Jitterbit Script

Introduction

The Jitterbit Script language can be used in all types of scripts within Jitterbit Harmony, including within operations and transformations. Scripts are created in Jitterbit Script language by default. This page provides information specific to the Jitterbit Script language, such as syntax, data types, operators, escape sequences, and control structures. Also see related pages for creating a script, using the script editor, and testing a script.

Syntax

In Jitterbit Script, scripts must be enclosed within a `<trans>` opening tag and `</trans>` closing tag, unless using functions that specifically call for code to be placed outside of these tags, such as several Database Functions.

Within those tags, a Jitterbit Script consists of built-in functions, variables, or logic to execute, separated by a semi-colon (`;`).

The result that is returned will be the returned value of the last statement of the script before the closing `</trans>` tag.

Comments

Within the `<trans>` ... `</trans>` tags, the use of two forward slashes `//` marks the start of a single-line comment, and affects the text to the end of that line. Comments will not be part of the script that is run or the transformed result.

For example, a comment on a single line might look like this:

Single-Line Comment in Jitterbit Script

```
<trans>
// This is a comment
DbLookup(...)
</trans>
```

You can also use a block-style comment:

Multi-Line Comment in Jitterbit Script

```
<trans>
/* This is a multi-line comment
This line is also a comment
This is the final line of the comment */
DbLookup(...)
</trans>
```

Control Structures

The Jitterbit Script language does not include control statements such as `if` or `while` loops. Instead, you can use Jitterbit functions to achieve the same functionality.

See the `Case`, `If`, and `While` functions under Logical Functions. The `Eval` function under General Functions can be used as a "try-catch" statement.

A maximum of 50,000 loop iterations is allowed for each individual loop in Jitterbit Scripts. To increase the allowed number of iterations per loop, see `jitterbit.scripting.while.max_iterations` in Scripting Jitterbit Variables.

Regular Expressions

Jitterbit Harmony supports regular expressions as means of specifying and recognizing strings of text, including particular characters, words, or character patterns. Jitterbit Harmony supports the regular expression syntax from the Perl 5 programming language.

On This Page

- Introduction
- Syntax
  - Comments
  - Control Structures
  - Regular Expressions
  - Escape Sequences
  - Operators
  - Operator Precedence
- Component Palette
  - Source Objects
  - Functions
  - Variables
  - Plugins
  - Operations
  - Notifications
  - Scripts
  - Endpoints
- Data Types
  - Arrays
  - Dictionaries

Related Articles

- JavaScript
- Mapping Source Objects
- Nodes and Fields
- Project Pane
- Script Editor
- Script Testing
- Script Types and Creation

Related Topics

- Cloud Studio
- Connectors
- Functions
- Notifications
- Operations
- Plugins
- Scripts
- Transformations
- Variables

Last updated: Nov 20, 2019

```
<trans>
RegExMatch(<input>,<expression>,...)
</trans>
```

Escape Sequences

Jitterbit Harmony recognizes these escape sequences when used in literal strings:

Sequence	Definition
`\t`	Tab
`\r`	Carriage return
`\n`	New line

Literal strings must be surrounded by double quotes (`"`) or single quotes (`'`).

Any other quote must be escaped if used in the string. For example:

```
$str = "String with line break.\nThat's the last line.";

$str = 'Tony "The Gun" Marcello';

$str = "Tony \"The Gun\" Marcello";
```

Operators

This is a summary of the operators supported by Jitterbit Script. Jitterbit Script will attempt to convert the arguments to enable the operation. If this is not possible, an error will be reported.

Operator	Definition
`=`	Assigns to a variable. The right-hand argument will be assigned to the left-hand argument.
`+`	Adds two numbers or concatenates two strings. If a string is added to anything else, both arguments are converted to strings. If both arguments are numbers, the result will be of type double.
`++`	Increments the value by 1. Example: `$count++;` if prefixed, assignment precedes the increment.
`+=`	Concatenates a string or adds numbers to the target variable. Example: `$count+=2` is the same as `$count=$count+2`.
`-`	Subtracts two numbers or if unary, multiplicative inverse. The result will be of type double.
`--`	Decrements a value by 1. Example: `$count--;` if prefixed, assignment precedes the decrement.
`-=`	Subtracts numbers from the target variable. Example: `$count-=2` is the same as `$count=$count-2`.
`/`	Divides two numbers. The result will be of type double.
`*`	Multiplies two numbers. The result will be of type double.
`^`	Raises the first argument to the power of the second argument. If both arguments are integers, the result is an integer.
`&`	Logical AND operator. The result will be of type boolean. && can also be used. This is always a short-circuit operator, meaning that if the left-hand argument evaluates to false, the right-hand argument will not be evaluated.
`|`	Logical OR operator. The result will be of type boolean. `||` can also be used. This is always a short-circuit operator, meaning that if the left-hand argument evaluates to true, the right-hand argument will not be evaluated.

