

Continuity Reactor HTTP REST API

Contents

Introduction	6
Conventions	6
Status Codes	7
SSL Required for Sandboxed Continuity Reactor	7
Working with Reactor Security	7
Stream HTTP API	8
Creating a Stream	8
HTTP Responses	8
Example	8
Comments	8
Sending Events to a Stream	9
HTTP Responses	9
Example	9
Comments	9
Reading Events from a Stream: Getting a Consumer-ID	10
HTTP Responses	10
Example	10
Comments	10
Reading Events from a Stream: Using the Consumer-ID	11
HTTP Responses	11
Example	11
Comments	11
Truncating a Stream	12
HTTP Responses	12
Example	12
Setting Time-To-Live Property of a Stream	13
HTTP Responses	13
Example	13
Reading Multiple Events	13
Dataset HTTP API	14
Listing all Datasets	14
Creating a Dataset	15
HTTP Responses	15
Example	15

Deleting a Dataset	16
HTTP Responses	16
Example	16
Deleting all Datasets	17
HTTP Responses	17
Truncating a Dataset	17
HTTP Responses	17
Data HTTP API (Deprecated)	18
Creating a new Table	18
HTTP Responses	18
Example	18
Comments	18
Writing Data to a Table	19
HTTP Responses	19
Example	19
Comments	19
Reading Data from a Table	20
HTTP Responses	20
Example	20
Comments	21
Increment Data in a Table	22
HTTP Responses	22
Example	22
Comments	22
Delete Data from a Table	23
HTTP Responses	23
Example	23
Comments	23
Deleting Data from a Dataset	24
HTTP Responses	24
Example	24
Comments	24
Encoding of Keys and Values	25
Counter Values	26
Query HTTP API	27
Submitting a Query	27

HTTP Responses	27
Comments	27
Status of a Query	28
HTTP Responses	28
Comments	28
Obtaining the Result Schema	28
HTTP Responses	28
Comments	28
Retrieving Query Results	29
HTTP Responses	29
Comments	29
Closing a Query	30
HTTP Responses	30
Canceling a Query	30
HTTP Responses	30
Procedure HTTP API	31
Executing Procedures	31
HTTP Responses	31
Example	31
Reactor Client HTTP API	32
Deploy an Application	32
Delete an Application	32
Start, Stop, Status, and Runtime Arguments	33
Examples	33
Container Information	34
Scale	35
Scaling Flowlets	35
Examples	35
Scaling Procedures	36
Example	36
Scaling Services	37
Example	37
Run History and Schedule	38
Example	38
Example	38
Promote	39

Example	39
Logging HTTP API	40
Downloading Logs	40
Example	40
Example	40
Comments	41
Metrics HTTP API	42
Metrics Requests	42
Examples	42
Comments	42
Time Range	44
Available Contexts	45
Available Metrics	47
Monitor HTTP API	48
Details of All Available System Services	48
HTTP Responses	48
Checking Status of All Reactor System Services	48
HTTP Responses	48
Checking Status of a Specific Reactor System Service	49
Example	49
HTTP Responses	49
Scaling System Services	50
Examples	50

Introduction

The Continuity Reactor has an HTTP interface for a multitude of purposes:

- **Stream:** sending data events to a Stream, or to inspect the contents of a Stream
- **Dataset:** interacting with Datasets, Dataset Modules, and Dataset Types
- **Data:** interacting with Datasets (deprecated)
- **Query:** sending ad-hoc queries to Reactor Datasets
- **Procedure:** sending calls to a stored Procedure
- **Reactor Client:** deploying and managing Applications, and managing the life cycle of Flows, Procedures, MapReduce jobs, Workflows, and Custom Services
- **Logging:** retrieving Application logs
- **Metrics:** retrieving metrics for system and user Applications (user-defined metrics)
- **Monitor:** checking the status of various Reactor services, both System and Custom

Note: The HTTP interface binds to port 10000. This port cannot be changed.

Conventions

In this API, *client* refers to an external application that is calling the Continuity Reactor using the HTTP interface.

In this API, *Application* refers to a user Application that has been deployed into the Continuity Reactor.

All URLs referenced in this API have this base:

```
http://<gateway>:10000/v2
```

where <gateway> is the URL of the Continuity Reactor. The base URL is represented as:

```
<base-url>
```

For example:

```
PUT <base-url>/streams/<new-stream-id>
```

means

```
PUT http://<gateway>:10000/v2/streams/<new-stream-id>
```

Text that are variables that you are to replace is indicated by a series of angle brackets (< >). For example:

```
PUT <base-url>/streams/<new-stream-id>
```

indicates that—in addition to the <base-url>—the text <new-stream-id> is a variable and that you are to replace it with your value, perhaps in this case *mystream*:

```
PUT <base-url>/streams/mystream
```

Status Codes

Common status codes returned for all HTTP calls:

Code	Description	Explanation
200	OK	The request returned successfully
400	Bad Request	The request had a combination of parameters that is not recognized
401	Unauthorized	The request did not contain an authentication token
403	Forbidden	The request was authenticated but the client does not have permission
404	Not Found	The request did not address any of the known URIs
405	Method Not Allowed	A request was received with a method not supported for the URI
409	Conflict	A request could not be completed due to a conflict with the current resource state
500	Internal Server Error	An internal error occurred while processing the request
501	Not Implemented	A request contained a query that is not supported by this API

Note: These returned status codes are not necessarily included in the descriptions of the API, but a request may return any of these.

SSL Required for Sandboxed Continuity Reactor

When you interact with a Sandboxed Continuity Reactor, the Continuity HTTP APIs require that you use SSL for the connection and that you authenticate your request by sending your API key in an HTTP header:

```
X-Continuity-APIKey: <api-key>
```

Parameter	Description
<api-key>	Continuity Reactor API key, obtained from an account at Continuity Accounts

Working with Reactor Security

When working with a Reactor cluster with security enabled (`security.enabled=true` in `continuity-site.xml`), all calls to the HTTP APIs must be authenticated. Clients must first obtain an access token from the authentication server (see the "Security" section of the guide). In order to authenticate, all client requests must supply this access token in the `Authorization` header of the request:

```
Authorization: Bearer wohng8Xae7thahfohshahphaeNeeM5ie
```

For Reactor-issued access tokens, the authentication scheme must always be `Bearer`.

Stream HTTP API

This interface supports creating Streams, sending events to a Stream, and reading single events from a Stream.

Streams may have multiple consumers (for example, multiple Flows), each of which may be a group of different agents (for example, multiple instances of a Flowlet).

In order to read events from a Stream, a client application must first obtain a consumer (group) id, which is then passed to subsequent read requests.

