**STRICTLY PRIVATE AND CONFIDENTIAL**

October 31, 2014

Version 2.6

Semios Integration Architecture

Table of Contents

[1 Introduction](#h.hu0pkoz3s67p)

[2 Deployment/Infrastructure](#h.xqb62fvrcxyg)

[3 Configuration](#h.xmu1ma3uwnr5)

[Users](#h.902aixv17iq)

[Data Sources](#h.1me0n1ttjwd1)

[Data Processing Functions](#h.ywnv8j2n8ki0)

[Data Ingestions](#h.dca8mbhfi33s)

[Data Aggregations](#h.1xkitfqorgd1)

[Data Source User Interface](#h.5l1qdhdwn79y)

[4 Application Programming Interface Overview](#h.70w09ury1m9x)

[REST Interface](#h.unu8bu8as859)

[API Requirements](#h.cfj597murr)

[Interface](#h.ws9r94c3l4ea)

[Authentication of User sessions](#h.r6kfsq6hwisd)

[Response Sizes](#h.2aqra6nrxha4)

[Response Time](#h.lpdg8t1lqdbr)

[Units of Measurement](#h.pe38nn3kw0iz)

[Data Types](#h.xsxys1m71w7m)

[REST Query Parameters](#h.hyr4zisvxx40)

[REST Request API](#h.d8w8l592jxvn)

[Queried Data](#h.efrw3jf46gy7)

[5 Application Programming Interface Reference](#h.ce8e50omr39i)

[User Login](#h.j2osv9ym4ttb)

[URI](#h.p3lcmtxrtg6l)

[Request Parameters](#h.x2en37vi9qzq)

[Response](#h.al5p1flvgnzz)

[Example Usage](#h.xso9car2lvmb)

[User Logout](#h.6ckb2af8a4rj)

[URI](#h.lrkc4tb3vrd2)

[Request Parameters](#h.uenpfl5kxmjy)

[Response](#h.f11wdpmbhwmw)

[Example Usage](#h.m7fjcgz66tyg)

[Data Ingest](#h.yncnhmhugzoo)

[URI](#h.dnfvpqmsdddy)

[Request Parameters](#h.734pkxuqiwij)

[Request Headers](#h.9yq4rkruwpt0)

[Response](#h.heegxaizewpc)

[Query Sensor Data](#h.q1wsbqp20m80)

[URI](#h.og6pslgs7ixs)

[Request Parameters](#h.lr5p4wcicdgx)

[Response](#h.pqvqh7x85h8m)

[Query Aggregate Data](#h.oiix1huwmm2k)

[URI](#h.2pkt57b61v9s)

[Request Parameters](#h.3796n0skmhlj)

[Response](#h.cpbw7gzcqiog)

[Query Task History](#h.aeuk76e8oa0e)

[URI](#h.6ewtny9jxrvi)

[Request Parameters](#h.7f7o63m5vr7s)

[Response](#h.v1oxdeib5rdz)

[Appendix](#h.fmi8rsrc3dvn)

[DPF Meta Data](#h.mcgwqoeixf7a)

[DPF Processed Tags](#h.cawmi2xfwosa)

[DPF Processed Data](#h.entv9b4z26pg)

[Semios Log Samples](#h.qmsva1rbbjlp)

[Semios Log Structure](#h.9r6fxvw1rvtr)

[Gateway](#h.b5ao2znqk7rp)

[Generic Node](#h.19lsr2br6dpq)

[Semios Sensor](#h.upi1cjg06s79)

[DPF Processing](#h.quchoh8dqt8v)

[Derivation Completion](#h.xpnbd9bpj7r0)

# 1 Introduction

This document specifies the high level architecture for Semios system integration with PHEMI Central. It describes the Semios - PHEMI Central system integration points and their use, and the deployment and configuration of PHEMI Central for Semios.

M2M integration with PHEMI Central is achieved with the PHEMI REST interface, which is also used internally by PHEMI Central's Web and Command Line interfaces. The PHEMI REST interface provides the capability to write data into the system (ingest) and read data from the system (query).

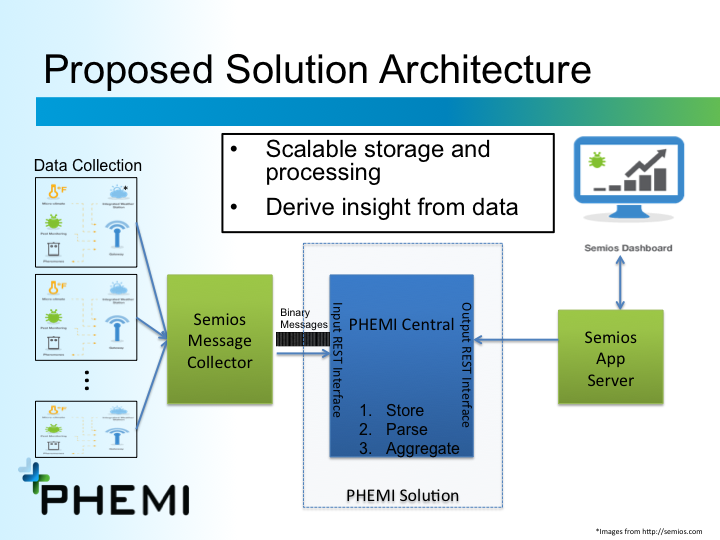
The Semios system uses PHEMI Central in a headless configuration utilizing the PHEMI REST interface. Semios raw sensor data is ingested into the system using this interface. Queries are conducted on Semios raw and derived data (e.g. sensor aggregation) using the same interface. For example, the Semios App Server queries PHEMI Central to provide analytical information back to users to satisfy the Semios “Read Logs” and “Drill Down” use cases.