`==`	Equivalence operator. Returns true if the arguments are equal. Returns false is they are not equal.
`!=`	Not equivalent operator. Returns true if the arguments are not equal. Returns false if they are equal.

`<`
`>`
`<=`
`>=`
	Comparison operators. Returns true or false.

`!`	Negation operator. Converts a true value to false and vice versa. Example: `!IsNull($org.workorder.id)`

`{ }`	Used to build an array. Examples:
	`$a={"London","Paris","New York"};`
	`$b={{"John",25},`
	`    {"Steve",32},`
	`    {"Daniel",26}`
	`   };`
	`$c={"\"quoted\"", '"quoted"'};`

Operator Precedence

This table shows the precedence of the operators, from highest to lowest:

Operators	Description
`( )`	Parentheses for grouping
`{ }`	Braces for arrays
`++ -- -`	pre- and post- unary operators, multiplicative inverse
`^`	Power
`!`	Negation
`* /`	Multiplication, division
`+ -`	Addition, subtraction
`< > == != <= >=`	Comparison operators
`& &&`	Logical AND
`| ||`	Logical OR
`= += -=`	Assignment

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.

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).

Source Objects

The **Source Objects** tab is present only for scripts created within a transformation:

Within the script, you can reference source data by the path of the field.

Add a source object to the Jitterbit Script using one of these methods:

- Drag the object from the palette to the script. The full reference path to the source object using the appropriate syntax will be inserted.
- Manually enter the full reference path to the source object.

For an explanation of the node/field notation, refer to Nodes and Fields.

This example assigns a source object to a global variable:

```
<trans>
$org.calculate.operand1=root$transaction.request$body$Calculate$input.
Operand1$;
</trans>
```

In addition, although nodes cannot be mapped to target fields, they are able to be used with some XML Functions.

To enter a node path to use with the function, manually enter the reference path of the desired XML node folder. For an explanation of the node/field notation and how to construct node reference paths, refer to R eference Paths under Mapping Source Objects.

Functions

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

Add a function to the Jitterbit Script 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, see 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 Jitterbit Script 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.

Global variables are accessed either by typing a $ sign before the variable name or by using the `Set()` a nd `Get()` functions.

Global variable names can be composed from these characters: letters (a-z, A-Z), numbers (0-9), periods, and underscores. Other characters, such as hyphens, are not recommended and may cause issues. (Note that in the case of local variables, the period character is not recommended either.)

This example references a global variable using $ to create a global dictionary:

```
<trans>
$org.lookup.currency=Dict();
$org.lookup.currency[1]="USD";
$org.lookup.currency[2]="EUR";
</trans>
```

This example assigns a global variable to another global variable using the `Set()` and `Get()` functions:

```
<trans>
Set("op2", Get("op"));
</trans>
```

Note that for certain Jitterbit variables, because they include a hyphen, you must reference them by using either the `Set` or `Get` functions:

```
<trans>
Get("jitterbit.networking.http.request.header.content-type");
</trans>
```

Local variables, although not listed within the **Variables** tab, are not listed because they are not available globally; however you can still use them locally within a script. Local variables have these characteristics:

- A local variable is defined and used only within a specific script. Therefore, conflict between a local variable's value in the current script vs. the value of a local variable with the same name in another script is not a concern.
- The local variable cannot be defined or retrieved by the `Set()` or `Get()` functions.
- `RunScript()` can pass arguments to the script, for example `RunScript(SCRIPT_ID, 5, "abc",...)`. Values in the script can be assigned to predefined local variables `_1`, `_2`, `....` In this example, `_1` represents the integer value of `5`, while `_2` represents the string value `"abc"`. The local variable must be defined before it can be referenced, except in the case of the input arguments `_1`, `_2`, `...` described above.
- The `ArgumentList()` function is available for redefinition of a list of local variables as input arguments.