Creating a Stream

A Stream can be created with an HTTP PUT method to the URL:

```
PUT <base-url>/streams/<new-stream-id>
```

Parameter	Description
<new-stream-id>	Name of the Stream to be created

HTTP Responses

Status Codes	Description
200 OK	The event either successfully created a Stream or the Stream already exists

Example

HTTP Method	PUT <base-url>/streams/mystream
Description	Create a new Stream named <i>mystream</i>

Comments

- The <new-stream-id> should only contain ASCII letters, digits and hyphens.
- If the Stream already exists, no error is returned, and the existing Stream remains in place.

Sending Events to a Stream

An event can be sent to a Stream by sending an HTTP POST method to the URL of the Stream:

```
POST <base-url>/streams/<stream-id>
```

Parameter	Description
<stream-id>	Name of an existing Stream

HTTP Responses

Status Codes	Description
200 OK	The event was successfully received
404 Not Found	The Stream does not exist

Note: The response will always have an empty body

Example

HTTP Method	POST <base-url>/streams/mystream
Description	Send an event to the existing Stream named <i>mystream</i>

Comments

- The body of the request must contain the event in binary form.
- You can pass headers for the event as HTTP headers by prefixing them with the *stream-id*:

```
<stream-id>.<property>:<string value>
```

After receiving the request, the HTTP handler transforms it into a Stream event:

1. The body of the event is an identical copy of the bytes found in the body of the HTTP post request.
2. If the request contains any headers prefixed with the *stream-id*, the *stream-id* prefix is stripped from the header name and the header is added to the event.

Reading Events from a Stream: Getting a Consumer-ID

Get a *Consumer-ID* for a Stream by sending an HTTP POST method to the URL:

```
POST <base-url>/streams/<stream-id>/consumer-id
```

Parameter	Description
<stream-id>	Name of an existing Stream

HTTP Responses

Status Codes	Description
200 OK	The event was successfully received and a new <code>consumer-id</code> was returned
404 Not Found	The Stream does not exist

Example

HTTP Method	POST <base-url>/streams/mystream/consumer-id
Description	Request a <i>Consumer-ID</i> for the Stream named <i>mystream</i>

Comments

- Streams may have multiple consumers (for example, multiple Flows), each of which may be a group of different agents (for example, multiple instances of a Flowlet).
- In order to read events from a Stream, a client application must first obtain a consumer (group) id, which is then passed to subsequent read requests.
- The *Consumer-ID* is returned in a response header and—for convenience—also in the body of the response:

```
X-Continuity-ConsumerId: <consumer-id>
```

Once you have the *Consumer-ID*, single events can be read from the Stream.

Reading Events from a Stream: Using the Consumer-ID

A read is performed as an HTTP POST method to the URL:

```
POST <base-url>/streams/<stream-id>/dequeue
```

Parameter	Description
<stream-id>	Name of an existing Stream

The request must pass the `Consumer-ID` in a header of the form:

```
X-Continuity-ConsumerId: <consumer-id>
```

HTTP Responses

Status Codes	Description
200 OK	The event was successfully received and the result of the read was returned
204 No Content	The Stream exists but it is either empty or the given <code>Consumer-ID</code> has read all the events in the Stream
404 Not Found	The Stream does not exist

Example

HTTP Method	POST <base-url>/streams/mystream/dequeue
Description	Read the next event from an existing Stream named <i>mystream</i>

Comments

The read will always return the next event from the Stream that was inserted first and has not been read yet (first-in, first-out or FIFO semantics). If the Stream has never been read from before, the first event will be read.

For example, in order to read the third event that was sent to a Stream, two previous reads have to be performed after receiving the `Consumer-ID`. You can always start reading from the first event by getting a new `Consumer-ID`.

The response will contain the binary body of the event in its body and a header for each header of the Stream event, analogous to how you send headers when posting an event to the Stream:

```
<stream-id>.<property>:<value>
```

Truncating a Stream

Truncation means the deletion of all events that were written to the Stream. This is permanent and cannot be undone. A Stream can be truncated with an HTTP POST method to the URL:

```
POST <base-url>/streams/<stream-id>/truncate
```

Parameter	Description
<stream-id>	Name of an existing Stream

HTTP Responses

Status Codes	Description
200 OK	The Stream was successfully truncated
404 Not Found	The Stream <stream-id> does not exist

Example

HTTP Method	POST <base-url>/streams/mystream/truncate
Description	Delete all events in the Stream named <i>mystream</i>

Setting Time-To-Live Property of a Stream

The Time-To-Live (TTL) property governs how long an event is valid for consumption since it was written to the Stream. The default TTL for all Streams is infinite, meaning that events will never expire. The TTL property of a Stream can be changed with an HTTP PUT method to the URL:

```
PUT <base-url>/streams/<stream-id>/config
```

Parameter	Description
<stream-id>	Name of an existing Stream

The new TTL value is passed in the request body as:

```
{ "ttl" : <ttl-in-seconds> }
```

Parameter	Description
<ttl-in-seconds>	Number of seconds that an event will be valid for since ingested

HTTP Responses

Status Codes	Description
200 OK	The stream TTL was changed successfully
400 Bad Request	The TTL value is not a non-negative integer
404 Not Found	The Stream does not exist

Example

HTTP Method	PUT <base-url>/streams/mystream/config with the new TTL value as a JSON string in the body: <pre>{ "ttl" : 86400 }</pre>
Description	Change the TTL property of the Stream named <i>mystream</i> to 1 day

Reading Multiple Events

Reading multiple events is not supported directly by the Stream HTTP API, but the command-line tool `stream-client` demonstrates how to view *all*, the *first N*, or the *last N* events in the Stream.

For more information, see the Stream Command Line Client `stream-client` in the `/bin` directory of the Continuuity Reactor SDK distribution.

For usage and documentation of options, run at the command line:

```
$ stream-client --help
```

Dataset HTTP API

The Dataset API allows you to interact with Datasets through HTTP. You can list, create, delete, and truncate Datasets. For details, see the Developer Guide:

[Continuity Reactor Advanced Features, Datasets section](#)

Listing all Datasets

You can list all Datasets in the Continuity Reactor by issuing an HTTP GET request to the URL:

```
GET <base-url>/data/datasets
```

The response body will contain a JSON-formatted list of the existing Datasets:

```
{
  "name": "continuity.user.purchases",
  "type": "com.continuity.api.dataset.lib.ObjectStore",
  "properties": {
    "schema": "...",
    "type": "..."
  },
  "datasetSpecs": {
    ...
  }
}
```

Creating a Dataset

You can create a Dataset by issuing an HTTP PUT request to the URL:

```
PUT <base-url>/data/datasets/<dataset-name>
```

with JSON-formatted name of the dataset type and properties in a body:

```
{
  "typeName": "<type-name>",
  "properties": {<properties>}
}
```

Parameter	Description
<dataset-name>	Name of the new Dataset
<type-name>	Type of the new Dataset
<properties>	Dataset properties, map of String to String.