During ingestion the sensor data are stored, processed and aggregated. The parsing, processing and aggregation tasks are done by custom Data Processing Functions (DPFs). The resulting data structures support the following calculations at scale by providing appropriate sensor aggregations:

* Degree Days
* Frost
* Fireblight

PHEMI Central provides efficient and scalable methods utilizing aggregations to calculate running averages, summations, minimums and maximums of the Semios data. To describe sets of aggregated data, sets are tagged with user defined labels, such as ‘Block’, ‘Property’ or ‘Crop Type’. By using the aggregations, the task of calculating combinatory values is aided and accelerated by PHEMI Central, simplifying the Semios application.

The following diagram illustrates the high level architecture of the Semios-PHEMI integration. The flow of data between PHEMI Central and the Semios components is managed by the PHEMI REST interface.



# 

# 2 Deployment/Infrastructure

# 

# 

# 

# 

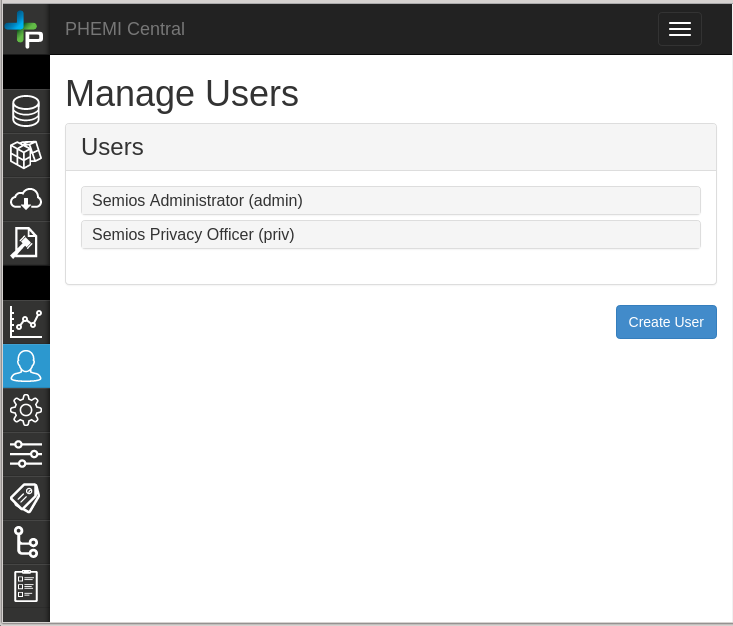
# 

# 3 Configuration

## Users

PHEMI Central is configured for Semios with a PHEMI administrator user **admin** that has Administrator and Analyst authorizations. A privacy officer **priv** also exists.

The following is a screenshot of the user interface showing the pre-populated Semios users:



Semios Users Screenshot

## Data Sources

A **Semios Sensors** data source is created and configured with a category of **Semios**. It’s Institutional Owner and Source owner are set to the **admin** user. The privacy officer for the data source is set to the **priv** user. The document format is **Binary** and the data source retention rule is **Do not delete**.

### Data Processing Functions

Data Processing Functions (DPFs) are instructions for parsing a document (e.g. a sensor log message) into digital assets such as a temperature reading. DPFs are discussed in more detail in another document.

The PHEMI-developed DPFs read and process Semios sensor log messages to produce digital assets that support Semios use cases. The DPFs perform additional calculations to facilitate efficient retrieval of data assets at scale by aggregating sensor readings.

To improve efficiency for many small files, it is preferred that DPFs process many ingested files before terminating. DPFs can be configured to run in different ways:

* When new data arrives: When data is ingested, the DPF is run immediately.
* Manually: When data is ingested, the DPF is not run until the user manually runs the DPF. This is done by hitting the “Execute” button on the Data Processing Function tab of the Datasource page.
* Periodic: The DPF is run on a periodic basis as specified by the periodic value, regardless if data has been ingested or not; the DPF processes ingested data starting at the first incompleted file.

The trigger setting for the Semios data processing function is set to \*Periodic for 5 Minutes\*. This means that when Semios data is ingested, derived data will not necessarily be generated immediately, but generated in 5 minute intervals.

### Data Ingestions

The process of inputting data into the system is referred to as *ingestion*. The system will ingest and process input documents. The document ingested by the system is considered the *raw data*.

As part of document ingestion, the system will annotate the document with *meta data* which describes the document. Meta data is not generated by examining the contents of the ingested document, but rather by looking at the document from an external point of view. For example, meta data includes descriptive information such as the data source the document was ingested into, the timestamp when the document was ingested, the name of the document and more. The full list of descriptive meta data is provided in the [DPF Meta Data](#h.mcgwqoeixf7a) section in the Appendix.

The ingested document can be processed further with a Data Processing Function (DPF). The data generated by the DPF is considered *derived data*. The DPF generated data is done by analyzing the ingested document and producing additional information. Any information that requires analyzing the contents of the document is considered derived data such as data mined information, calculated information and inferred information. For a list of derived data and their types for Semios, please see the [DPF Processed Tags](#h.cawmi2xfwosa) and [DPF Processed Data](#h.entv9b4z26pg) sections in the Appendix.

The DPF can also generate information that system will aggregate over specific time based *bins.* The data generated by the system based on the information provided by the DPF is considered *aggregated data*. The bin types specified in [Query Aggregate Data](#h.oiix1huwmm2k). The aggregated data is specified in [Data Aggregations](#h.1xkitfqorgd1).

