting error.

Events

Events come in several different types:

- **Polling**: These triggers poll against your target service, and compare responses to identify changed states in your target resource. When it can tell there's been a change, it will trigger the rest of your FLO.
- **Webhook**: These triggers are initiated by services that support webhook frameworks. The platform will set up a webhook connection with the service programmatic. After, it will listen to broadcasts from your service, and then trigger the rest of its FLO once the event has been triggered
- **Invokable**: These triggers are initiated by external services that simply need an endpoint to trigger, like Salesforce Outbound Messages. Functionally they run similarly to webhooks, but require the webhook to be set up by the owner of an account in your target service.

Polling Events

Polling Events are one of the two types of Events that your Connector can contain. Polling Events are designed to automatically run on a recurring basis based on the user's plan (typically 1-5 minutes). This means that these Events will need to grab a collection of records, then process collections of records and determine which records have happened since its last iteration.

To be able to keep track of the last record that was collected, Polling Events use a tool called the "since" value. This value allows your API to understand when the last query sent from your Connector was, and can take many forms, although it often is a timestamp sent in your HTTP request. When the user turns on a FLO, the platform will call the API to get the current since value. Then, at a specified interval, the Event will call the API again and use the since value saved during the previous run to fetch new records. The engine will automatically execute the FLO once for every new record.

The Since Value

The "since" value is a single piece of data that persists between executions of an Event. This means that the data that was set on a previous run of an Event can be compared against a current run's dataset. Usually the "since" value is a timestamp or the ID of the last record collected of a previous iteration, that you can use to compare against the records collected on this iteration.

For example, if a service returns a list of all records, and each record looks like so…

```
{
 "recordName" : "A",
 "createdAt" : "2017-02-15 8:58:00"
}
```

… then, so long as I can keep track of the time the last iteration occurred, I can filter out my list of records to only include those whose "createdAt" value is greater than that timestamp. The "since" value would keep track of that timestamp to filter by, and then would be reset to the current time after the list was filtered.

The "since" value can be defined in the Control.Since module, and once defined may be referred to with Mustache as "{{since}}" throughout that Event method.

Building a Polling Event

Polling Events have a fairly well-defined structure that can be used in most Connectors. Generally speaking, they'll always include these steps, in approximately this order:

Get a List of Items from an API

Filter that list based off of the since value that was set on a previous execution. As a side note, sometimes services will allow you to provide a timestamp as a query parameter in the initial request. In those cases, this step is unnecessary, and instead you would pass the since value as that query parameter in the initial API interaction.

Update your "since" value with the Control.Since module

(Optional, but often needed) Map over each item in the filtered list to render that item into the right format

End the method with the new list of formatted and filtered items.

Note: A Polling Event must return a list, even if that list contains just one item in a given iteration

Testing a Polling Event

To test a Polling Event in the connector builder, you can artificially set the "since" value in the "Run" window, inside the "Since" text area. Inside of this area, put in any value that you want to simulate as the since value from the previous iteration of a method execution. You can also leave it as "null" to simulate the first run of your Polling Event.