For more detailed information on each type of variable and examples, refer to Variables.

Plugins

The **Plugins** tab provides a list of plugins that can be run inside the script:

Add a plugin to the Jitterbit Script using one of these methods:

- Drag the plugin from the palette to the script. Both the `RunPlugin()` function and the plugin reference will be inserted into the script.
- Begin typing the plugin name and then press `Control+Space` to display a list of autocomplete suggestions. Select a plugin to insert the plugin reference into the script.
- Manually enter the plugin reference.

The plugin reference is contained within `<TAG>` and `</TAG>` tags and is constructed of the type of the project component (`plugin`), followed by a colon (`:`), followed by the plugin name from the XML manifest. Note that the plugin name from the XML manifest is not necessarily the same as the display name in the Management Console.

TIP: Only plugins that are available to use within a script will be listed. Additional plugins can be applied on an activity within an operation. For documentation on using plugins, including how to configure a plugin at the activity level and set global variables, see Plugins.

Operations

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

Add an operation to the Jitterbit Script using one of these methods:

- Drag the operation from the palette to the script. Both the `RunOperation()` function and the operation reference will be inserted into the script.
- Begin typing the operation name and then press `Control+Space` to display a list of autocomplete suggestions. Select an operation to insert the operation reference into the script.
- Manually enter the operation reference.

The operation reference is contained within `<TAG>` and `</TAG>` tags and is constructed of the type of the project component (`operation`), followed by a colon (`:`), followed by the user-provided operation name.

In addition to using the `RunOperation()` function, you can also use the operation reference with other functions from the **Functions** tab that take the operation reference as an argument. These include functions listed in General Functions that use an `operationInstanceGUID`, `operationId`, or `operationTag` as a parameter. For example:

- `CancelOperation()`
- `GetLastOperationRunStartTime()`
- `GetOperationQueue()`
- `RunOperation()`
- `WaitForOperation()`

Notifications

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

Add a notification to the Jitterbit Script using one of these methods:

- Drag the notification from the palette to the script. Both the `SendEmailMessage()` function and the notification reference will be inserted into the script.
- Begin typing the notification name and then press `Control+Space` to display a list of autocomplete suggestions. Select a notification to insert the notification reference into the script.
- Manually enter the notification reference.

The notification reference is contained within `<TAG>` and `</TAG>` tags and is constructed of the type of the project component (`email`), followed by a colon (`:`), followed by the user-provided email notification name.

Scripts

The **Scripts** tab provides a list of all other scripts that are project components, including both Jitterbit Scripts and JavaScripts, that are available to use in the script:

Add another script within the Jitterbit Script using one of these methods:

- Drag the script from the palette to the script. Both the `RunScript()` function and the script reference will be inserted into the script.
- Begin typing the script name and then press `Control+Space` to display a list of autocomplete suggestions. Select a script to insert the script reference into the script.
- Manually enter the script reference.

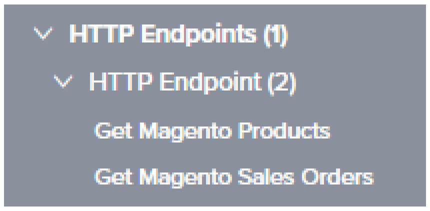
The script reference is contained within `<TAG>` and `</TAG>` tags and is constructed of the type of the project component (`script`), followed by a colon (`:`), followed by the user-provided script name.

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 Jitterbit Script 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 endpoint, 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 as listed below, you use functions from the **Functio ns** tab that take the connector reference as the argument. The functions listed below are available to be used with the listed connections and activities.

- **Database Endpoints:** This category lists any configured database connections and associated activities, both of which are able to be used in scripts depending on the specific function.
  - **Connections:** Any functions listed in Database Functions that use a `databaseId` as a parameter can be used with database connections, including:
    - `CacheLookup()`
    - `CallStoredProcedure()`
    - `DBCloseConnection()`
    - `DBExecute()`
    - `DBLookup()`
    - `DBLookupAll()`
    - `DBRollbackTransaction()`
    - DBWrite()
  - **Activities:** Any functions listed in Database Functions that use a database target as a parameter can be used with database activities, namely:
    - DBLoad()
- **File Share Endpoints**, **FTP Endpoints**, **HTTP Endpoints**, **Local Storage Endpoints**, and **Tem porary Storage Endpoints:** These categories include any configured file share, FTP, HTTP, loc al storage, or temporary storage connections (***not*** able to be used in a script) and associated activities (able to be used in a script), respectively.