HTTP Responses

Status Codes	Description
200 OK	Requested Dataset was successfully created
404 Not Found	Requested Dataset type was not found
409 Conflict	Dataset with the same name already exists

Example

HTTP Request	PUT <base-url>/data/datasets/mydataset
Body	<pre>{ "typeName": "com.continuity.api.dataset.table.Table", "properties": { "ttl": "3600000" } }</pre>
Description	Creates a Dataset named "mydataset" of the type "table" and time-to-live property set to 1 hour

Deleting a Dataset

You can delete a Dataset by issuing an HTTP DELETE request to the URL:

```
DELETE <base-url>/data/datasets/<dataset-name>
```

HTTP Responses

Status Codes	Description
200 OK	Dataset was successfully deleted
404 Not Found	Dataset named <dataset-name> could not be found

Example

HTTP Request	DELETE <base-url>/data/datasets/mydataset
Description	Deletes the Dataset named "mydataset"

Deleting all Datasets

You can delete all Datasets by issuing an HTTP DELETE request to the URL:

```
DELETE <base-url>/data/unrecoverable/datasets
```

HTTP Responses

Status Codes	Description
200 OK	All Datasets were successfully deleted

Truncating a Dataset

You can truncate a Dataset by issuing an HTTP POST request to the URL:

```
POST <base-url>/data/datasets/<dataset-name>/admin/truncate
```

This will clear the existing data from the Dataset. This cannot be undone.

HTTP Responses

Status Codes	Description
200 OK	Dataset was successfully truncated

Data HTTP API (Deprecated)

The Data API allows you to interact with Continuity Reactor Tables (the core Datasets) through HTTP. You can create Tables, truncate Tables, and read, write, modify, or delete data.

For Datasets other than Tables, you can truncate the Dataset using this API.

Creating a new Table

To create a new table, issue an HTTP PUT method to the URL:

```
PUT <base-url>/tables/<table-name>
```

Parameter	Description
<table-name>	Name of the Table to be created

HTTP Responses

Status Codes	Description
200 OK	The event was successfully received and the Table was either created or already exists
409 Conflict	A Dataset of a different type already exists with the given name

Example

HTTP Method	PUT <base-url>/tables/streams/mytable
Description	Create a new Table named <i>mytable</i>

Comments

This will create a Table with the name given by <table-name>. Table names should only contain ASCII letters, digits and hyphens. If a Table with the same name already exists, no error is returned, and the existing Table remains in place.

However, if a Dataset of a different type exists with the same name—for example, a key/value Table or KeyValueTable—this call will return a 409 Conflict error.

Writing Data to a Table

To write to a table, send an HTTP PUT method to the table's URI:

```
PUT <base-url>/tables/<table-name>/rows/<row-key>
```

Parameter	Description
<table-name>	Name of the Table to be written to
<row-key>	Row identifier

HTTP Responses

Status Codes	Description
200 OK	The event was successfully received and the Table was successfully written to
400 Bad Request	The JSON String map is not well-formed or cannot be parsed as a map from String to String
404 Not Found	A Table with the given name does not exist

Example

HTTP Method	PUT <base-url>/tables/mytable/rows/status
Description	Write to the existing Table named <i>mytable</i> in a row identified as <i>status</i>

Comments

In the body of the request, you must specify the columns and values that you want to write to the Table as a JSON String map. For example:

```
{ "x": "y", "y": "a", "z": "1" }
```

This writes three columns named *x*, *y*, and *z* with values *y*, *a*, and *1*, respectively.

Reading Data from a Table

To read data from a Table, address the row that you want to read directly in an HTTP GET method to the table's URI:

```
GET <base-url>/tables/<table-name>/rows/<row-key>[?<column-identifier>]
```

Parameter	Description
<table-name>	Name of the Table to be read from
<row-key>	Row identifier
<column-identifiers>	An optional combination of attributes and values such as: start=<column-id> stop=<column-id> columns=<column-id>,<column-id>

HTTP Responses

Status Codes	Description
200 OK	The event was successfully received and the Table was successfully read from
400 Bad Request	The column list is not well-formed or cannot be parsed
404 Not Found	A Table with the given name does not exist

Example

HTTP Method	GET <base-url>/tables/mytable/rows/status
Description	Read from an existing Table named <i>mytable</i> , a row identified as <i>status</i>

Comments

The response will be a JSON String representing a map from column name to value. For example, reading the row that was written in the [Writing Data to a Table](#), the response is:

```
{ "x": "y", "y": "a", "z": "1" }
```

If you are only interested in selected columns, you can specify a list of columns explicitly or give a range of columns.

For example:

To return only columns *x* and *y*:

```
GET ... /rows/<row-key>?columns=x,y
```

To return all columns equal to or greater than (inclusive) *c5*:

```
GET ... /rows/<row-key>?start=c5
```

To return all columns less than (exclusive, not including) *c5*:

```
GET ... /rows/<row-key>?stop=c5
```

To return all columns equal to or greater than (inclusive) *c2* and less than (exclusive, not including) *c5*:

```
GET ... /rows/<row-key>?start=c2&stop=c5
```

Increment Data in a Table

You can perform an atomic increment of cells of a Table's row, and receive back the incremented values, by issue an HTTP POST method to the row's URL:

```
POST <base-url>/tables/<table-name>/rows/<row-key>/increment
```

Parameter	Description
<table-name>	Name of the Table to be read from
<row-key>	Row identifier of row to be read

HTTP Responses

Status Codes	Description
200 OK	The event successfully incremented the row of the Table
400 Bad Request	The JSON String is not well-formed; or cannot be parsed as a map from a String to a Long; or one of the existing column values is not an 8-byte long value
404 Not Found	A table with the given name does not exist

Example

HTTP Method	POST <base-url>/streams/mytable/rows/status/increment
Description	To increment the columns of <i>mytable</i> , in a row identified as <i>status</i> , by 1

Comments

In the body of the method, you must specify the columns and values that you want to increment them by as a JSON map from Strings to Long numbers, such as:

```
{ "x": 1, "y": 7 }
```

This HTTP call has the same effect as the corresponding Java Table Increment method.

If successful, the response contains a JSON String map from the column keys to the incremented values.

For example, if the existing value of column *x* was 4, and column *y* did not exist, then the response would be:

```
{ "x":5, "y":7 }
```

Column *y* is newly created.