Webhook Events

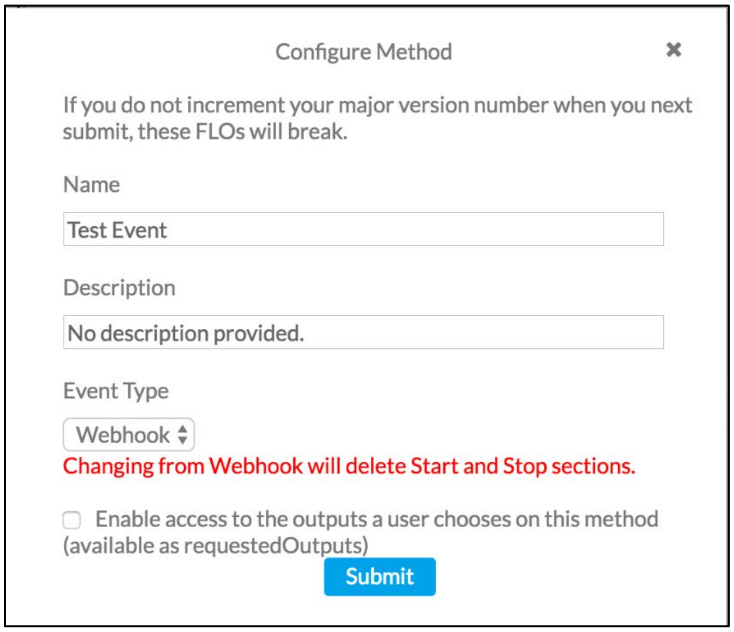
General

The platform also allows you to leverage your service's webhook functionality to build connector events that are triggered by your webhooks. Webhook Events will only run when the corresponding webhook is triggered, as opposed to Polling Events, which will run a check at a given time interval.

When the user turns on a FLO that starts with a Webhook Event, the platform will call an API endpoint to set up a webhook with your service. From then on, whenever the webhook is activated, the event will run and trigger the rest of that user's FLO. When the FLO is turned off, the platform will then teardown your webhook connection.

To declare an event as a webhook event, click on the ellipses by the event name, click **Configure Method**, and then under Event Type, click **Webhook**.

Once you do, you'll see three sections pop up underneath your Event's name: Start, Core, and Stop.


Configure Method

If you do not increment your major version number when you next submit, these FLOs will break.

Name

Test Event

Description

No description provided.

Event Type

Webhook

Changing from Webhook will delete Start and Stop sections.

Enable access to the outputs a user chooses on this method (available as requestedOutputs)

- **Start** is the operation that will run when the platform's engines programmatically set up a webhook with your service.
- **Core** is the operation that will run whenever the webhook is triggered.
- **Stop** is the operation that will run when the platform's engines programmatically tear down a webhook to your service.

Fundamentally, each of these sections are still just a series of modules, the same as **Core** in a Polling Event or in an Action.

Start

The Start section of your Webhook Event is run by the platform whenever a user turns on a FLO that uses your Webhook Event as its trigger. The Start section is responsible for hitting the API endpoint of your service responsible for creating a new webhook connection.

Often times you'll need information from inside your service's response from webhook setup in the rest of your Webhook Event. The platform allows you to access this through "{{input.blob.fieldName}}". Input.blob is generated from the final output of your Start section.

For example, let's say your service requires a webhook ID to tear down the webhook connection, and your service returns the webhook ID in the response under the field name "id." You would need to make sure your Start section outputs that response. Then you can access this value in your Stop section by using "{{input.blob.id}}". Additionally, you can also access the URL that the webhook will be set up on by using "{{input.url}}".

Core

The Core section of a Webhook Event is run by the platform whenever your webhook is triggered. This is where the main piece of functionality of your Event will be defined, and is often the most involved aspect of your Event.

The functionality of this section will depend on your service and what the goal of your event is. However, the final output of this section must always be a single object that matches the format of your "output" UI - just like an Action.

To reference data coming in from your webhook when your event is triggered, use "{{input.data.fieldName}}".

