Cask Data Application Platform HTTP RESTful API

Contents

Introduction

The Cask Data Application Platform (CDAP) 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
- **Query:** sending ad-hoc queries to CDAP Datasets
- **Procedure:** sending calls to a stored Procedure
- **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 CDAP 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 CDAP using the HTTP interface.

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

All URLs referenced in this API have this base URL:

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

where `<host>` is the URL of the CDAP server. The base URL is represented as:

```
<base-url>
```

For example:

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

means

```
PUT http://<host>: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.

Working with CDAP Security

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

```
Authorization: Bearer wohng8Xae7thahfohshahphaeNeeM5ie
```

For CDAP-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 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).

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>
```

In cases where it is acceptable to have some events lost if the system crashes, you can send events to a Stream asynchronously with higher throughput by sending an HTTP POST method to the `async` URL:

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

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

HTTP Responses

Status Codes	Description
`200 OK`	The event was successfully received and persisted
`202 ACCEPTED`	The event was successfully received but may not be persisted. Only the asynchronous endpoint will return this status code
`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

Reading events from an existing Stream is performed as an HTTP GET method to the URL:

```
GET <base-url>/streams/<stream-id>/events?start=<startTime>&end=<endTime>&limit=<limit>
```

Parameter	Description
`<stream-id>`	Name of an existing Stream
`<startTime>`	Optional timestamp in milliseconds to start reading events from (inclusive); default is 0
`<endTime>`	Optional timestamp in milliseconds for the last event to read (exclusive); default is the maximum timestamp (2^63)
`<limit>`	Optional maximum number of events to read; default is unlimited

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 there are no events that satisfy the request
`404 Not Found`	The Stream does not exist

The response body is an JSON array, with the Stream event objects as array elements:

```
[
  {"timestamp" : ... , "headers": { ... }, "body" : ... },
  {"timestamp" : ... , "headers": { ... }, "body" : ... }
]
```

Field	Description
`timestamp`	Timestamp in milliseconds of the Stream event at ingestion time
`headers`	A JSON map of all custom headers associated with the Stream event
`body`	A printable string representing the event body; non-printable bytes are hex escaped in the format `\x[hex-digit][hex-digit]`, e.g. `\x05`

Example

HTTP Method	`GET <base-url>/streams/mystream/events?limit=1`
Description	Read the initial event from an existing Stream named *mystream*
Response body	`[ {"timestamp" : 1407806944181, "headers" : { }, "body" : "Hello World"  } ]`

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 *mystream*

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:
	`{ "ttl" : 86400 }`

Description	Change the TTL property of the Stream named *mystream* to 1 day

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 CDAP Developer Guide Advanced Features, Datasets section

Listing all Datasets

You can list all Datasets in CDAP 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":"cdap.user.purchases",
   "type":"co.cask.cdap.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	`{"typeName":"co.cask.cdap.api.dataset.table.Table", "properties":{"ttl":"3600000"}}`
Description	Creates a Dataset named "mydataset" of the type "table" and time-to-live property set to 1 hour

Updating an Existing Dataset

You can update an existing Dataset's table and properties by issuing an HTTP PUT request to the URL:

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

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

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

Note: The Dataset must exist, and the instance and type passed must match with the existing Dataset.

Parameter	Description
`<dataset-name>`	Name of the existing Dataset
`<type-name>`	Type of the existing Dataset
`<properties>`	Dataset properties as a map of String to String

HTTP Responses

Status Codes	Description
`200 OK`	Requested Dataset was successfully updated
`404 Not Found`	Requested Dataset instance was not found
`409 Conflict`	Dataset Type provided for update is different from the existing Dataset Type

Example

HTTP Request	`PUT <base-url>/data/datasets/mydataset/properties`
Body	`{"typeName":"co.cask.cdap.api.dataset.table.Table", "properties":{"ttl":"7200000"}}`
Description	For the "mydataset" of type "Table", updates the Dataset and its time-to-live property to 2 hours

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 (see **Note** below) 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

Note: This operation will only be successful if the property `enable.unrecoverable.reset` in `cdap-site.xml` is set to `true`. Otherwise, this operation will return "403 Forbidden".

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

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/explore/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/explore/queries/<query-handle>/status
```

Parameter	Description
`<query-handle>`	Handle obtained when the query was submitted

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/explore/queries/<query-handle>/schema
```

Parameter	Description
`<query-handle>`	Handle obtained when the query was submitted

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/explore/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, the default is 20.