Delete Data from a Table

To delete from a table, submit an HTTP DELETE method:

```
DELETE <base-url>/tables/<table-name>/rows/<row-key>[?<column-identifier>]
```

Parameter	Description
<table-name>	Name of the Table to be deleted from
<row-key>	Row identifier
<column-identifiers>	An optional combination of attributes and values such as: <div>start=<column-id> stop=<column-id> columns=<column-id>,<column-id></div>

HTTP Responses

Status Codes	Description
200 OK	The event successfully deleted the data of the Table
404 Not Found	A table with the given name does not exist

Example

HTTP Method	DELETE <base-url>/tables/mytable/rows/status
Description	Deletes from an existing Table named <i>mytable</i> , a row identified as <i>status</i>

Comments

Similarly to [Reading Data from a Table](#), explicitly list the columns that you want to delete by adding a parameter of the form `?columns=<column-key>,...>`. See the examples under [Reading Data from a Table](#).

Deleting Data from a Dataset

To clear a Dataset of all data, submit an HTTP POST request:

```
POST <base-url>/datasets/<dataset-name>/truncate
```

Parameter	Description
<dataset-name>	Name of the Dataset to be truncated

HTTP Responses

Status Codes	Description
200 OK	The event successfully deleted the data of the Dataset
404 Not Found	A Dataset with the given name does not exist

Example

HTTP Method	POST <base-url>/datasets/mydataset/truncate
Description	Delete all of the data from an existing Dataset named <i>mydataset</i>

Comments

Note that this works not only for Tables but with other Datasets, including user-defined Custom Datasets.

Encoding of Keys and Values

The URLs and JSON bodies of your HTTP requests contain row keys, column keys and values, all of which are binary byte Arrays in the Java API.

You need to encode these binary keys and values as Strings in the URL and the JSON body (the exception is the [Increment Data in a Table](#) method, which always interprets values as long integers).

The encoding parameter of the URL specifies the encoding used in both the URL and the JSON body.

For example, if you append a parameter `encoding=hex` to the request URL, then all keys and values are interpreted as hexadecimal strings, and the returned JSON from read requests also has keys and values encoded as hexadecimal string.

Be aware that the same encoding applies to all keys and values involved in a request.

For example, suppose you incremented table *counters*, row *a*, column *x* by 42:

```
POST <base-url>/tables/counters/rows/a/increment {"x":42}
```

Now the value of column *x* is the 8-byte number 42. If you query for the value of this column:

```
GET <base-url>/tables/counters/rows/a?columns=x
```

The returned JSON String map will contain a non-printable string for the value of column *x*:

```
{"x": "\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000*"} 
```

Note the Unicode escapes in the string, and the asterisk at the end (which is the character at code point 42).

To make this legible, you can specify hexadecimal notation in your request; that will require that you also encode the row key (*a*, encoded as *61*) and the column key (*x*, encoded as *78*) in your request as hexadecimal:

```
GET <base-url>/tables/counters/rows/61?columns=78&encoding=hex
```

The response now contains both the column key and the value as hexadecimal strings:

```
{"78": "000000000000002a"}
```

The supported encodings are:

Encoding	Description
<code>encoding=ascii</code>	Only ASCII characters are supported and are mapped to bytes one-to-one (Default)
<code>encoding=hex</code>	Hexadecimal strings. Example: the ASCII string <code>a:b</code> is represented as <code>613A62</code>
<code>encoding=url</code>	URL encoding (also known as %-encoding or percent-encoding). URL-safe characters use ASCII-encoding, while other bytes values are escaped using a % sign. Example: the hexadecimal value <code>613A62</code> (ASCII string <code>a:b</code>) is represented as the string <code>a%3Ab</code> .
<code>encoding=base64</code>	URL-safe Base-64 encoding without padding. For more information, see Internet RFC 2045 . Example: the hexadecimal value <code>613A62</code> is represented as the string <code>YTpi</code> .

If you specify an encoding that is not supported, or you specify keys or values that cannot be decoded using that encoding, the request will return HTTP code 400 *Bad Request*.

Counter Values

Your Table values may frequently be counters (numbers), whereas the row and column keys might not be numbers.

In such cases, it is more convenient to represent your Table values as numeric strings, by specifying `counter=true`. For example:

```
GET <base-url>/tables/counters/rows/a?columns=x&counter=true
```

The response now contains the column key as text and the row value as a numeric string:

```
{ "x" : "42" }
```

Note that you can also specify the `counter=true` parameter when writing to a Table. This allows you to specify values as numeric strings while using a different encoding for row and column keys.

Query HTTP API

This interface supports submitting SQL queries over Datasets. Executing a query is asynchronous:

- first, **submit** the query;
- then poll for the query's **status** until it is finished;
- once finished, retrieve the **result schema** and the **results**;
- finally, **close the query** to free the resources that it holds.

Submitting a Query

To submit a SQL query, post the query string to the `queries` URL:

```
POST <base-url>/data/queries
```

The body of the request must contain a JSON string of the form:

```
{
  "query": "<SQL-query-string>"
}
```

where `<SQL-query-string>` is the actual SQL query.

HTTP Responses

Status Codes	Description
200 OK	The query execution was successfully initiated, and the body will contain the query-handle used to identify the query in subsequent requests
400 Bad Request	The query is not well-formed or contains an error, such as a nonexistent table name.

Comments

If the query execution was successfully initiated, the body will contain a handle used to identify the query in subsequent requests:

```
{ "handle": "<query-handle>" }
```

Status of a Query

The status of a query is obtained using a HTTP GET request to the query's URL:

```
GET <base-url>/data/queries/<query-handle>
```

HTTP Responses

Status Codes	Description
200 OK	The query exists and the body contains its status
404 Not Found	The query handle does not match any current query.

Comments

If the query exists, the body will contain the status of its execution and whether the query has a results set:

```
{
  "status": "<status-code>",
  "hasResults": <boolean>
}
```

Status codes include `INITIALIZED`, `RUNNING`, `FINISHED`, `CANCELED`, `CLOSED`, `ERROR`, `UNKNOWN`, and `PENDING`.

Obtaining the Result Schema

If the query's status is `FINISHED` and it has results, you can obtain the schema of the results:

```
GET <base-url>/data/queries/<query-handle>/schema
```

HTTP Responses

Status Codes	Description
200 OK	The query was successfully received and the query schema was returned in the body
404 Not Found	The query handle does not match any current query

Comments

The query's result schema is returned in a JSON body as a list of columns, each given by its name, type and position; if the query has no result set, this list is empty:

```
[
  { "name": "<name>", "type": "<type>", "position": <int> },
  ...
]
```

The type of each column is a data type as defined in the [Hive language manual](#).

Retrieving Query Results

Query results can be retrieved in batches after the query is finished, optionally specifying the batch size in the body of the request:

```
POST <base-url>/data/queries/<query-handle>/next
```

The body of the request can contain a JSON string specifying the batch size:

```
{
  "size":<int>
}
```

If the batch size is not specified, it defaults to 20.