Stop

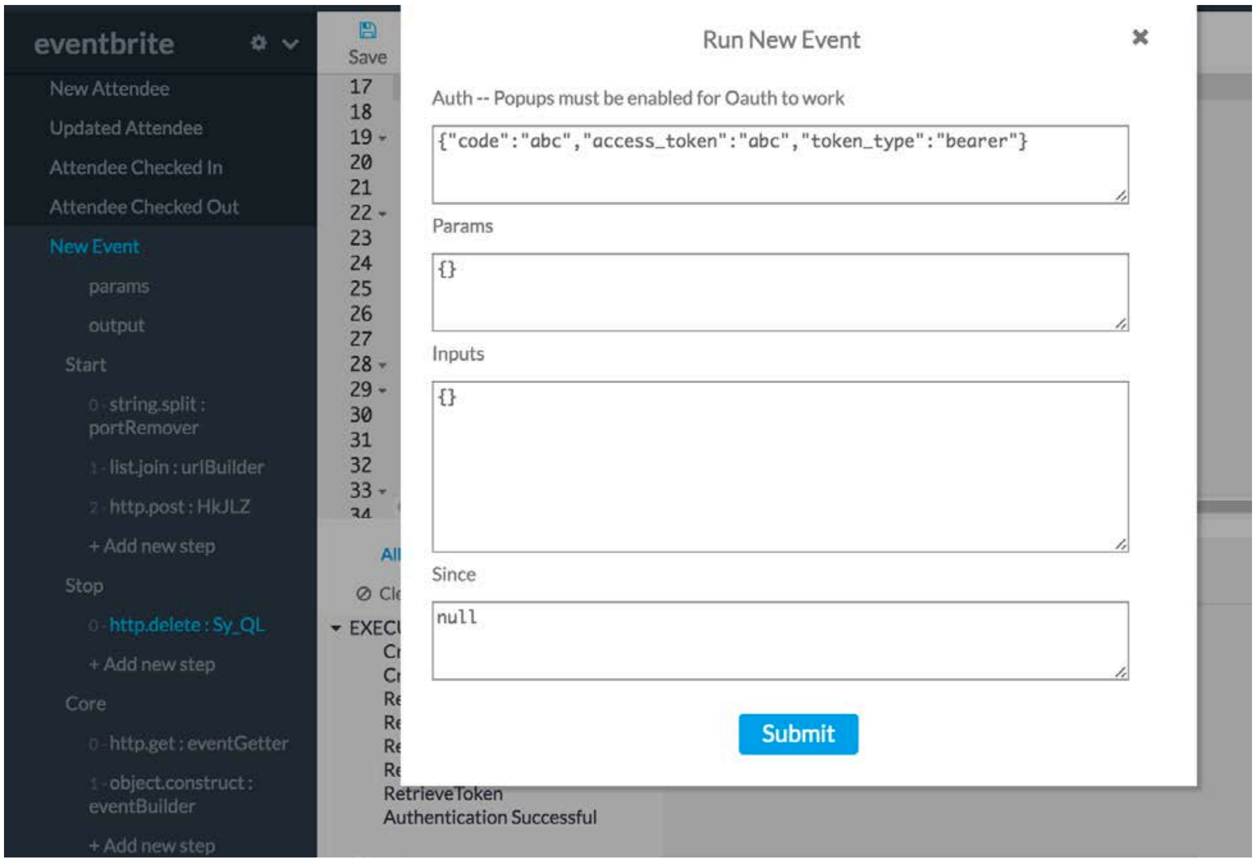
The Stop section of a webhook is run by the platform whenever a user removes your webhook event from a FLO, or turns off/deletes a FLO using your Event.

The Stop section is responsible for interacting with the API endpoint for your service that shuts off a webhook connection. To refer to any values that were passed from the service when a webhook connection was created, use "{{input.blob.fieldName}}"