To summarize, there are four types of data:

* Raw Data: The input document provided by an external entity
* Meta Data: Descriptive information about the input document gathered without examining the contents of the input document
* Derived Data: Data mined, calculated or inferred data generated by a DPF by analyzing the contents of the input document
* Aggregated Data: Calculated time based data

### Data Aggregations

PHEMI Central supports aggregations over multiple contiguous readings from the same sensor and use them to compute the following derived values:

* number of data points in the aggregation (count)
* minimum reading
* maximum reading
* sum of readings
* sum of squares for the readings

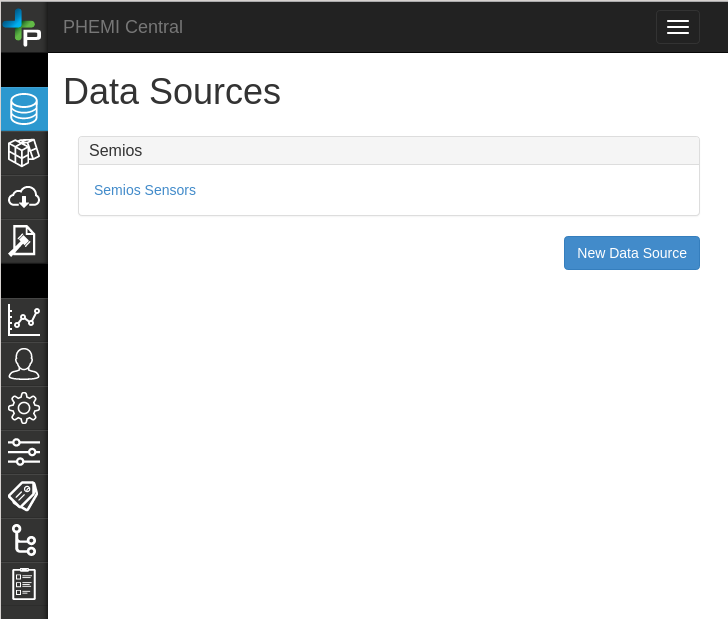
The aggregations are currently computed on a hourly, daily, monthly and yearly basis, per sensor. Several separate aggregations are computed for each type of sensor, depending on the specific sensor.

Wherever PHEMI Central returns an aggregated value (e.g., average hourly temperature), the number of data points used to calculate the aggregated value is also provided. The Semios web application can use this count to determine how many data points were not received by the Semios aggregator, fetching replacement data from an external web service if necessary.

The sum and the sum of squares for the readings are used to calculate the mean and the standard deviation.

### Data Source User Interface

PHEMI provides a graphical user interface to configure and use a data source. The following screenshot shows a list of data sources:

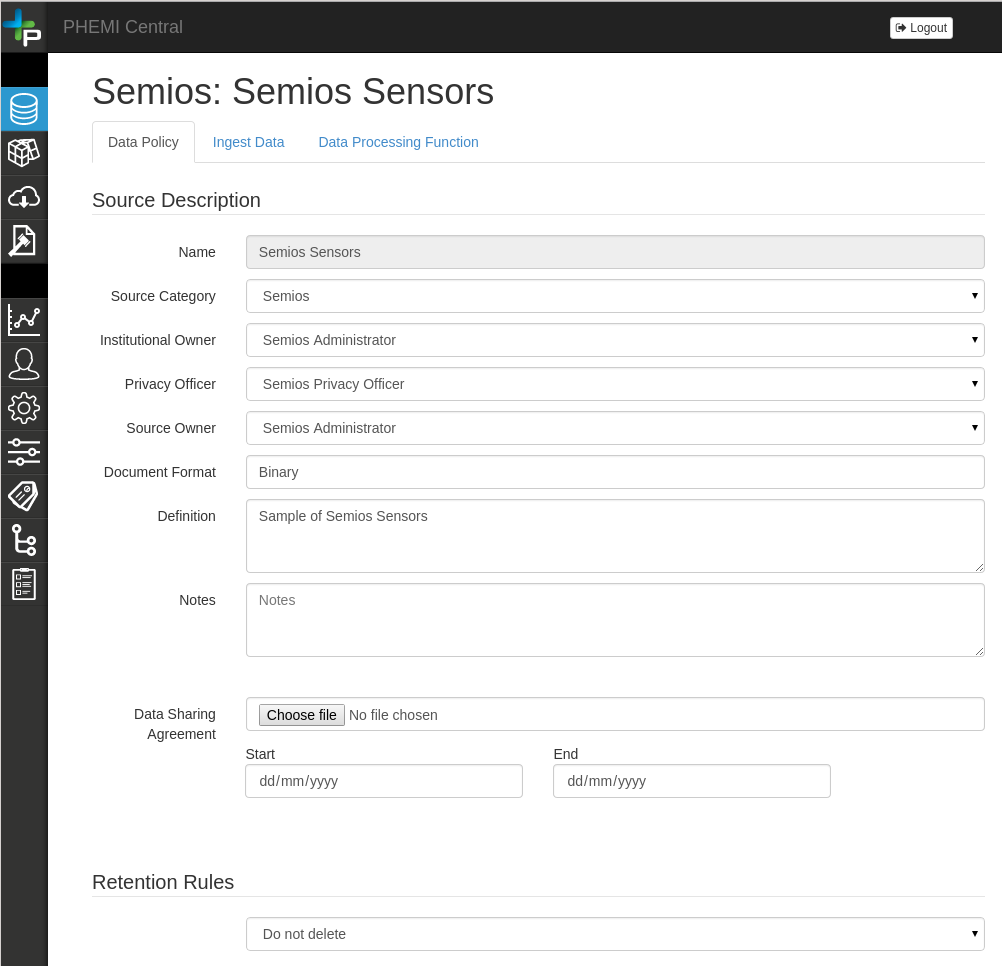


List Data Sources Screenshot

When a data source is selected, a screen is displayed showing:

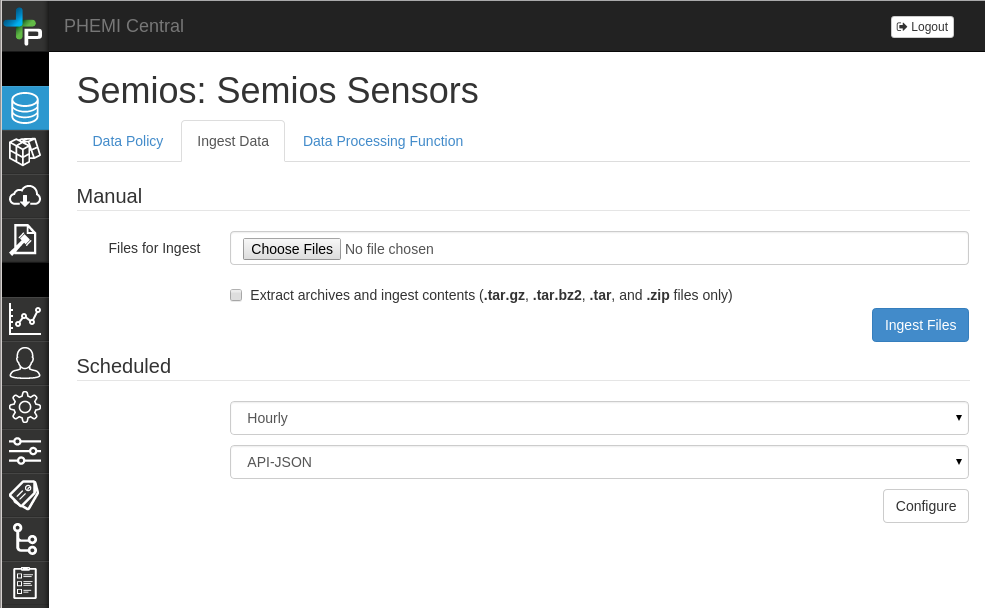
* **Data Policy** - Provide the information such as ownership, privacy, data sharing and other policy information
* **Ingest Data** - Provides a method of ingesting data by uploading a single or set of files
* **Data Processing Function** - The DPF to be used to process the ingested data to generate a set of derived data

The following is a screenshot for the Data Policy Details page:



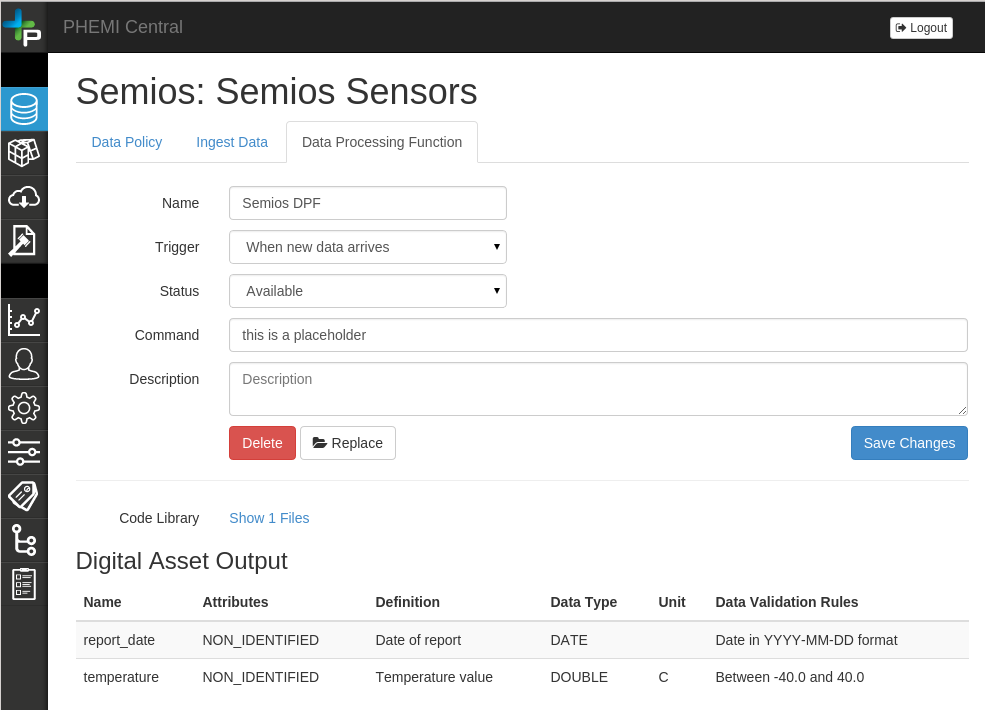
Data Source Data Policy Details Screenshot

The following is a screenshot for ingesting data to a data source:



Data Source Ingest Data Screenshot

The following is a screenshot for uploading a DPF to the data source so it can process the Semios log information:



Data Source DPF Upload Screenshot

# 

# 

# 4 Application Programming Interface Overview

## 

## REST Interface

PHEMI Central provides an HTTP REST interface to support data ingest and querying. REST is an architecture style for designing networked applications using HTTP as its stateless, client-server, cache-able communications protocol. RESTful applications use HTTP requests and their associated methods (POST, PUT, GET, DELETE) to create, update, read, and delete data (<http://en.wikipedia.org/wiki/Representational_state_transfer>). PHEMI uses JSON as its data format and follows the HAL standard for JSON: for definitions see <http://stateless.co/hal_specification.html>.