HTTP Responses

Status Codes	Description
200 OK	The event was successfully received and the result of the query was returned in the body
404 Not Found	The query handle does not match any current query

Comments

The results are returned in a JSON body as a list of columns, each given as a structure containing a list of column values.:

```
[
  { "columns": [ <value_1>, <value_2>, ..., ] },
  ...
]
```

The value at each position has the type that was returned in the result schema for that position. For example, if the returned type was `INT`, then the value will be an integer literal, whereas for `STRING` or `VARCHAR` the value will be a string literal.

Repeat the query to retrieve subsequent results. If all results of the query have already been retrieved, then the returned list is empty.

Closing a Query

The query can be closed by issuing an HTTP DELETE against its URL:

```
DELETE <base-url>/data/queries/<query-handle>
```

This frees all resources that are held by this query.

HTTP Responses

Status Codes	Description
200 OK	The query was closed
400 Bad Request	The query was not in a state that could be closed; either wait until it is finished, or cancel it
404 Not Found	The query handle does not match any current query

Canceling a Query

Execution of a query can be canceled before it is finished with an HTTP POST:

```
POST <base-url>/data/queries/<query-handle>/cancel
```

After this, the query can only be closed.

Note: This operation is not supported with CDH 4.x distributions. It will return a 500 Internal Server Error status code.

HTTP Responses

Status Codes	Description
200 OK	The query was canceled
400 Bad Request	The query was not in a state that can be canceled
404 Not Found	The query handle does not match any current query

Procedure HTTP API

This interface supports sending calls to the methods of an Application's Procedures. See the [Reactor Client HTTP API](#) for how to control the life cycle of Procedures.

Executing Procedures

To call a method in an Application's Procedure, send the method name as part of the request URL and the arguments as a JSON string in the body of the request.

The request is an HTTP POST:

```
POST <base-url>/apps/<app-id>/procedures/<procedure-id>/methods/<method-id>
```

Parameter	Description
<app-id>	Name of the Application being called
<procedure-id>	Name of the Procedure being called
<method-id>	Name of the method being called

HTTP Responses

Status Codes	Description
200 OK	The event successfully called the method, and the body contains the results
400 Bad Request	The Application, Procedure and method exist, but the arguments are not as expected
404 Not Found	The Application, Procedure, or method does not exist

Example

HTTP Method	POST <base-url>/apps/WordCount/procedures/RetrieveCounts/methods/getCount
Description	Call the <code>getCount()</code> method of the <i>RetrieveCounts</i> Procedure in the <i>WordCount</i> Application with the arguments as a JSON string in the body: <div><pre>{ "word": "a" }</pre></div>

Reactor Client HTTP API

Use the Reactor Client HTTP API to deploy or delete Applications and manage the life cycle of Flows, Procedures, MapReduce jobs, Workflows, and Custom Services.

Deploy an Application

To deploy an Application from your local file system, submit an HTTP POST request:

```
POST <base-url>/apps
```

with the name of the JAR file as a header:

```
X-Archive-Name: <JAR filename>
```

and its content as the body of the request:

```
<JAR binary content>
```

Invoke the same command to update an Application to a newer version. However, be sure to stop all of its Flows, Procedures and MapReduce jobs before updating the Application.

To list all of the deployed applications, issue an HTTP GET request:

```
GET <base-url>/apps
```

This will return a JSON String map that lists each Application with its name and description.

Delete an Application

To delete an Application together with all of its Flows, Procedures and MapReduce jobs, submit an HTTP DELETE:

```
DELETE <base-url>/apps/<application-name>
```

Parameter	Description
<application-name>	Name of the Application to be deleted

Note that the <application-name> in this URL is the name of the Application as configured by the Application Specification, and not necessarily the same as the name of the JAR file that was used to deploy the Application. Note also that this does not delete the Streams and Datasets associated with the Application because they belong to your account, not the Application.

Start, Stop, Status, and Runtime Arguments

After an Application is deployed, you can start and stop its Flows, Procedures, MapReduce jobs, Workflows, and Custom Services, and query for their status using HTTP POST and GET methods:

```
POST <base-url>/apps/<app-id>/<element-type>/<element-id>/<operation>
GET  <base-url>/apps/<app-id>/<element-type>/<element-id>/status
```

Parameter	Description
<app-id>	Name of the Application being called
<element-type>	One of flows, procedures, mapreduce, workflows Or services
<element-id>	Name of the element (<i>Flow, Procedure, MapReduce, Workflow, or Custom Service</i>) being called
<operation>	One of start Or stop

Examples

	Example / Description
HTTP Method	POST <base-url>/apps/HelloWorld/flows/WhoFlow/start
	Start a Flow <i>WhoFlow</i> in the Application <i>HelloWorld</i>
HTTP Method	POST <base-url>/apps/Count/procedures/GetCounts/stop
	Stop the Procedure <i>GetCounts</i> in the Application <i>Count</i>
HTTP Method	GET <base-url>/apps/HelloWorld/flows/WhoFlow/status
	Get the status of the Flow <i>WhoFlow</i> in the Application <i>HelloWorld</i>

When starting an element, you can optionally specify runtime arguments as a JSON map in the request body:

```
POST <base-url>/apps/HelloWorld/flows/WhoFlow/start
```

with the arguments as a JSON string in the body:

```
{ "foo": "bar", "this": "that" }
```

The Continuuity Reactor will use these these runtime arguments only for this single invocation of the element. To save the runtime arguments so that the Reactor will use them every time you start the element, issue an HTTP PUT with the parameter `runtimeargs`:

```
PUT <base-url>/apps/HelloWorld/flows/WhoFlow/runtimeargs
```

with the arguments as a JSON string in the body:

```
{ "foo": "bar", "this": "that" }
```

To retrieve the runtime arguments saved for an Application's element, issue an HTTP GET request to the element's URL using the same parameter `runtimeargs`:

```
GET <base-url>/apps/HelloWorld/flows/WhoFlow/runtimeargs
```

This will return the saved runtime arguments in JSON format.

Container Information

To find out the address of an element's container host and the container's debug port, you can query the Reactor for a Procedure, Flow or Service's live info via an HTTP GET method:

```
GET <base-url>/apps/<app-id>/<element-type>/<element-id>/live-info
```

Parameter	Description
<app-id>	Name of the Application being called
<element-type>	One of <i>flows</i> , <i>procedures</i> or <i>services</i>
<element-id>	Name of the element (<i>Flow</i> , <i>Procedure</i> or <i>Custom Service</i>)

Example:

```
GET <base-url>/apps/WordCount/flows/WordCounter/live-info
```

The response is formatted in JSON; an example of this is shown in:

[Continuity Reactor Testing and Debugging Guide](#)

To find out the address of a Service's container host and the container's debug port, you can query the Reactor for the live info of a Service's Twill Runnable via an HTTP GET method:

```
GET <base-url>/apps/<app-id>/services/<service-id>/runnables/<runnable-id>/live-info
```

Parameter	Description
<app-id>	Name of the Application being called
<service-id>	Name of the Custom Service being called
<runnable-id>	Name of the Twill Runnable being called

Example:

```
GET <base-url>/apps/WordCount/services/CounterService/runnables/CountRunnable/live-info
```

The response is formatted in JSON.

