JavaScript

Introduction

JavaScript is available to use in scripts created as a project component only (not in scripts used within a transformation). This page provides information about JavaScript support in Jitterbit Harmony as well as some examples to get you started. Also see related pages for creating a script, using the script editor, and testing a script.

JavaScript Support in Jitterbit Harmony

Jitterbit Harmony's JavaScript engine supports the ECMA-262 v5.1 standard as specified at ECMA International. This version of JavaScript has native support of JSON and the definition and use of functions inside scripts. Jitterbit's JavaScript confirms to standard JavaScript object manipulation and behavior.

WARNING: While Jitterbit supports JavaScript based on the standard, not all JavaScript functionality is available. Jitterbit does ***not*** support these JavaScript features:

- Document Object Model (DOM) web APIs
- Mozilla built-in functions and objects
- Certain JavaScript types such as `Set` and `Map`
- Access to Java objects

Simple data types, arrays, and JSON objects are fully supported. Jitterbit maps are also supported within JavaScript. JavaScript treats Jitterbit maps as JSON objects, and Jitterbit Scripts treat JSON objects as Jitterbit maps. JSON properties are accessed using map keys.

For example, given this JSON object defined in JavaScript:

```
var $myObj = {
 "name":"John",
 "age":30,
 "cars": {
   "car1":"Ford",
   "car2":"BMW",
   "car3":"Fiat"
   }
 };
```

In a Jitterbit Script, the object would be accessed by a map. Access the `"car3"` property like this:

```
$mycar = $myObj["cars"]["car3"];
```

After you have created a new JavaScript in Cloud Studio, you can enter the script directly within the script editor. In JavaScripts used in Cloud Studio, scripts must be enclosed within a `<javascript>` opening tag and `</javascript>` closing tag.

Loop Iterations

The maximum number of loop iterations allowed in Jitterbit Harmony is 50,000. The maximum number of loop iterations in JavaScript is per script, not per loop.

For example, one JavaScript script containing three loops, where each loop executes 25,000 iterations, would be a total of 75,000 iterations running within the one script.

On This Page

Related Articles

- JavaScript Jitterbit and Common Functions
- JavaScript Standard Properties and Functions
- Jitterbit Script
- Project Pane
- Script Editor
- Script Testing
- Script Types and Creation

Related Topics

- Cloud Studio
- Functions
- Scripts
- Variables

Last updated: Jan 29, 2020

To increase the maximum number of iterations allowed in any one JavaScript script, manually add `JavaScriptMaxIterations = X`, where `X` is greater than `50000`, to `[Settings]` in `jitterbit.conf`.

For more information on increasing the maximum number of loops allowed, see `[Settings]` under Editing the Configuration File - jitterbit.conf.

For an example of a loop, see JavaScript Loop later on this page under Examples.

Component Palette

The script component palette provides access to various components that can be used within a script. You can use components within a script by dragging them from the component palette, using the autocomplete feature, or manually typing or pasting in the correct syntax.

NOTE: At this time, adding plugins, operations, notifications, or other scripts to a JavaScript is not supported.

CAUTION: If a script calls other project components that have not yet been deployed, those components must be deployed before you can run the script successfully. This is true even if those components are not added to any operations (that is, if they appear only within the **Components** tab of the project pane).

Functions

The **Functions** tab provides a list of functions available to use in a script:

The functions available to use in a JavaScript are available under four categories: **Jitterbit**, **Keywords**, **Common Functions**, and **Math**. For detailed information on each function available in JavaScript in Jitterbit Harmony, refer to these pages:

- JavaScript Jitterbit and Common Functions
- JavaScript Standard Properties and Functions

Add a function to the JavaScript using one of these methods:

- Drag the function from the palette to the script. The appropriate syntax for the script language will be inserted.
- Begin typing the function name and then press `Control+Space` to display a list of autocomplete suggestions. Select a function to insert the appropriate syntax for the script language.
- Manually enter the appropriate syntax for the script language.