  - **Activities:** Any functions listed in Cryptographic Functions, Database Functions, File Functions, or Salesforce Functions that use a `sourceId` or `targetId` as a parameter can be used with these types of activities.

    NOTE: At this time, this category also includes configured API connections and activities, which are unable to be used with these functions.

    - `ArchiveFile()`
    - `Base64EncodeFile()`
    - `DBLoad()`
    - `DBWrite()`
    - `DeleteFile()`
    - `DeleteFiles()`
    - `DirList()`

        - `FileList()`
        - `FlushAllFiles()`
        - `FlushFile()`
        - `ReadFile()`
        - `SfLookupAllToFile()`
        - `WriteFile()`
- **Salesforce Endpoints:** This category includes any configured Salesforce connections. Salesforce activities are not included, as they are not able to be used in a script.
    - **Connections:** Any functions listed in Salesforce Functions that use a `salesforceOrg` as a parameter can be used with a Salesforce connection, including:
        - `LoginToSalesforceAndGetTimeStamp()`
        - `SalesforceLogin()`
        - `SetSalesforceSession()`
        - `SfCacheLookup()`
        - `SfLookup()`
        - `SfLookupAll()`
        - `SfLookupAllToFile()`
- **NetSuite Endpoints:** This category includes any configured NetSuite connections. NetSuite activities are not included, as they are not able to be used in a script.
    - **Connections:** Any functions listed in NetSuite Functions that use a `netSuiteOrg` as a parameter can be used with a NetSuite connection, including:
        - `NetSuiteGetSelectValue`
        - `NetSuiteGetServerTime`
        - `NetSuiteLogin`

Data Types

All source objects and global variables that are not null have a data type associated with them. Various data types can be changed using the functions in the Conversion Functions category.

These data types are supported in Jitterbit Scripts:

Type	Description	Classification
`binary`	Binary	Data
`bit`	Bit	Data
`bool`	Boolean	Logical
`int`	Integer	Numerical
`double`	Double	Numerical
`float`	Float	Numerical
`long`	Long	Numerical
`date`	Date	Date & Time
`timespan`	Timespan (Date with a time)	Date & Time
`string`	String	String
`array`	Array	Collection
`dictionary`	Dictionary (also known as a `map`)	Collection
`instance`	Instance in a data source or target	Schema
`node`	Node in a schema of a data source or target	Schema
`type`	Any of these types	Data
`null`	Null value	Data
`var`	Variable reference, either local or global	Script

Arrays

An array is a collection of variables. Each member in the collection can be of any supported type, including arrays. The members of an array can be accessed using the `Get()` and `Set()` methods or using the `[] array` syntax. Arrays are zero-indexed, and indices are numeric, sequential, and cannot be skipped.

You can also create arrays of global variables. An array global variable is an array of other global variables that in turn can be arrays.

Setting an Array

You can set values in an array using the `Set()` function with this syntax:

```
type Set(string name, type value [, int index1, int index2, ...])
```

This sets the value of the global variable with the given name to value, and returns the value. If the first argument is an array or the name of an array global variable, you can set the value of an array variable by specifying its index (or indices for multi-dimensional arrays) as the third argument.

Not all the items in an array have to be of the same type. For example, you can create an array that holds a date, an integer, and a string. You can even create arrays inside other arrays.

This example creates an array with three variables of different types where each entry represents the current date and time:

```
<trans>
$right_now = Now();
Set($now, $right_now, 0);
Set($now, Long($right_now), 1);
Set($now, String($right_now), 2);
</trans>
```

As arrays are zero-indexed, the first element is at index 0 and the last element is at index (size-1). To append data to an array, pass either a negative index value or the size of the array (`Length($arr)`). Set ting an element with an index larger than the size of the array results in an index out-of-range error. Setting non-array data elements can also be done using the `$de_name` syntax.