Scale

Scaling Flowlets

You can query and set the number of instances executing a given Flowlet by using the `instances` parameter with HTTP GET and PUT methods:

```
GET <base-url>/apps/<app-id>/flows/<flow-id>/flowlets/<flowlet-id>/instances
PUT <base-url>/apps/<app-id>/flows/<flow-id>/flowlets/<flowlet-id>/instances
```

with the arguments as a JSON string in the body:

```
{ "instances" : <quantity> }
```

Parameter	Description
<app-id>	Name of the Application being called
<flow-id>	Name of the Flow
<flowlet-id>	Name of the Flowlet
<quantity>	Number of instances to be used

Examples

HTTP Method	GET <base-url>/apps/HelloWorld/flows/WhoFlow/flowlets/saver/ instances
Description	Find out the number of instances of the Flowlet <i>saver</i> in the Flow <i>WhoFlow</i> of the Application <i>HelloWorld</i>
HTTP Method	PUT <base-url>/apps/HelloWorld/flows/WhoFlow/flowlets/saver/ instances with the arguments as a JSON string in the body: <pre>{ "instances" : 2 }</pre>
Description	Change the number of instances of the Flowlet <i>saver</i> in the Flow <i>WhoFlow</i> of the Application <i>HelloWorld</i>

Scaling Procedures

In a similar way to [Scaling Flowlets](#), you can query or change the number of instances of a Procedure by using the `instances` parameter with HTTP GET and PUT methods:

```
GET <base-url>/apps/<app-id>/procedures/<procedure-id>/instances
PUT <base-url>/apps/<app-id>/procedures/<procedure-id>/instances
```

with the arguments as a JSON string in the body:

```
{ "instances" : <quantity> }
```

Parameter	Description
<app-id>	Name of the Application
<procedure-id>	Name of the Procedure
<quantity>	Number of instances to be used

Example

HTTP Method	GET <base-url>/apps/HelloWorld/procedures/Greeting/instances instances
Description	Find out the number of instances of the Procedure <i>Greeting</i> in the Application <i>HelloWorld</i>

Scaling Services

You can query or change the number of instances of a Service's runnable by using the `instances` parameter with HTTP GET and PUT methods:

```
GET <base-url>/apps/<app-id>/services/<service-id>/runnables/<runnable-id>/instances
PUT <base-url>/apps/<app-id>/services/<service-id>/runnables/<runnable-id>/instances
```

with the arguments as a JSON string in the body:

```
{ "instances" : <quantity> }
```

Parameter	Description
<app-id>	Name of the Application
<service-id>	Name of the Service
<runnable-id>	Name of the Twill Runnable
<quantity>	Number of instances to be used

Example

HTTP Method	GET <base-url>/apps/HelloWorld/services/WhoService/runnables/WhoRunnable/instances
Description	Retrieve the number of instances of the Twill Runnable <i>WhoRunnable</i> of the Service <i>WhoService</i>

Run History and Schedule

To see the history of all runs of selected elements (Flows, Procedures, MapReduce jobs, Workflows, and Services), issue an HTTP GET to the element's URL with the `history` parameter. This will return a JSON list of all completed runs, each with a start time, end time and termination status:

```
GET <base-url>/apps/<app-id>/<element>/<element-id>/history
```

Parameter	Description
<app-id>	Name of the Application
<element-type>	One of flows, procedures, mapreduce, workflows Or services
<element-id>	Name of the element

Example

HTTP Method	GET <base-url>/apps/HelloWorld/flows/WhoFlow/history
Description	Retrieve the history of the Flow <i>WhoFlow</i> of the Application <i>HelloWorld</i>
Returns	{ "runid": "...", "start": 1382567447, "end": 1382567492, "status": "STOPPED" }, { "runid": "...", "start": 1382567383, "end": 1382567397, "status": "STOPPED" }

The *runid* field is a UUID that uniquely identifies a run within the Continuuity Reactor, with the start and end times in seconds since the start of the Epoch (midnight 1/1/1970).

For Services, you can retrieve the history of a Twill Service using:

```
GET <base-url>/apps/<app-id>/services/<service-id>/history
```

Example

HTTP Method	GET <base-url>/apps/HelloWorld/services/WhoService/history
Description	Retrieve the history of the Service <i>WhoService</i> of the Application <i>HelloWorld</i>
Returns	{ "runid": "...", "start": 1382567447, "end": 1382567492, "status": "STOPPED" }, { "runid": "...", "start": 1382567383, "end": 1382567397, "status": "STOPPED" }

For Workflows, you can also retrieve:

- the schedules defined for a workflow (using the parameter `schedules`):

```
GET <base-url>/apps/<app-id>/workflows/<workflow-id>/schedules
```

- the next time that the workflow is scheduled to run (using the parameter `nextruntime`):

```
GET <base-url>/apps/<app-id>/workflows/<workflow-id>/nextruntime
```

Promote

To promote an Application from your local Continuity Reactor to your Sandbox Continuity Reactor, send a POST request with the host name of your Sandbox in the request body. You must include the API key for the Sandbox in the request header.

Example

Promote the Application *HelloWorld* from your Local Reactor to your Sandbox:

```
POST <base-url>/apps/HelloWorld/promote
```

with the API Key in the header:

```
X-Continuity-APIKey: <api-key> {"hostname":"<sandbox>.continuity.net"}
```

Parameter	Description
<api-key>	Continuity Reactor API key, obtained from an account at Continuity Accounts
<sandbox>	Sandbox located on continuity.net

Logging HTTP API

Downloading Logs

You can download the logs that are emitted by any of the *Flows*, *Procedures*, or *MapReduce* jobs running in the Continuity Reactor. To do that, send an HTTP GET request:

```
GET <base-url>/apps/<app-id>/<element-type>/<element-id>/logs?start=<ts>&stop=<ts>
```

Parameter	Description
<app-id>	Name of the Application being called
<element-type>	One of flows, procedures, Or mapreduce
<element-id>	Name of the element (<i>Flow</i> , <i>Procedure</i> , <i>MapReduce</i> job) being called
<ts>	<i>Start</i> and <i>stop</i> times, given as seconds since the start of the Epoch.