Running Webhook Tests

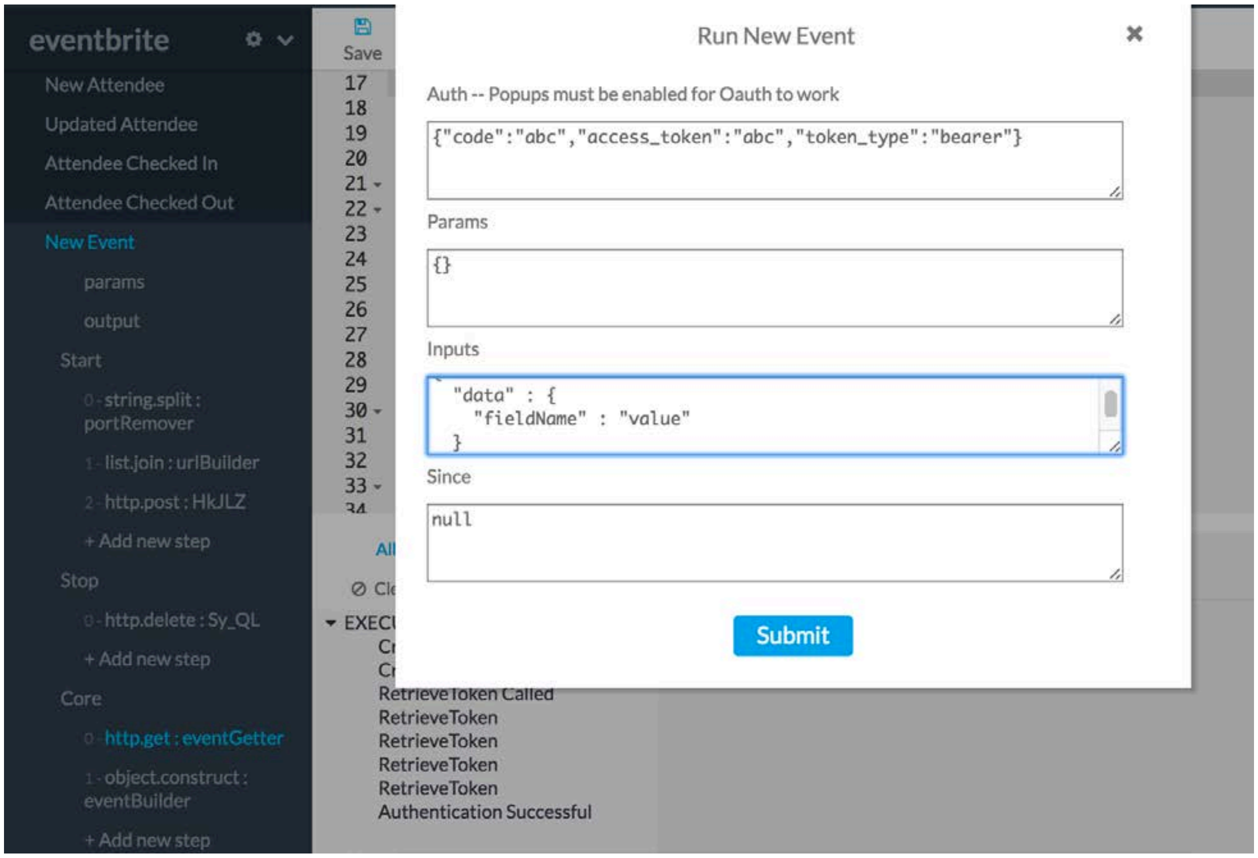
Webhook tests come in two varieties, running each segment on their own, and fully setting up a webhook and "listening" for webhook triggers.

The first type is fundamentally very similar to running an Action. To run this style, select any module within the Start/Stop/Core section you would like to run, and then click **Run**.



Auth:
```
{"code":"abc","access_token":"abc","token_type":"bearer"}
```
Params:
```
{}
```
Inputs:
```
{}
```
Since:
```
null
```

While testing Core, put any data you would like to test inside a "data" object in the "input" section, like so:



Auth -- Popups must be enabled for Oauth to work

```
{"code":"abc","access_token":"abc","token_type":"bearer"}
```