Parameter	Description
`<query-handle>`	Handle obtained when the query was submitted

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/explore/queries/<query-handle>
```

This frees all resources that are held by this query.

Parameter	Description
`<query-handle>`	Handle obtained when the query was submitted

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

List of Queries

To return a list of queries, use:

```
GET <base-url>/data/explore/queries?limit=<limit>&cursor=<cursor>&offset=<offset>
```

Parameter	Description
`<limit>`	Number of results to return in the response.; by default, 50 results will be returned
`<cursor>`	Specifies if the results returned should be in the forward or reverse direction by specifying `next` or `prev`
`<offset>`	Offset for pagination, returns the results that are greater than offset if the cursor is `next` or results that are less than offset if cursor is `prev`

Comments

The results are returned as a JSON array, with each element containing information about the query:

```
[
  {"timestamp":1407192465183,"statement":"SHOW TABLES","status":"FINISHED",
   "query_handle":"319d9438-903f-49b8-9fff-ac71cf5d173d","has_results":true,"is_active":false},
  ...
]
```

Download Query Results

To download the results of a query, use:

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

The results of the query are returned in CSV format.

Parameter	Description
`<query-handle>`	Handle obtained when the query was submitted or via a list of queries

Comments

The query results can be downloaded only once. The RESTful API will return a Status Code `409 Conflict` if results for the `query-handle` are attempted to be downloaded again.

HTTP Responses

Status Codes	Description
`200 OK`	The HTTP call was successful.
`404 Not Found`	The query handle does not match any current query.
`409 Conflict`	The query results was already downloaded.

Hive Table Schema

You can obtain the schema of the underlying Hive Table with:

```
GET <base-url>/data/explore/datasets/<dataset-name>/schema
```

Parameter	Description
`<dataset-name>`	Name of the Dataset whose schema is to be retrieved

Comments

The results are returned as a JSON Map, with `key` containing the column names of the underlying table and `value` containing the column types of the underlying table:

```
{
  "key": "array<tinyint>",
  "value": "array<tinyint>"
}
```

HTTP Responses

Status Codes	Description
`200 OK`	The HTTP call was successful.
`404 Not Found`	The dataset was not found.

Procedure HTTP API

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

Service HTTP API

This interface supports making requests to the methods of an Application's Services. See the CDAP Client HTTP API for how to control the life cycle of Services.

Requesting Service Methods

To make a request to a Service's method, send the method's path as part of the request URL along with any additional headers and body.

The request type is defined by the Service's method:

```
<REQUEST-TYPE> <base-url>/apps/<app-id>/services/<service-id>/methods/<method-id>
```

Parameter	Description
`<REQUEST-TYPE>`	One of GET, POST, PUT and DELETE. This is defined by the handler method.
`<app-id>`	Name of the Application being called
`<service-id>`	Name of the Service being called
`<method-id>`	Name of the method being called

HTTP Responses

Status Codes	Description
`503 Service Unavailable`	The service is unavailable. For example, it may not yet have been started.

Other responses are defined by the Service's method.

Example

HTTP Method	`GET <base-url>/apps/ExampleApplication/services/PingService/methods/ping`
Description	Make a request to the `ping` endpoint of the PingService in ExampleApplication.
Response status code	`200 OK`

CDAP Client HTTP API

Use the CDAP 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 (*Flow*, *Procedure*, *MapReduce*, *Workflow*, or *Custom Service*) being called
`<operation>`	One of `start` or `stop`

You can retrieve the status of multiple elements from different applications and element types using an HTTP POST method:

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

with a JSON array in the request body consisting of multiple JSON objects with these parameters:

Parameter	Description
`"appId"`	Name of the Application being called
`"programType"`	One of `flow`, `procedure`, `mapreduce`, `workflow` or `service`
`"programId"`	Name of the element (*Flow*, *Procedure*, *MapReduce*, *Workflow*, or *Custom Service*) being called

The response will be the same JSON array with additional parameters for each of the underlying JSON objects:

Parameter	Description
`"status"`	Maps to the status of an individual JSON object's queried element if the query is valid and the element was found.
`"statusCode"`	The status code from retrieving the status of an individual JSON object.
`"error"`	If an error, a description of why the status was not retrieved (the specified element was not found, the requested JSON object was missing a parameter, etc.)

Note that the `status` and `error` fields are mutually exclusive.

Examples

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

	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"}
```

CDAP will use these these runtime arguments only for this single invocation of the element. To save the runtime arguments so that CDAP 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.

To retrieve the status of multiple programs in different applications, use the HTTP POST command:

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

with the arguments for the different applications and programs as a JSON string map in the body, such as:

```
[{"appId":"MyApp1","programType":"Flow","programId":"MyFlow1"},
 {"appId":"MyApp1","programType":"Procedure","programId":"MyProc2"},
 {"appId":"MyApp3","programType":"Service","programId":"MySvc1"}]
```

If there was no procedure named `MyProc2` in the application `MyApp1`, a possible response could be:

```
[{"appId":"MyApp1","programType":"Flow","programId":"MyFlow1","status":"RUNNING","statusCode":200},
 {"appId":"MyApp1","programType":"Procedure","programId":"MyProc2","statusCode":404,"error":"Program:
 {"appId":"MyApp3","programType":"Service","programId":"MySvc1,"status":"STOPPED","statusCode":200}]
```

Container Information

To find out the address of an element's container host and the container's debug port, you can query CDAP 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 `flows`, `procedures` or `services`
`<element-id>`	Name of the element (*Flow*, *Procedure* or *Custom Service*)

Example:

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

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

CDAP Testing and Debugging Guide.