Example

HTTP Method	GET <base-url>/apps/CountTokens/flows/CountTokensFlow/ logs?start=1382576400&stop=1382576700
Description	Return the logs for all the events from the Flow <i>CountTokensFlow</i> of the <i>CountTokens</i> Application, beginning Thu, 24 Oct 2013 01:00:00 GMT and ending Thu, 24 Oct 2013 01:05:00 GMT (five minutes later)

You can download the logs that are emitted by the Twill Runnable of a Service in a Reactor Application by sending an HTTP GET request:

```
GET <base-url>/apps/<app-id>/services/<service-id>/runnables/<runnable-id>/logs?start=<ts>&stop=<ts>
```

Parameter	Description
<app-id>	Name of the Application being called
<service-id>	Name of the Service being called
<runnable-id>	Name of the Twill Runnable being called
<ts>	<i>Start</i> and <i>stop</i> times, given as seconds since the start of the Epoch.

Example

HTTP Method	GET <base-url>/apps/CountTokens/services/CountTokensService/runnables/CountTokensRunnable/ logs?start=1382576400&stop=1382576700
Description	Return the logs for all the events of the Runnable <i>CountTokensRunnable</i> from the Service <i>CountTokensService</i> of the <i>CountTokens</i> Application, beginning Thu, 24 Oct 2013 01:00:00 GMT and ending Thu, 24 Oct 2013 01:05:00 GMT (five minutes later)

Comments

The output is formatted as HTML-embeddable text; that is, characters that have a special meaning in HTML will be escaped. A line of the log may look like this:

```
2013-10-23 18:03:09,793 - INFO [FlowletProcessDriver-source-0-  
executor:c.c.e.c.StreamSource@-1] - source: Emitting line: this is an & character
```

Note how the context of the log line shows the name of the Flowlet (*source*), its instance number (0) as well as the original line in the Application code. The character & is escaped as &; if you don't desire this escaping, you can turn it off by adding the parameter &escape=false to the request URL.

Metrics HTTP API

As Applications process data, the Continuity Reactor collects metrics about the Application's behavior and performance. Some of these metrics are the same for every Application—how many events are processed, how many data operations are performed, etc.—and are thus called system or Reactor metrics.

Other metrics are user-defined and differ from Application to Application. For details on how to add metrics to your Application, see the section on User-Defined Metrics in the the Developer Guide:

[Continuity Reactor Operations Guide](#)

Metrics Requests

The general form of a metrics request is:

```
GET <base-url>/metrics/<scope>/<context>/<metric>?<time-range>
```

Parameter	Description
<scope>	Either <code>reactor</code> (system metrics) or <code>user</code> (user-defined metrics)
<context>	Hierarchy of context; see Available Contexts
<metric>	Metric being queried; see Available Metrics
<time-range>	A Time Range or <code>aggregate=true</code> for all since the Application was deployed

Examples

HTTP Method	GET <base-url>/metrics/reactor/apps/HelloWorld/flows/WhoFlow/flowlets/saver/process.bytes?aggregate=true
Description	Using a <i>System</i> metric, <i>process.bytes</i>
HTTP Method	GET <base-url>/metrics/user/apps/HelloWorld/flows/WhoFlow/flowlets/saver/names.bytes?aggregate=true
Description	Using a <i>User-Defined</i> metric, <i>names.bytes</i>
HTTP Method	GET <base-url>/metrics/user/apps/HelloWorld/services/WhoService/runnables/WhoRun/names.bytes?aggregate=true
Description	Using a <i>User-Defined</i> metric, <i>names.bytes</i> in a Service's Twill Runnable

Comments

The scope must be either `reactor` for system metrics or `user` for user-defined metrics.

System metrics are either Application metrics (about Applications and their Flows, Procedures, MapReduce and Workflows) or they are Data metrics (relating to Streams or Datasets).

User metrics are always in the Application context.

For example, to retrieve the number of input data objects ("events") processed by a Flowlet named *splitter*, in the Flow *CountRandomFlow* of the Application *CountRandom*, over the last 5 seconds, you can issue an HTTP GET method:

```
GET <base-url>/metrics/reactor/apps/CountRandom/flows/CountRandomFlow/flowlets/  
splitter/process.events?start=now-5s&count=5
```

This returns a JSON response that has one entry for every second in the requested time interval. It will have values only for the times where the metric was actually emitted (shown here "pretty-printed", unlike the actual responses):

```
HTTP/1.1 200 OK  
Content-Type: application/json  
{ "start":1382637108,"end":1382637112,"data":[  
  { "time":1382637108,"value":6868 },  
  { "time":1382637109,"value":6895 },  
  { "time":1382637110,"value":6856 },  
  { "time":1382637111,"value":6816 },  
  { "time":1382637112,"value":6765 } ] }
```

If you want the number of input objects processed across all Flowlets of a Flow, you address the metrics API at the Flow context:

```
GET <base-url>/metrics/reactor/apps/CountRandom/flows/  
CountRandomFlow/process.events?start=now-5s&count=5
```

Similarly, you can address the context of all flows of an Application, an entire Application, or the entire Reactor:

```
GET <base-url>/metrics/reactor/apps/CountRandom/  
flows/process.events?start=now-5s&count=5  
GET <base-url>/metrics/reactor/apps/CountRandom/  
process.events?start=now-5s&count=5  
GET <base-url>/metrics/reactor/process.events?start=now-5s&count=5
```

To request user-defined metrics instead of system metrics, specify `user` instead of `reactor` in the URL and specify the user-defined metric at the end of the request.

For example, to request a user-defined metric for the *HelloWorld* Application's *WhoFlow* Flow:

```
GET <base-url>/metrics/user/apps/HelloWorld/flows/  
WhoFlow/flowlets/saver/names.bytes?aggregate=true
```

To retrieve multiple metrics at once, instead of a GET, issue an HTTP POST, with a JSON list as the request body that enumerates the name and attributes for each metrics. For example:

```
POST <base-url>/metrics
```

with the arguments as a JSON string in the body:

```
Content-Type: application/json  
[ "/reactor/collect.events?aggregate=true",  
  "/reactor/apps/HelloWorld/process.events?start=1380323712&count=6000" ]
```

If the context of the requested metric or metric itself doesn't exist the system returns status 200 (OK) with JSON formed as per above description and with values being zeroes.