For additional documentation on using functions and comprehensive documentation on Jitterbit Script and JavaScript functions, refer to Functions.

Variables

The **Variables** tab provides access to variables that are available globally to use throughout a project, including global variables, project variables, and Jitterbit variables:

Add a variable to the JavaScript using one of these methods:

- Drag the variable from the palette to the script. The appropriate syntax for the script language will be inserted.
- Begin typing the variable name and then press `Control+Space` to display a list of autocomplete suggestions. Select a variable to insert the appropriate syntax for the script language.
- Manually enter the appropriate syntax for the script language.

NOTE: Local variables are not listed because they are not available globally; however you can still use them locally within a script.

Global Variables

All Jitterbit global variables can be accessed and updated from a JavaScript. Any newly defined JavaScript global variables will become Jitterbit global variables.

The syntax used for setting and retrieving a global variable depends on whether the global variable name contains a period.

> **WARNING:** The `Jitterbit.SetVar` and `Jitterbit.GetVar` functions are designed to allow the use of variables that contain periods within the variable name. However, using periods in a variable name is not recommended. As the value will be converted to a string when the variable is set, these functions cannot be used with complex data types such as arrays, dictionaries, or JSON objects. Instead, it is recommended that you create Jitterbit variables without periods and instead use underscores in place of periods and use the standard dollar sign $ syntax as described below.

> **TIP:** Additional information on the `Jitterbit.GetVar` and `Jitterbit.SetVar` functions is in the next section under Functions.

Setting a Global Variable

- **Names without periods (recommended):** A global variable that does not contain any periods in its name can be created initially or updated using the command `var $`, or updated using a dollar sign $ without `var`.
  - **`var $`:** Using `var` and beginning with a dollar sign `$`, the code example `var $serverURL="https://www.example.com"` creates or updates a global variable called "`serverURL`" with a value of "`https://www.example.com`". New global variables that are being initialized must precede the $ with `var`.
  - **`$`:** Prefixed with a dollar sign $, the code example `$serverURL="https://www.example.com"` updates the same global variable called "`serverURL`" with the same URL. This works only for global variables that are already initialized.
- **Names with periods (not recommended):** A global variable that contains periods in its name can be created initially or updated only with the `Jitterbit.SetVar` function.
  - **`Jitterbit.SetVar`:** Using `Jitterbit.SetVar`, the code example `Jitterbit.SetVar("$server.URL", "https://www.example.com")` creates or updates a global variable called "`server.URL`" with a value of "`https://www.example.com`" that will be treated as a string. Note that the dollar sign $ ***must*** be included within the variable name, or the variable will not be global to the Harmony system.

Getting a Global Variable

- **Names without periods:** The value of a global variable that does not contain any periods in its name can be retrieved by prefixing with a dollar sign $.
  - **`$`:** Prefixed with a dollar sign $, the code example `$serverURL` retrieves the value of the global variable "`serverURL`".
- **Names with periods:** The value of a global variable that contains periods in its name can be retrieved only with the `Jitterbit.GetVar` function.
  - **`Jitterbit.GetVar`:** Using `Jitterbit.GetVar`, the code example `Jitterbit.GetVar("$server.URL")` returns the string value of the global variable called "`server.URL`". Note that the dollar sign $ ***must*** be included within the variable name to read the global value from the Harmony system.

Project Variables

Project variables are first created as a project component within Cloud Studio. Once a project variable is created, you can set values for them through Cloud Studio, the Management Console, or Citizen Integrator. Learn more about creating and updating project variables under Project Variables.

In Jitterbit JavaScript, the syntax used for retrieving the value of a project variable depends on whether the project variable name contains a period.

- **Names without periods:** The value of a project variable that does not contain any periods in its name can be retrieved by beginning with a dollar sign $.
  - **`$`:** Prefixed with a dollar sign $, the code example `$org_netsuite_auth_username` retrieves the value of the project variable called "`org_netsuite_auth_username`".