Data is ingested by an HTTP POST to a specific URL, while data is queried by an HTTP GET to a specific URL.

If an *incorrect* URL is specified, the following will occur:

* If the REST operation is a GET or POST, a **404 Not Found** code is returned with a message stating the resource could not be found
* If the REST operation is something other than GET or POST (i.e., PUT or DELETE), a **405 Method Not Allowed** code is returned with a message stating the resource method is not supported by the resource.

## 

## API Requirements

The following requirements are defined with respect to the API.

### Interface

As described above, the interface is built as a RESTful web service using HTTP as the underlying transport protocol.

### Authentication of User sessions

An authenticated user session needs to be created before using the interface. During this process, a valid user credential is required from which a user session is created as the user logs in successfully. A cookie that holds the session information is required to perform further operations, such as ingesting data, querying aggregations and logging out, etc.

*Future: The REST interface will allow for stateless “signed” Token based authentication.*

### Response Sizes

All query responses should be small enough to be buffered in memory on the client side (i.e., a maximum of a few megabytes of decompressed data).

This requirement can only be met if the query for multiple identifiers encompasses a reasonably sized result set (e.g., identifiers returned across a reasonably short time range).

### Response Time

Unless otherwise noted, aggregated queries using a single tag will return within a 250 ms response time, exclusive of network latency between Semios and PHEMI Central.

When multiple identifiers are queried (e.g., a set of sensor ids returned by a range query), the response time will be less than the total number of identifiers in the set multiplied times 250 ms (e.g., 10 sensors in the set would give a max response time of 2.5 seconds). For best performance a single identifier is preferred for the query.

### Units of Measurement

Each data item should be ingested as the same base type (one of DOUBLE, LONG or STRING). The units of measurements must also be consistent (for example, a duration in milliseconds units should always be milliseconds).

### Data Types

Ingested data and query parameter data types must be consistent. More specifically, the data types output by the DPF must be consistent with the data types used to query data (see Query Parameters below).

## REST Query Parameters

A query must provide parameters that specify: the field to search, the value to search for, the data type of the field, and the comparison operator to use ( =, >, <, etc.). URI query parameters follow the MongoDB convention of prefixing an operator with a double underscore ‘\_\_’; for example, ‘\_\_eq’ is equals, ‘\_\_ge’ is greater than or equal to, etc. (see the Query Operator Suffix Chart below).

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Symbol** | **Description** | **Example** |
| *[None]* |  | No operator provided so by default, it means equals | agg\_id=id=sensor\_2 |
| \_\_eq | = | Equals | agg\_id\_\_eq=id=sensor\_2 |
| \_\_gt | > | Greater than | bin\_value\_\_gt=2014 |
| \_\_gte | >= | Greater than or equals to | bin\_value\_\_gte=2014 |
| \_\_lt | < | Less than | bin\_value\_\_lt=2015 |
| \_\_lte | <= | Less than or equals to | bin\_value\_\_lte=2015 |
| \_\_in | [ ] | In a set of comma separated values | agg\_id\_\_in=id=sensor\_1,id=sensor\_2 |

Query Operator Suffix Chart

The data type must be specified as part of the query parameter and it must match the data type that is output by the data processing function that processed the ingested data. Valid data types are: DATE, DOUBLE, LONG and STRING. If no type is specified in the query parameter, the default type used is STRING. If the type specified in the query parameter does not match the type specified to the system when the document was ingested, the following will occur:

* If a type conversion cannot occur such as converting a non-numeric STRING to a LONG, then a **400** code is returned indicating a type conversion error occurred
* If a type conversion could occur, but is of the wrong type then no results will be returned *[logged as defect AGIL-1426 REST Query should return error if datatype is mismatched]*

Query parameters or values must be URL encoded when required (for example, a space is encoded as %20).

An example query parameter that matches (equals) a specific device id with a LONG data type would be:

derived:model\_\_eq=LONG:38

### REST Request API

All API requests start with the same URI prefix containing the data source identifier. The following is the API request that generates the URI prefix for a specific data source name:

GET /rest/data\_sources?name=Semios

An HTTP response of 200 OK indicates success and contains the following JSON:

{  
 "\_links": {  
 "curies": [  
 {  
 "href": ["http://docs.phemi.com/relations/{rel}"](http://docs.phemi.com/relations/%7Brel%7D),  
 "name": "ph",  
 "templated": true  
 }  
 ],  
 "ph:data\_source": [  
 {  
 "href": "/rest/data\_sources/53ea9cd8e779892f24ee8972",  
 "title": "Semios"  
 }  
 ],  
 "ph:find": {  
 "href": "/rest/data\_sources/{id}",  
 "templated": true  
 },  
 "self": {  
 "href": "/rest/data\_sources?name=Semios"  
 }  
 },  
 "count": 1  
}

From the response, use the URI specified in *["\_links"]["ph:data\_source"]["href"]* for all future requests (i.e., "/rest/data\_sources/53ea9cd8e779892f24ee8972"). The URI prefix /rest/data\_sources/<id> in the returned response is used throughout this document.

Note that the URI prefix cannot be taken literally from the above example but must be determined when the Semios Application starts for the first time. Once the URI is obtained, it does not have to be re-queried for in subsequent sessions as it is valid for the life of a specific datasource.

### Queried Data

The Semios Web Application retrieves device log records from PHEMI Central using the Query Data APIs.

There are three kinds of data to retrieve:

* Metadata: Descriptive information generically describing the ingested data
* Derived data: individual pieces of data extracted and transformed from ingested data used for later analysis
* Aggregated data: calculated data based on the derived data
* Raw Data: The data that was ingested

To support each type of data, separate HTTP GET APIs are provided.