Params

```
{}
```

Inputs

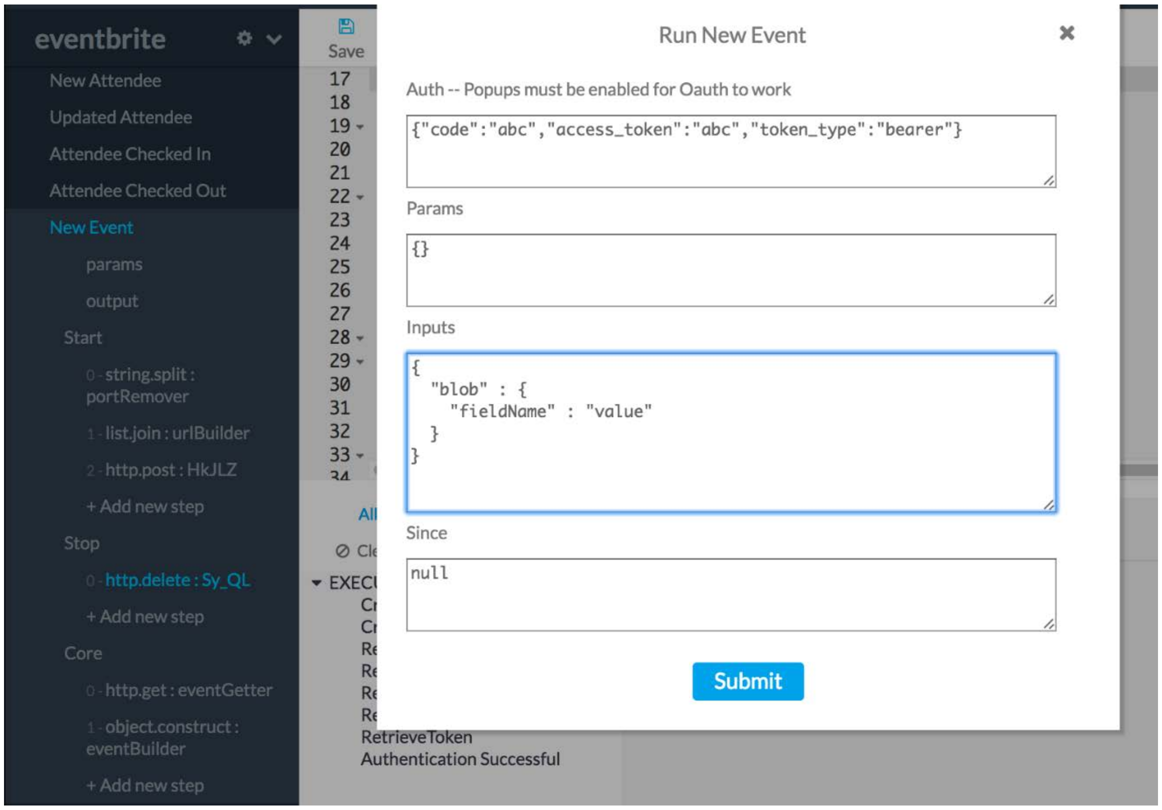
```
  "data" : {
    "fieldName" : "value"
  }
```

Since

```
null
```

```
Retrieve Token Called
RetrieveToken
RetrieveToken
RetrieveToken
RetrieveToken
Authentication Successful
```

While testing Stop, put any information you need inside of a "blob" object in the "input" section, like so:



Auth:

```
{"code":"abc","access_token":"abc","token_type":"bearer"}
```

Params:

```
{}
```

Inputs:

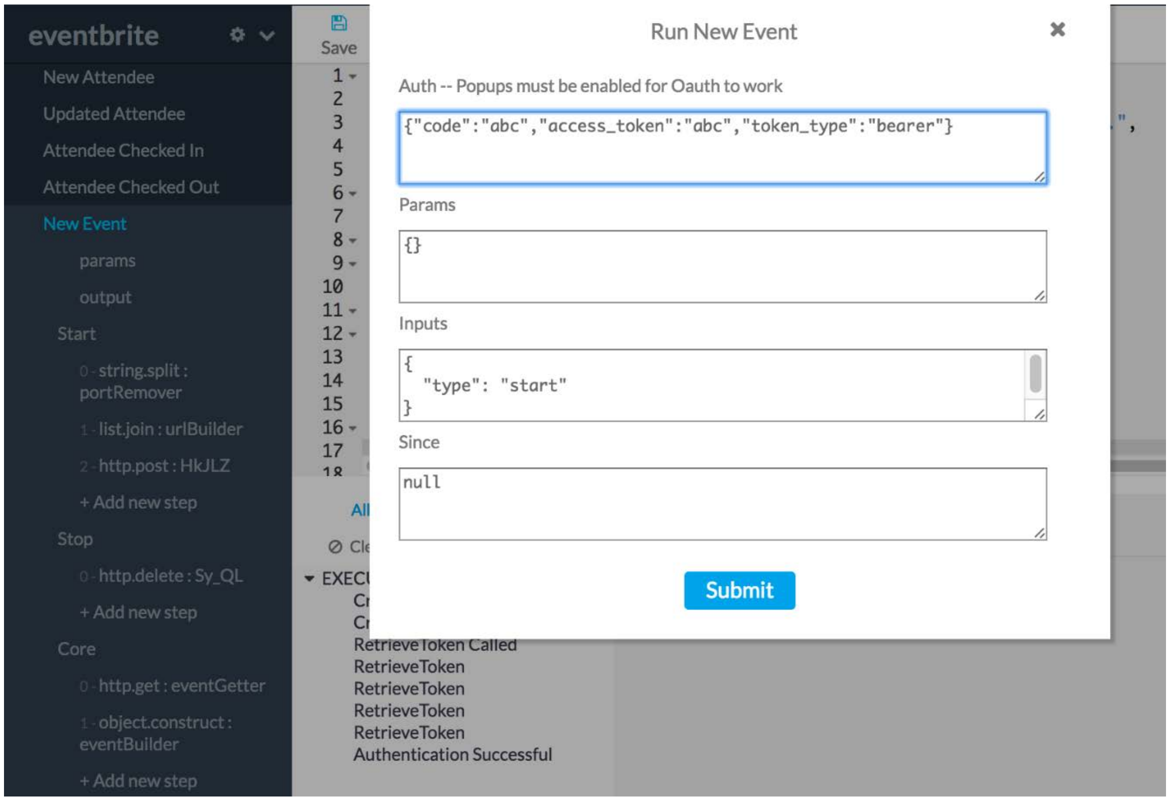
```
{
  "blob" : {
    "fieldName" : "value"
  }
}
```

Since:

```
null
```

The second type of testing fully sets up a webhook and listens for updates. This type will set up a webhook for you, display logs in the log window for each time your "Core" section runs, and gives you an option to close down your webhooks.

To use this type of test, click the name of your webhook event, and click **Run** without changing/passing in any inputs.



```
{"code":"abc","access_token":"abc","token_type":"bearer"}
```

```
{}
```

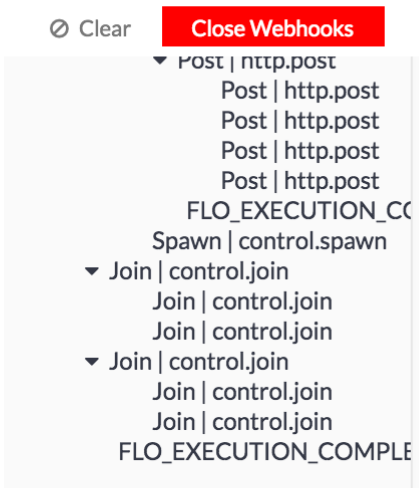
```
{
  "type": "start"
}
```

```
null
```

```
RetrieveToken Called
RetrieveToken
RetrieveToken
RetrieveToken
RetrieveToken
Authentication Successful
```

Then, inside of your service, perform whatever actions causes your webhook to trigger. You should see logs start coming in, processing the brand new webhook trigger event.