  Names with periods: The value of a project variable that contains periods in its name can be retrieved only with the `Jitterbit.GetVar` function.
  - **`Jitterbit.GetVar`:** Using `Jitterbit.GetVar`, the code example `Jitterbit.GetVar("$server.URL")` returns the value of the project variable called "`server.URL`". Note that the dollar sign $ ***must*** be included within the variable name.

Jitterbit Variables

The Harmony system defines certain global variables that are always available throughout a project, known as Jitterbit variables (or known as predefined global variables). These can be used to fetch global information such as the name of the current source file or the name of the current operation. Learn more under Jitterbit Variables.

In Jitterbit JavaScript, Jitterbit variables predefined by Harmony are accessible only with the `Jitterbit.GetVar` function. This is because all Jitterbit variables predefined by Jitterbit contain periods within the variable name.

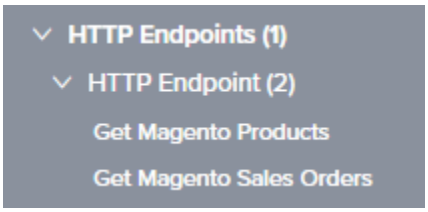
- **`Jitterbit.GetVar:`** Using `Jitterbit.GetVar`, the code example `Jitterbit.GetVar("$jitterbit.operation.error")` retrieves the value of the Jitterbit variable "`jitterbit.operation.error`". Note that the dollar sign $ ***must*** be included within the variable name.

Endpoints

The **Endpoints** tab provides a list of endpoints from the project that are available to use in the script:

The types of endpoints that can be used within scripts depends on if there are functions that support taking the specific type of connection or activity reference as an argument. The endpoint references must be used in conjunction with those functions in order to be valid in the script.

Connections and activities that can be used in the script appear within categories that list the number of each item available under each category. Activity names are preceded by square brackets containing the type of interaction with the data resource that is specific to the activity type (for example, read, write, query, upsert, GET, POST, etc.). In order to show up here, connections and activities must already be configured within the project. For example, if there is a single configured HTTP connection in the project, with two activities configured using that connection, they appear grouped as follows:



Add a connection or activity to the JavaScript using one of these methods:

- Drag the connection or activity from the palette to the script. The connection or activity reference will be inserted into the script.
- Begin typing the connection or activity name and then press `Control+Space` to display a list of autocomplete suggestions. Select a connection or activity to insert the appropriate reference into the script.
- Manually enter the connection or activity reference.

Connection references are contained within `<TAG>` and `</TAG>` tags and are constructed of the type of the project component (`connection`), followed by a colon (`:`), followed by the type of connection, followed by the user-provided connection name.

Activity references are longer, as the connection reference from which they are associated must also be included in the path. Activity references are contained within `<TAG>` and `</TAG>` tags and are constructed of the type of the project component (`activity`), followed by a colon (`:`), followed by the type of connection, followed by the type of activity, followed by the user-provided activity name.

Depending on the specific connection or activity type, you use functions from the **Functions** tab that take the connector reference as the argument. For more information, refer to the documentation on JavaScript functions:

- JavaScript Jitterbit and Common Functions
- JavaScript Standard Properties and Functions

Examples

These JavaScript examples are provided for reference.

JavaScript File Functions

JavaScript File Functions

```
<javascript>
// This script will:
// * Generate some random numbers
// * Write them to a target file
// * Then read them back in
// * Write output to the Operation Log
// ***********************************************

// Get 200 random numbers between 1 and 10000
var mystring = getRandomNumbers(200,1,10000);

// Write the data to a file
Jitterbit.WriteFile("<TAG>activity:tempstorage/Temporary Storage Endpoint
/tempstorage_write/tmpdata</TAG>", mystring);

// Read the data back in from the file
var filedata = Jitterbit.ReadFile("<TAG>activity:tempstorage/Temporary
Storage Endpoint/tempstorage_read/tmpdata</TAG>");

// Output to the Operation Log
WriteToOperationLog("Read file, output: " + filedata);

// Displays the data in the result of the Studio test script tab
SetScriptResult(filedata);

/////////////////

function getRandomNumbers(howMany,min,max) {
  var output = "";

  for (var i=0; i<howMany; i++) {
    output = output + getRandomNumber(min,max) + " \n";
  }

  return output;
}

function getRandomNumber(min,max) {
  return Math.floor((Math.random() * max) + min);
}

/////////////////
</javascript>
```

