Continuuity Reactor HTTP REST API

Contents

Introduction

The Continuuity 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.
- **Data:** interacting with DataSets (currently limited to Tables).
- **Procedure:** sending queries to a Procedure.
- **Reactor:** deploying and managing Applications.
- **Logs:** retrieving Application logs.
- **Metrics:** retrieving metrics for system and user Applications (user-defined metrics).
- **Monitor:** checking the status of various Reactor services.

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 Continuuity Reactor using the HTTP interface.

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

All URLs referenced in this API have this base:

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

where `<gateway>` is the URL of the Continuuity 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 Continuuity Reactor

When you interact with a Sandboxed Continuuity Reactor, the Continuuity 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-Continuuity-ApiKey: <api-key>
```

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

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 *mystream*

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 *mystream*

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 `consumer-id` was returned
`404 Not Found`	The Stream does not exist

Example

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

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-Continuuity-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-Continuuity-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 `Consumer-ID` 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 *mystream*

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>
```

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.

Run at the command line:

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

for usage and documentation of options.

Data HTTP API

The Data API allows you to interact with Continuuity 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 *mytable*

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 *mytable* in a row identified as *status*

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 *mytable*, a row identified as *status*

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 *mytable*, in a row identified as *status*, 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: `start=<column-id> | stop=<column-id> | columns=<column-id>,<column-id>`

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 *mytable*, a row identified as *status*

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 *mydataset*

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*"}
```

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
`encoding=ascii`	Only ASCII characters are supported and are mapped to bytes one-to-one (Default)
`encoding=hex`	Hexadecimal strings. Example: the ASCII string `a:b` is represented as `613A62`
`encoding=url`	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 `613A62` (ASCII string `a:b`) is represented as the string `a%3Ab`.
`encoding=base64`	URL-safe Base-64 encoding without padding. For more information, see Internet RFC 2045. Example: the hexadecimal value `613A62` is represented as the string `YTpi`.

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.

Procedure HTTP API

This interface supports sending queries to the methods of an Application's 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 `getCount()` method of the *RetrieveCounts* Procedure in the *WordCount* Application with the arguments as a JSON string in the body: `{"word":"a"}`

Reactor Client HTTP API

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

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 elements and Workflows, 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`, or `workflows`
`<element-id>`	Name of the element (*Flow*, *Procedure*, *MapReduce*, or *WorkFlow*) being called
`<operation>`	One of `start` or `stop`

Examples

HTTP Method	`POST <base-url>/apps/HelloWorld/flows/WhoFlow/start`
Description	Start a Flow *WhoFlow* in the Application *HelloWorld*

HTTP Method	`POST <base-url>/apps/WordCount/procedures/RetrieveCounts/stop`
Description	Stop the Procedure *RetrieveCounts* in the Application *WordCount*

HTTP Method	`GET <base-url>/apps/HelloWorld/flows/WhoFlow/status`
Description	Get the status of the Flow *WhoFlow* in the Application *HelloWorld*

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 or Flow's live info via an HTTP GET method:

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

Parameter	Description
`<app-id>`	Name of the Application being called
`<element-type>`	One of either `flows` or `procedures`
`<element-id>`	Name of the element (*Flow* or *Procedure*)

Example:

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

The response is formatted in JSON; an example of this is shown in the Continuuity Reactor Testing and Debugging Guide.

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 *saver* in the Flow *WhoFlow* of the Application *HelloWorld*

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

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/flows/WhoFlow/procedure/saver/ instances`
Description	Find out the number of instances of the Procedure *saver* in the Flow *WhoFlow* of the Application *HelloWorld*

Run History and Schedule

To see the history of all runs of an element, issue an HTTP GET to the element's URL with `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>/flows/<flow-id>/history
```

Parameter	Description
`<app-id>`	Name of the Application
`<flow-id>`	Name of the Flow

Example

HTTP Method	`GET <base-url>/apps/HelloWorld/flows/WhoFlow/history`
Description	Retrieve the history of the Flow *WhoFlow* of the Application *HelloWorld*
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 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 Continuuity Reactor to your Sandbox Continuuity 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-Continuuity-ApiKey: <api-key> {"hostname":"<sandbox>.continuuity.net"}
```

Parameter	Description
`<api-key>`	Continuuity Reactor API key, obtained from an account at Continuuity Accounts
`<sandbox>`	Sandbox located on `continuuity.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 Continuuity 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 (*Flow*, *Procedure*, *MapReduce* job) being called
`<ts>`	*Start* and *stop* 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 *CountTokensFlow* of the *CountTokens* 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 &amp; 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 Continuuity 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 Continuuity 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 `reactor` (system metrics) or `user` (user-defined metrics)
`<context>`	Hierarchy of context; see Available Contexts
`<metric>`	Metric being queried; see Available Metrics
`<time-range>`	A Time Range or `aggregate=true` 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 *System* metric, *process.bytes*

HTTP Method	`GET <base-url>/metrics/user/apps/HelloWorld/flows/WhoFlow/flowlets/saver/names.bytes?aggregate=true`
Description	Using a *User-Defined* metric, *names.bytes*

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" ]
```

Time Range

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

Time Range	Description
`start=now-30s&end=now`	The last 30 seconds. The begin time is given in seconds relative to the current time. You can apply simple math, using `now` for the current time, `s` for seconds, `m` for minutes, `h` for hours and `d` for days. For example: `now-5d-12h` is 5 days and 12 hours ago.
`start=1385625600&` `end=1385629200`	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
`start=1385625600&` `count=3600`	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 Continuuity 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`
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 Continuuity 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
`process.busyness`	A number from 0 to 100 indicating how "busy" the Flowlet is; note that you cannot aggregate over this metric
`process.errors`	Number of errors while processing
`process.events.processed`	Number of events/data objects processed
`process.events.in`	Number of events read in by the Flowlet
`process.events.out`	Number of events emitted by the Flowlet
`store.bytes`	Number of bytes written to DataSets
`store.ops`	Operations (writes and read) performed on DataSets
`store.reads`	Read operations performed on DataSets
`store.writes`	Write operations performed on DataSets

These metrics are available in the Mappers and Reducers context:

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

These metrics are available in the Procedures context:

Procedures Metric	Description
`query.requests`	Number of requests made to the Procedure
`query.failures`	Number of failures seen by the Procedure

These metrics are available in the Streams context:

Streams Metric	Description
`collect.events`	Number of events collected by the Stream
`collect.bytes`	Number of bytes collected by the Stream

These metrics are available in the DataSets context:

DataSets Metric	Description
`store.bytes`	Number of bytes written
`store.ops`	Operations (reads and writes) performed
`store.reads`	Read operations performed
`store.writes`	Write operations performed

Monitor HTTP API

Reactor internally uses a variety of services that are critical to its functionality. Hence, the ability to check the health of those services can act as an useful initial debug step. This is faciliated by the Metrics HTTP API. To check the status of a service, send a HTTP GET request:

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

The status of these Reactor services can be checked.

Service Name	Service-Id	Description of the Service
`Metrics`	`metrics`	Service that handles metrics related requests
`Transaction`	`transaction`	Service handling transactions
`Streams`	`streams`	Service handling Stream management
`App Fabric`	`appfabric`	Service handling Application Fabric requests

Note that the service status checks are more useful when the Reactor is running in a distributed cluster mode and that some of the status checks may not work in the Local Reactor 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 not running or not found