To close your webhook, you can click this button, and see logs for the teardown process.



```
    Post | http.post
        Post | http.post
        Post | http.post
        Post | http.post
        Post | http.post
      FLO_EXECUTION_CC
    Spawn | control.spawn
  Join | control.join
    Join | control.join
    Join | control.join
  Join | control.join
    Join | control.join
    Join | control.join
  FLO_EXECUTION_COMPLE
```

Example

Here's an example from the Eventbrite connector.

Start:

```
{
 "brick": "string.split",
 "id": "portRemover",
 "inputs": {
  "string": {
   "_type": "string",
   "_array": false,
   "_value": "{{input.url}}"
   },
   "delimiter": {
    "_type": "string",
    "_array": false,
    "_value": ":443"
   }
 },
 "outputs": {
  "output": {
   "_type": "string",
   "_array": true
  }
 }
},
{
 "brick": "list.join",
 "id": "urlBuilder",
 "inputs": {
  "list": {
   "_availableTypes": [
    "number",
    "string",
    "boolean",
    "Date"
   ],
   "_type": "string",
   "_array": true,
   "_value": "{{portRemover.output}}"
  },
  "delimiter": {
   "_type": "string",
   "_array": false,
   "_value": ""
  }
 },
 "outputs": {
  "string": {
   "_type": "string",
   "_array": false
  }
 }
},
{
 "brick": "http.post",
 "id": "makeWebhook",
 "inputs": {
  "url": {
   "_availableTypes": [
    "string"
   ],
   "_type": "string",
   "_array": false,
   "_value":
"https://www.eventbriteapi.com/v3/webhooks/?token={{auth.access_token}}"
  },
  "body": {
   "_availableTypes": [
    "object",
    "string"
   ],
   "_type": "object",
   "_array": false,
   "_value": {
    "actions": "event.created",
    "endpoint_url": "{{urlBuilder.string}}"
   }
  },
  "headers": {
   "_type": "object",
   "_array": false,
   "_value": {
    "Authorization": "Bearer {{auth.access_token}}",
    "Content-Type": "application/json"
    }
   }
  },
 "outputs": {
  "body": {
   "_type": "object",
   "_array": false
  },
  "statusCode": {
   "_type": "number",
   "_array": false
  }
 }
},
{
 "brick": "control.let",
 "id": "blob",
 "inputs": {
  "id": "{{makeWebhook.body.id}}"
 },
 "outputs": {}
}
```

Core:

```
{
 "brick": "http.get",
 "id": "eventGetter",
 "inputs": {
  "url": {
   "_availableTypes": [
    "string"
   ],
   "_type": "string",
   "_array": false,
   "_value":
"{{input.data.api_url}}?token={{auth.access_token}}&expand=promotional_code"
  },
  "headers": {
   "_type": "object",
   "_array": false,
   "_value": {
    "Authorization": "Bearer {{auth.access_token}}"
   }
  }
 },
 "outputs": {
  "statusCode": {
   "_type": "number",
   "_array": false
  },
  "body": {
   "_type": "object",
   "_array": false
  }
 }
},
{
 "brick": "object.construct",
 "id": "eventBuilder",
 "inputs": {
  "Event": {
  "Name": "{{eventGetter.body.name.text}}",
  "Description": "{{eventGetter.body.description.text}}",
  "URL": "{{eventGetter.body.url}}",
  "Start Time": "{{eventGetter.body.start.utc}}",
  "End Time": "{{eventGetter.body.end.utc}}",
  "Created At": "{{eventGetter.body.created}}",
  "Changed At": "{{eventGetter.body.changed}}",
  "Capacity": "{{eventGetter.body.capacity}}",
  "Status": "{{eventGetter.body.status}}",
  "Venue ID": "{{eventGetter.body.venue_id}}",
  "Category ID": "{{eventGetter.body.category_id}}",
  "Subcategory ID": "{{eventGetter.body.subcategory_id}}",
  "Online Event?": "{{eventGetter.body.online_event}}",
  "Listed?": "{{eventGetter.body.listed}}",
  "Shareable?": "{{eventGetter.body.shareable}}",
  "Password": "{{eventGetter.body.password}}",
  "Invite Only?": "{{eventGetter.body.invite_only}}"
  }
 },
 "outputs": {
  "output": {
   "_type": "object",
   "_array": false
  }
 }
}
```