JavaScript Math Functions

JavaScript Math Functions

```
<javascript>
// Create 200 random numbers
var $output = getRandomNumbers(200);

WriteToOperationLog($output);
SetScriptResult($output);

/////////////////

function getRandomNumbers(howMany) {
  var output = "";

  for (var i=0; i<howMany; i++) {
    output = output + Math.floor((Math.random() * 10000) + 1) + " \n";
  }

  return output;
}

/////////////////
</javascript>
```

JavaScript Loop

JavaScript Loop

```
<javascript>
// Create 100 random numbers

var $output = "";

for (var i=0; i<100; i++) {
  $output = $output + Math.floor((Math.random() * 10000) + 1) + " \n";
}

SetScriptResult($output);
</javascript>
```

JavaScript JSON Example 1

JavaScript JSON Example 1

```
<javascript>
WriteToOperationLog("\n\n Parsing JSON...");

var jsonData = Jitterbit.ReadFile("<TAG>activity:tempstorage/Temporary
Storage Endpoint/tempstorage_read/JSON Data</TAG>");
var $jsonObj = JSON.parse(jsonData);

WriteToOperationLog("Value of 'status' is: " + $jsonObj.status);
WriteToOperationLog("Value of 'operation' is: " + $jsonObj.operation);
WriteToOperationLog("Value of 'serverUrl' is: " + $jsonObj.serverUrl);

var $firstOrg = $jsonObj.orgAttrs[0];

WriteToOperationLog("First Org ID is: " + $firstOrg.orgId);
WriteToOperationLog("First Org Name is: " + $firstOrg.orgName);
</javascript>
```

JavaScript JSON Example 2

JavaScript JSON Example 2

```
<javascript>
WriteToOperationLog("\n\n Parsing JSON...");

var jsonData = Jitterbit.ReadFile("<TAG>activity:tempstorage/Temporary
Storage Endpoint/tempstorage_read/JSON Data</TAG>");
var $jsonObj = JSON.parse(jsonData);

WriteToOperationLog("Status: " + $jsonObj.status);
WriteToOperationLog("Operation: " + $jsonObj.operation);

var orgs = "";
var needComma = false;

for (var i=0; i<$jsonObj.orgAttrs.length; i++) {
  if (needComma) orgs = orgs + ",";
  orgs = orgs + $jsonObj.orgAttrs[i].orgId;
  needComma = true;
}

WriteToOperationLog("Org IDs: " + orgs);

// You can modify existing JSON values
// Any changes will be reflected in the Jitterbit system as a map variable
// Here we'll insert a random number as an authentication token
var randomNumber = Math.floor((Math.random() * 10000) + 1);
$jsonObj.authenticationToken = randomNumber;
</javascript>
```

JavaScript JSON Example 3

JavaScript JSON Example 3

```
<javascript>
// This script uses JSON stringify
// to create a property value structure
// and then pushes it to an API

var $complexAPI = {
  "properties": [
    {
      "property": "email",
      "value": $email
    },
    {
      "property": "firstname",
      "value": $firstname
    },
    {
      "property": "lastname",
      "value": $lastname
    },
    {
      "property": "website",
      "value": $website
    },
    {
      "property": "phone",
      "value": $phone
    }
  ]
}

var $outputJSON = JSON.stringify($complexAPI);
Jitterbit.WriteFile("<TAG>activity:http/HTTP Endpoint/http_post/Example
HTTP POST</TAG>", $outputJSON);
WriteToOperationLog($outputJSON);
SetScriptResult($outputJSON);

</javascript>
```