# 

# 

# 

# 5 Application Programming Interface Reference

## User Login

The user logs into PHEMI Central and creates a new active session by sending a POST request.

### 

### URI

POST /rest/user\_sessions

### Request Parameters

The post request expects valid user credential in JSON format in the payload body.

{

“username”: ”<valid username>”,

“password”: “<valid password>”

}

User Session Request Body Example

### Response

The following are the HTTP status codes that can be expected:

* **201 Created** - Successful login, the session has been created for the current user.
* **400 Bad Request** - Failed login, JSON body is not well formated or validation errors.
* **401 Unauthorized** - Failed login, user credentials are incorrect.
* **409 Conflict** - Failed login, there is already an active session for the current user.

For a successful login (**201 Created** return code), the body of the response will contain status code, session id, and URI for the session; additionally, a session cookie is returned . The following is an example:

{

"code": 201,

"id": "phemi-agile-session:b5a04a2353814a679a9df303a387aca2"

}

Where,

* code**:** The status code.
* id**:** The session id.

### Example Usage

The following is a Python example of logging in, saving the user session cookie, making a request and logging out:

import urllib  
import httplib2  
import json  
  
http = httplib2.Http()  
  
*# Login*   
url = **'http://<hostname>:<port>/rest/user\_sessions'**  
body = {**'username'**: **'<username>'**, **'password'**: **'<password>'**}  
headers = {**'Content-type'**: **'application/json'**}  
login\_response, login\_content = http.request(url,   
 **'POST'**,   
 headers=headers,   
 body=json.dumps(body))  
  
*# Save the login session cookie into the header for the next call*  
headers = {**'Cookie'**: login\_response[**'set-cookie'**]}  
  
*# Make a REST call and include the login session cookie in the header*  
url = **'http://<hostname>:<port>/rest/data\_sources/<datasource\_id>/raw\_data'**  
query\_response, query\_content = http.request(url,  
 **'GET'**,   
 headers=headers)  
  
*# Log out using the login session cookie in the header*  
url = **'http://<hostname>:<port>/rest/user\_sessions'**  
headers = {**'Cookie'**: login\_response[**'set-cookie'**]}  
logout\_response, logout\_content = http.request(url,   
 **'DELETE'**,   
 headers=headers)

## User Logout

The user can logout and delete a user session by sending a DELETE request containing the cookie for the user session.

### URI

DELETE /rest/user\_sessions

### Request Parameters

The delete does not expect a JSON body or request parameters. However, a cookie that contains the user session information is required.

### Response

The following are the HTTP status codes that can be expected:

* **204 No Content** - The session for the current user has been deleted.
* **404 Not Found** - The specified user session does not exist.

### Example Usage

The following is an example of logging out:

import urllib  
import httplib2  
import json  
  
http = httplib2.Http()  
  
*# Login*   
url = **'http://<hostname>:<port>/rest/user\_sessions'**  
body = {**'username'**: **'<username>'**, **'password'**: **'<password>'**}  
headers = {**'Content-type'**: **'application/json'**}  
login\_response, login\_content = http.request(url,   
 **'POST'**,   
 headers=headers,   
 body=json.dumps(body))  
  
*# Include the login session in the*   
headers = {**'Cookie'**: login\_response[**'set-cookie'**]}  
  
url = **'http://<hostname>:<port>/rest/user\_sessions'**  
headers = {**'Cookie'**: login\_response[**'set-cookie'**]}  
logout\_response, logout\_content = http.request(url,   
 **'DELETE'**,   
 headers=headers)

## Data Ingest

The Semios Message Collector submits data to PHEMI Central by issuing an HTTP POST to the REST ingest URI.

The ingested data contains:

* a version identifier
* a device identifier
* a set of non-hierarchical logs (data)
* a set of non-hierarchical tags

### URI

POST /rest/data\_sources/<datasource\_id>/ingestions

### Request Parameters

The post expects a JSON payload with the Semios device data. See the [appendix](#h.qmsva1rbbjlp) for more information.

### Request Headers

Prefer: respond-async

The presence of the **Prefer** header with the value **respond-async** will cause requests to be executed in the background. Control is returned to the REST client immediately. If this header is used, a return code of **202** **Accepted** is returned.

### Response

The following are the HTTP status codes that can be expected:

* **200 OK -** The ingestion request has been completed successfully, the raw data has been persisted in PHEMI Central
* **202 Accepted** - The ingestion request has been accepted, but not completed
* **400 Bad Request** - The JSON body is not well formed
* **401 Unauthorized** - The user session is invalid
* **403 Forbidden** - The user session does not have permission to perform an ingestion
* **404 Not Found** - The Data Source does not exist
* **500 Internal Server Error** - An internal system error has occurred

In the event of a **202 Accepted** return code, the body of the response will contain status and meta information. The following is an example:

{

"\_links": {

"curies": [

{

"href": "http://docs.phemi.com/rels/{rel}",

"name": "ph",

"templated": true

}

],

"ph:task\_history": {

"href": "/rest/tms/task\_histories/54108814e779891b9b1a1eb4",

"title": "Ingest and Derive for Data Source ECG"

},

"self": {

"href": "/rest/tms/ingest\_statuses/54108814e779891b9b1a1eb4"

}

},

"error": null,

"result": null,

"status": "pending"

}

*Ingestion Response Example*

If the task is still running, the return status code will continue to be **202 Accepted**; otherwise, if the task finished and was successful, a **200 OK** response code will be returned. If the task failed, a **500 Internal Server Error** will be returned along with error details in the response body.