Stop:

```
{
 "brick": "http.delete",
 "id": "teardown",
 "inputs": {
  "url": {
   "_type": "string",
   "_array": false,
   "_value":
"https://www.eventbriteapi.com/v3/webhooks/{{input.blob.id}}/?token={{auth.access_t
oken}}"
   },
   "body": {
    "_availableTypes": [
     "object",
     "string"
    ],
    "_type": "object",
    "_array": false,
    "_value": {}
   },
  "headers": {
   "_type": "object",
   "_array": false,
   "_value": {
    "Authorization": "Bearer {{auth.access_token}}",
    "Content-Type": "application/json"
   }
  }
 },
 "outputs": {
  "statusCode": {
   "_type": "number",
   "_array": false
  },
  "body": {
   "_type": "object",
   "_array": false
  }
 }
}
```

Invokable Events

General

Invokable events are events that are triggered by an external service. External services will essentially trigger the event inside your connector by invoking the FLO it occupies.

Data from the external service will be passed into your connector event, which you can then use/transform like normal throughout the rest of your event. Most of the time, this will involve simply taking the information passed to you from the service, and formatting the data to match your desired UI structure.

To set up an invokable event from inside your FLO, simply select the invokable event and configure it as you need. Then, select the "</>" icon on the bottom of your event. There, you will see the URL that your external service will need to send data to, to properly trigger your FLO.

From there, whenever your service makes a call to the platform, your FLO will trigger your invokable event and subsequently the rest of that FLO.

Implementation

Note: There are two types of invokable methods. Standard "invokable", which is explained in detail below. There is also the "user-invokable" type, which allows you to close out a connection in the middle of your connector, using the HTTP.Close module. This second type is almost exactly the same, so unless otherwise noted, assume that the following directions work for both types.

To implement an invokable method, first go to the event that you would like to set to be "invokable". When you hover over its name, you'll see an ellipses menu pop up on the right. Click it, and then select "Configure Method". On this page, you should then select "invokable" under the "Event Type" menu. If you are trying to use "user-invokable" invokable methods, then keep your "Event Type" set to "Standard", and instead click on the name of your method and scroll to the bottom. After the last "]" and before the last "}", add in a "user-invokable" key, and set it to true. Make sure there isn't also an "invokable" key set to true in this area, if so, remove it.

Once that is complete, you can get started on actually building your connector method.

Just like any event, you'll need to build your UI for your params and outputs. After that, all you have to do is set up your modules to consume whatever information is coming from your external service and map it into a format that matches your outputs. Similar to what an Action does.

The information passed from your service is accessible through the Mustache reference "{{input.data.valuePathHere}}", or "{{input.raw.-body/headers/query-}}" if you are using "user-invokable" triggers.

Testing

Building an invokable event is fairly straightforward, however, testing them is far more difficult. Because the invokable event is triggered by an external service via the FLO it's built on, you cannot trigger an invokable event until it has been built into a FLO.

To get around this, it's best to understand what the shape of your data coming from your service is, by either looking at the documentation, or by setting up an API endpoint FLO, and seeing what the body of the trigger request is.

Then, inside of your event method, add an Object.Construct that contains the copy-and-pasted object from your trigger request that you can then use throughout the rest of your event method to test how to properly shape your incoming data. Once you've been able to process your data correctly, you'll need to remove the Object.Construct, and replace those references with "{{input.data}}".

Then, to truly test, you'll need to deploy your connector and build a FLO with it. At this point, you can then grab the invoke URL from the newly-made FLO and test to see if your data mapping was correct.