Here are examples of setting values in an array:

```
// Set the n:th entry in an array to the string "value"
Set($arr, "value", n-1);

// Another way to set the n:th entry of an array
Set($arr, "value", Length($arr));

// Sets the value to a new variable at the end of the array
Set($arr, "value", -1);

// Set the n:th entry of the m:th array
Set($record_set, "value", m-1, n-1);
```

TIP: For additional syntax that can be used to define values in an array, see Dictionary and Array Functions.

Accessing an Array

You can access the items of an array using the `Get()` method with this syntax:

```
type Get(string name[, int index1, int index2, ...])
```

This returns the value of the global variable with the given name. If the first argument is an array or the name of an array global variable, you can get a specific variable by specifying its index (or indices for a multi-dimensional array such as a record-set) as the second argument.

Some Functions have array return values. For example, `SelectNodesFromXMLAny()` returns the results of an XPath query as an array. As another example, `DbExecute()` returns a record set as a two-dimensional array: rows first, then columns. This returns the selected data as an array of arrays (representing the selected rows and columns):

```
<trans>
$resultSet = DbExecute("<TAG>endpoint:database/My Database</TAG>", "select
Result from SimpleCalculatorResults");
$firstRow = Get($resultSet, 0);
$thirdColumnOfSecondRow = $resultSet[2][3];
$secondColumnOfThirdRow = Get($resultSet, 3, 2);
</trans>
```

Arrays are zero-indexed. To access the n:th variable of an array called `"arr"`, use `Get("arr", n-1)`. For multi-dimensional arrays you need to specify all the indices. To access the `n:th` column of the `m:th` row in an array called `ResultSet` you would use `Get("ResultSet", m-1, n-1)`. Attempting to get a variable beyond the end of the array will result in an array out-of-range error.

These are examples of retrieving values from an array:

```
// Return the third array variable
Get($arr, 2);

// Another way to return the third array variable
Get("arr", 2);

// Get the n:th variable of the m:th array in arr
Get($arr, m-1, n-1);
```

This example shows how you can first create a script that builds and returns an array. The second block shows running that script and assignings its returned value to a variable.

Build Array

```
<trans>
// Script to build and return an array
a = Array();
a[Length(a)] = "A";
a[Length(a)] = "B";
a[Length(a)] = "C";
a;
</trans>
```

Call Script to Get Array

```
<trans>
// Call the script to retrieve the array
$Arr = RunScript("<TAG>script:Build Array</TAG>");
</trans>
```

TIP: For additional syntax that can be used to access values in an array, see Dictionary and Array Functions.

Dictionaries

In Jitterbit Script, a dictionary is a special type of global variable array that holds key-value pairs. The steps and functions are:

1. Initialize the dictionary using the `Dict` function:

   ```
   $d = Dict();
   ```

2. Load data with a key and a value:

   ```
   $d['4011'] = 'Banana';
   $d['4063'] = 'Tomato';
   ```

3. Check if the key already exists in the dictionary using the `HasKey` function:

   ```
   HasKey($d,'4011'); // Returns true
   ```

   In the example, we have already loaded the key '4011' so `HasKey` would return `true`.

   ```
   HasKey($d,'4089'); // Returns false
   ```

   If the key were not already loaded, for example '4089' the `HasKey` would return `false`.
4. Look up the value in the dictionary by passing the key and getting back the value:

   ```
   $d['4011']; // Returns 'Banana'
   ```

   In this example, the value returned would be `Banana`.

With dictionaries, keep these characteristics in mind:

- The scope of dictionaries is limited to the workflow. For example, if an operation loads `$MyDict` with 10,000 records, only those operations that are linked using On Success or On Failure paths, or with RunOperation() will have access to that dictionary. But, if an operation uses chunking and threading, and has a transformation that populates a dictionary, the dictionary will be inconsistent. This is because Jitterbit Harmony does not take the values assigned to variables by multiple operation threads and concatenate into a single value set. This is true for all global variables or arrays. Use the default chunking/threading values when building an operation that populates dictionaries.
- Dictionaries, because they use a binary search, are very fast at finding keys and returning values. A key can usually be found within five to six tries. In contrast, compare this type of search to looping through a 10,000 record array to find a key.
- Dictionaries are not written to memory, so they will not materially impact available server memory for processing.