Time Range

The time range of a metric query can be specified in various ways:

Time Range	Description
<code>start=now-30s&end=now</code>	The last 30 seconds. The begin time is given in seconds relative to the current time. You can apply simple math, using <code>now</code> for the current time, <code>s</code> for seconds, <code>m</code> for minutes, <code>h</code> for hours and <code>d</code> for days. For example: <code>now-5d-12h</code> is 5 days and 12 hours ago.
<code>start=1385625600&end=1385629200</code>	From Thu, 28 Nov 2013 08:00:00 GMT to Thu, 28 Nov 2013 09:00:00 GMT, both given as since the start of the Epoch
<code>start=1385625600&count=3600</code>	The same as before, but with the count given as a number of seconds

Instead of getting the values for each second of a time range, you can also retrieve the aggregate of a metric over time. The following request will return the total number of input objects processed since the Application *CountRandom* was deployed, assuming that the Reactor has not been stopped or restarted (you cannot specify a time range for aggregates):

```
GET <base-url>/metrics/reactor/apps/CountRandom/process.events?aggregate=true
```

Available Contexts

The context of a metric is typically enclosed into a hierarchy of contexts. For example, the Flowlet context is enclosed in the Flow context, which in turn is enclosed in the Application context. A metric can always be queried (and aggregated) relative to any enclosing context. These are the available Application contexts of the Continuity Reactor:

System Metric	Context
One Flowlet of a Flow	/apps/<app-id>/flows/<flow-id>/flowlets/<flowlet-id>
All Flowlets of a Flow	/apps/<app-id>/flows/<flow-id>
All Flowlets of all Flows of an Application	/apps/<app-id>/flows
One Procedure	/apps/<app-id>/procedures/<procedure-id>
All Procedures of an Application	/apps/<app-id>/procedures
All Mappers of a MapReduce	/apps/<app-id>/mapreduce/<mapreduce-id>/mappers
All Reducers of a MapReduce	/apps/<app-id>/mapreduce/<mapreduce-id>/reducers
One MapReduce	/apps/<app-id>/mapreduce/<mapreduce-id>
All MapReduce of an Application	/apps/<app-id>/mapreduce
One Twill Runnable	/apps/<app-id>/services/<service-id>/runnables/<runnable-id>
One Service	/apps/<app-id>/services/<service-id>
All Services of an Application	/apps/<app-id>/services
All elements of an Application	/apps/<app-id>
All elements of all Applications	/

Stream metrics are only available at the Stream level and the only available context is:

Stream Metric	Context
A single Stream	/streams/<stream-id>

Dataset metrics are available at the Dataset level, but they can also be queried down to the Flowlet, Procedure, Mapper, or Reducer level:

Dataset Metric	Context
A single Dataset in the context of a single Flowlet	/datasets/<dataset-id>/apps/<app-id>/flows/<flow-id>/flowlets/<flowlet-id>
A single Dataset in the context of a single Flow	/datasets/<dataset-id>/apps/<app-id>/flows/<flow-id>
A single Dataset in the context of a specific Application	/datasets/<dataset-id>/<any application context>
A single Dataset across all Applications	/datasets/<dataset-id>
All Datasets across all Applications	/

Available Metrics

For Continuity Reactor metrics, the available metrics depend on the context. User-defined metrics will be available at whatever context that they are emitted from.

These metrics are available in the Flowlet context:

Flowlet Metric	Description
<code>process.busyness</code>	A number from 0 to 100 indicating how “busy” the Flowlet is; note that you cannot aggregate over this metric
<code>process.errors</code>	Number of errors while processing
<code>process.events.processed</code>	Number of events/data objects processed
<code>process.events.in</code>	Number of events read in by the Flowlet
<code>process.events.out</code>	Number of events emitted by the Flowlet
<code>store.bytes</code>	Number of bytes written to Datasets
<code>store.ops</code>	Operations (writes and read) performed on Datasets
<code>store.reads</code>	Read operations performed on Datasets
<code>store.writes</code>	Write operations performed on Datasets

These metrics are available in the Mappers and Reducers context:

Mappers and Reducers Metric	Description
<code>process.completion</code>	A number from 0 to 100 indicating the progress of the Map or Reduce phase
<code>process.entries.in</code>	Number of entries read in by the Map or Reduce phase
<code>process.entries.out</code>	Number of entries written out by the Map or Reduce phase

These metrics are available in the Procedures context:

Procedures Metric	Description
<code>query.requests</code>	Number of requests made to the Procedure
<code>query.failures</code>	Number of failures seen by the Procedure

These metrics are available in the Streams context:

Streams Metric	Description
<code>collect.events</code>	Number of events collected by the Stream
<code>collect.bytes</code>	Number of bytes collected by the Stream

These metrics are available in the Datasets context:

Datasets Metric	Description
<code>store.bytes</code>	Number of bytes written
<code>store.ops</code>	Operations (reads and writes) performed
<code>store.reads</code>	Read operations performed
<code>store.writes</code>	Write operations performed

Monitor HTTP API

Reactor internally uses a variety of System Services that are critical to its functionality. This section describes the REST APIs that can be used to see into System Services.

Details of All Available System Services

For the detailed information of all available System Services, use:

```
GET <base-url>/system/services
```

HTTP Responses

Status Codes	Description
200 OK	The event successfully called the method, and the body contains the results

Checking Status of All Reactor System Services

To check the status of all the System Services, use:

```
GET <base-url>/system/services/status
```

HTTP Responses

Status Codes	Description
200 OK	The event successfully called the method, and the body contains the results

Checking Status of a Specific Reactor System Service

To check the status of a specific System Service, use:

```
GET <base-url>/system/services/<service-name>/status
```

The status of these Reactor System Services can be checked:

Service	Service-Name	Description of the Service
Metrics	metrics	Service that handles metrics related HTTP requests
Transaction	transaction	Service handling transactions
Streams	streams	Service handling Stream management
App Fabric	appfabric	Service handling Application Fabric requests
Log Saver	log.saver	Service aggregating all system and application logs
Metrics Processor	metrics.processor	Service that aggregates all system and application metrics
Dataset Executor	dataset.executor	Service that handles all data-related HTTP requests
Explore Service	explore.service	Service that handles all HTTP requests for ad-hoc data exploration

Note that the Service status checks are more useful when the Reactor is running in a distributed cluster mode.

Example

HTTP Method	GET <base-url>/system/services/metrics/status
Description	Returns the status of the Metrics Service

HTTP Responses

Status Codes	Description
200 OK	The service is up and running
404 Not Found	The service is either not running or not found

Scaling System Services

The number of instances for system services can be queried and changed by using these commands:

```
GET <base-url>/system/services/<service-name>/instances
PUT <base-url>/system/services/<service-name>/instances
```

with the arguments as a JSON string in the body:

```
{ "instances" : <quantity> }
```

Parameter	Description
<system-name>	Name of the system service
<quantity>	Number of instances to be used

Examples

HTTP Method	GET <base-url>/system/services/metrics/instances instances
Description	Determine the number of instances being used for the metrics HTTP service
HTTP Method	PUT <base-url>/system/services/metrics/instances instances with the arguments as a JSON string in the body: <pre>{ "instances" : 2 }</pre>
Description	Sets the number of instances of the metrics HTTP service to 2