In this example, the task is still running as the status shows on the last line. To determine the status of the ingestion at a later time, use the ***self href*** link specified in this response. In this example, the following is the link to use to determine the running status:

/rest/tms/ingest\_statuses/54108814e779891b9b1a1eb4

The following is an example of a status while the ingest task is still running:

{

"\_links": {

"curies": [

{

"href": "http://docs.phemi.com/rels/{rel}",

"name": "ph",

"templated": true

}

],

"ph:task\_history": {

"href": "/rest/tms/task\_histories/54108814e779891b9b1a1eb4",

"title": "Ingest and Derive for Data Source Semios Retention"

},

"self": {

"href": "/rest/tms/ingest\_statuses/54108814e779891b9b1a1eb4"

}

},

"error": null,

"result": null,

"status": "running"

}

If the task is done, then the status can be queried using the task history URL. In the above example, the task history URL would be:

/rest/tms/task\_histories/54108814e779891b9b1a1eb4

The following is an example of the above task’s history when it is finished:

{

"\_id": {

"$oid": "54108814e779891b9b1a1eb4"

},

"\_links": {

"curies": [

{

"href": "http://docs.phemi.com/rels/{rel}",

"name": "ph",

"templated": true

}

],

"self": {

"href": "/rest/tms/task\_histories/54108814e779891b9b1a1eb4"

}

},

"action": "chained",

"error": "",

"extra\_props": {

"tasks": [

{

"\_id": {

"$oid": "54468bb0e779891099693f44"

},

"action": "ingest",

"error": "",

"extra\_props": {

"data\_source\_id": "543d87d8e779897cfc5d3330",

"data\_source\_name": "Semios Sensors",

"total\_metasize": 212,

"total\_objects": 1,

"total\_rawsize": 780

},

"finished": "2014-10-21T16:37:04.576Z",

"name": "Semios Retention",

"queued": "2014-10-21T16:37:04.464Z",

"started": "2014-10-21T16:37:04.498Z",

"status": "finished",

"uniquename": "datasource-Semios Sensors",

"username": "admin"

},

{

"\_id": {

"$oid": "54468bb0e779891099693f45"

},

"action": "derive",

"error": "",

"extra\_props": {},

"finished": "2014-10-21T16:37:19.683Z",

"name": "Semios Retention",

"queued": "2014-10-21T16:37:04.488Z",

"started": "2014-10-21T16:37:04.583Z",

"status": "finished",

"uniquename": "datasource-Semios Retention",

"username": "admin"

}

]

},

"finished": "2014-10-21T16:37:19.686Z",

"name": "Ingest and Derive for Data Source Semios Sensors",

"queued": "2014-10-21T16:37:04.491Z",

"started": "2014-10-21T16:37:04.494Z",

"status": "finished",

"uniquename": "Ingest and Derive for Data Source Semios Sensors",

"username": "admin"

}

Notice that the task status is set to “finished” and the time the task finished is also given. If the DPF has not been configured to process the derived data immediately, a finished task only indicates that the ingestion of raw data is complete (the raw data has been persisted in PHEMI Central). To determine when generation of derived data has been completed, follow the instructions in the [Derivation Completion](#h.xpnbd9bpj7r0) section.

## Query Sensor Data

The REST interface provides the ability to query data based on various criteria such as attribute values and ranges. Query parameters or values must be URL escaped when required.

### URI

GET /rest/data\_sources/540e5056e77989182bd5aad0/raw\_data?derived:id\_\_gte=STRING:sensor\_1130&derived:id\_\_lt=STRING:sensor\_1130

### Request Parameters

* <attribute-name>\_\_<operator>=<lexicoder>:<filter-value>
  + <attribute-name> - the raw-data attribute name to query for
  + <operator> - defines how a query should be performed
    - types:
      * \_\_eq - performs an exact match
        + e.g. derived:attribute\_\_eq=DOUBLE:33.44
      * \_\_lt - query for values less than
        + e.g. derived:attribute\_\_lt=DATE:2014-10-15T00:54:32.750Z
      * \_\_lte - query for values less than or equal
        + e.g. derived:attribute\_\_lte=LONG:345543
      * \_\_gt - query for values greater than
        + e.g. derived:attribute\_\_gt=STRING:aaaaa
      * \_\_gte - query for values greater than or equal
        + e.g. derived:attribute\_\_gte=DOUBLE:5.50
      * \_\_in - query for values in a list
        + e.g. derived:attribute\_\_in=STRING:blue,red,green,purple
  + <lexicoder> - specifies the data type of the query value
    - determined by Data Processing function (DPF) that creates derived data during data ingestion.
    - types:
      * STRING - a string data-type. The default value.
      * DOUBLE - a double data type
      * LONG - a long data type
      * DATE - a date data type - accepts milliseconds since the epoch or an ISO 8601 date string.
  + <filter-value> - the value to query by.
  + e.g. /rest/datasources/<datasource-id>/raw\_data?derived:id\_\_eq=STRING:sensor\_1138
* skip=<number of rows to skip in the query result before returning subsequent rows in the resultset>
  + default skip value is 0, meaning don’t skip any results
    - an exception will be thrown if a number is provided that is less than 0
  + e.g. a query finds 50 rows (identified as row #1 to row #50):
    - no skip defined return: rows #1 to #50 (default is 0)
    - skip=0 return: rows #1 to #50
    - skip=1 return: rows #2 to #50
    - skip=10 return: rows #11 to #50
    - skip=50 return: none
    - skip=100 return: none
    - *Note*: The max number of rows returned in the resultset is capped by the ‘limit’ parameter value discussed below.
  + e.g. /rest/datasources/<datasource-id>/raw\_data?derived:id\_\_eq=STRING:sensor\_1138&skip=50
    - query by device id (sensor), skip the first 50 rows of the query results and return row #51 onwards
* limit=<the maximum number of rows to return in the resultset>
  + default is 50 rows, largest limit value is 2500
    - an exception will be thrown if a limit value greater than 2500 or less than 1 is provided.
  + e.g. a query finds 100 rows (identified as row #1 to row #100)
    - no limit defined return: rows #1 to #50 (default is 50)
    - limit=1 return: only row #1
    - limit=2 return: rows #1 and #2
    - limit=10 return: rows #1 to #10
    - limit=100 return: rows #1 to #100
    - limit=200 return: rows #1 to #100
  + /rest/datasources/<datasource-id>/raw\_data?derived:id\_\_eq=STRING:sensor\_1138&limit=100
    - limit the result set to 100 rows -> row #1 up to #100 of the query results will be returned in the result set.
  + when ‘limit’ is used in conjunction with ‘skip’, pagination-type functionality can be realized:
    - e.g. a query finds 100 rows (identified as row #1 to row #100)
      * skip=0,limit=50 return: rows #1 to #50
      * skip=1,limit=50 return: rows #2 to #51
      * skip=50,limit=50 return: rows #51 and #100 returned
      * skip=75,limit=50 return: rows #76 to #100 returned
      * skip=99,limit=50 return: only row #100
      * skip=100,limit=50 return: none
* order\_by=<a comma-separated list of attributes specifying how the result set being returned by a query should be ordered>
  + For example,