Scale

You can retrieve the instance count executing different elements from various applications and different element types using an HTTP POST method:

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

with a JSON array in the request body consisting of multiple JSON objects with these parameters:

Parameter	Description
`"appId"`	Name of the Application being called
`"programType"`	One of `flow`, `procedure`, or `service`
`"programId"`	Name of the element (*Flow*, *Procedure*, or *Custom Service*) being called
`"runnableId"`	Name of the *Flowlet* or *Runnable* if querying either a *Flow* or *Service*. This parameter does not apply to *Procedures* because the `programId` is the same as the `runnableId` for a *Procedure*

The response will be the same JSON array with additional parameters for each of the underlying JSON objects:

Parameter	Description
`"requested"`	Maps to the number of instances the user requested for the program defined by the individual JSON object's parameters
`"provisioned"`	Maps to the number of instances that are actually running for the program defined by the individual JSON object's parameters.
`"statusCode"`	The status code from retrieving the instance count of an individual JSON object.
`"error"`	If an error, a description of why the status was not retrieved (the specified element was not found, the requested JSON object was missing a parameter, etc.)

Note that the `requested` and `provisioned` fields are mutually exclusive of the `error` field.

Example

To retrieve the instance count of multiple program runnables in multiple applications, use the HTTP POST command:

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

with the arguments as a JSON string in the body:

```
[{"appId":"MyApp1","programType":"Flow","programId":"MyFlow1","runnableId":"MyFlowlet5"},
 {"appId":"MyApp1","programType":"Procedure","programId":"MyProc2"},
 {"appId":"MyApp3","programType":"Service","programId":"MySvc1,"runnableId":"MyRunnable1"}]
```

If there was no procedure named `MyProc2` in the application `MyApp1`, a possible response could be:

```
[ {"appId":"MyApp1","programType":"Flow","programId":"MyFlow1",
    "runnableId":"MyFlowlet5","provisioned":2,"requested":2,"statusCode":200},
  {"appId":"MyApp1","programType":"Procedure","programId":"MyProc2",
    "provisioned":0,"requested":1,"statusCode":200},
  {"appId":"MyApp3","programType":"Service","programId":"MySvc1,
    "runnableId":"MyRunnable1","statusCode":404,"error":"Runnable: MyRunnable1 not found"} ]
```

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/procedures/Greeting/instances instances`
Description	Find out the number of instances of the Procedure *Greeting* in the Application *HelloWorld*

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 *WhoRunnable* of the Service *WhoService*

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 *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 CDAP, 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 *WhoService* of the Application *HelloWorld*
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
  ```

Logging HTTP API

Downloading Logs

You can download the logs that are emitted by any of the *Flows*, *Procedures*, *MapReduce* jobs, or *Services* running in CDAP. 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`, `mapreduce`, or `services`
`<element-id>`	Name of the element (*Flow*, *Procedure*, *MapReduce* job, *Service*) 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, CDAP 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 CDAP 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, CDAP 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 `cdap` (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/system/apps/HelloWorld/flows/WhoFlow/flowlets/saver/process.busyness?aggregate=true`
Description	Using a *System* metric, *process.busyness*

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*
HTTP Method	`GET <base-url>/metrics/user/apps/HelloWorld/services/WhoService/runnables/WhoRun/names.bytes?aggregate=true`
Description	Using a *User-Defined* metric, *names.bytes* in a Service's Twill Runnable

Comments

The scope must be either `cdap` 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/system/apps/CountRandom/flows/CountRandomFlow/flowlets/
        splitter/process.events.processed?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/system/apps/CountRandom/flows/
  CountRandomFlow/process.events.processed?start=now-5s&count=5
```

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

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

To request user-defined metrics instead of system metrics, specify `user` instead of `cdap` 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
[ "/system/collect.events?aggregate=true",
"/system/apps/HelloWorld/process.events.processed?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
`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 CDAP has not been stopped or restarted (you cannot specify a time range for aggregates):

```
GET <base-url>/metrics/system/apps/CountRandom/process.events.processed?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 CDAP:

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 CDAP 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

CDAP internally uses a variety of System Services that are critical to its functionality. This section describes the RESTful 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 CDAP 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 CDAP System Service

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

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

The status of these CDAP System Servcies 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 CDAP 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

In distributed CDAP installations, 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

Note: In standalone CDAP, these commands will return a Status Code `400 Bad Request`.

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:
	`{ "instances" : 2 }`

Description	Sets the number of instances of the metrics HTTP service to 2

Details of A Deployed Application

For detailed information on an application that has been deployed, use:

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

The information will be returned in the body of the response.

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

HTTP Responses

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