/rest/datasources/<datasource-id>/raw\_data?derived:id\_\_eq=STRING:sensor\_113&order\_by=derived:data|temperature,-derived:data|wind\_speed

* + - the above query would order the result set by:
      * derived:data|temperature
      * and then by -derived:data|wind\_speed
* the ‘-’ before the name implies reverse ordering so instead of ordering in ascending order, it would be in descending order
  + The order\_by attributes are applied only to the result set returned by the query and have no effect on what rows are returned for a specific query. As such, the result set would contain the same rows regardless of what order\_by criteria was used, just ordered differently
    - an exception to this is if one or more of the order\_by attributes matches an attribute in the query itself. In this case, the presence of a ‘-’ will be used to make a more efficient query and could potentially return rows from the database in a different order than a query not using the ‘-’.
    - /rest/datasources/<datasource-id>/raw\_data?derived:id\_\_lte=STRING:sensor\_113&order\_by=-derived:id

### Time based Queries

For efficient time based queries, a “timeline” attribute is generated automatically by combining the device id and log timestamp fields. The timeline field is used in executing time based queries.

The timeline attribute is specified by <device id>|<ISO date>.

The following is an example timeline attribute entry created from an ingested Semios json document:

|  |  |
| --- | --- |
| derived:timeline | gateway\_53bee4cd5295273c53d17ce6|2014-09-22T23:59:59 |

#### URI

The following is a sample REST query that returns the latest reading document for a specific device starting from a given day:

GET

/rest/data\_sources/54498f0fe779891aa9ae81bc/raw\_data?derived:timeline\_\_lte=STRING:gateway\_53bee4cd5295273c53d17ce6|2014-09-23T23:59:59&derived:timeline\_\_gt=STRING:gateway\_53bee4cd5295273c53d17ce6|&order\_by=-derived:timeline&limit=1

* Using the “\_\_lte” (less than or equal to) operand together with the “order\_by=-derived:timeline” criteria produces a descending scan (specified by preceding ‘derived:timeline’ with a ‘-’) starting from the last second of the given date.
* Using “derived:timeline\_\_gt=STRING:<device-id>|” in conjunction with “derived:timeline\_\_lte” will ensure only rows belonging to the device in question will be returned by the query. If this parameter was not specified, another device’s data could potentially be returned if no rows were found for the device being queried for.
* Using the limit=1 operand then returns the latest device json document for the given device starting from that date.

#### 

#### URI

The following query will get all the readings on 2014-09-22 for a given device id:

GET

/rest/data\_sources/54498f0fe779891aa9ae81bc/raw\_data?derived:timeline\_\_lt=STRING:gateway\_53bee4cd5295273c53d17ce6|2014-09-23T00:00:00&derived:timeline\_\_gt=STRING:gateway\_53bee4cd5295273c53d17ce6|2014-09-21T23:59:59

* The date range is specified by the following operand and values:

|  |  |
| --- | --- |
| \_\_lt (less than) | 2014-09-23T00:00:00 |
| \_\_gt (greater than) | 2014-09-21T23:59:59 |

* Additional filtering criteria like “limit=<n>” can also be used if needed.

### Response

{

"\_embedded": {

"ph:raw": [

{

"\_links": {

"self": {

"href": "/rest/data\_sources/540e5056e77989182bd5aad0/raw\_data/04D67DBCD2401000"

}

},

"derived:data|temperature": "4.7",

"derived:data|time": "2014-10-15T00:54:32.750Z",

"derived:data|wind\_speed": "7.9",

"derived:id": "sensor\_1138",

....

},

{

"\_links": {

"self": {

"href": "/rest/data\_sources/540e5056e77989182bd5aad0/raw\_data/04D669F769001000"

}

},

"derived:data|temperature": "4.8",

"derived:data|time": "2014-10-15T00:54:32.750Z",

"derived:data|wind\_speed": "7.7",

"derived:id": "sensor\_1139",

....

}

]

},

"\_links": {

"curies": [

{

"href": "http://docs.phemi.com/rels/{rel}",

"name": "ph",

"templated": true

}

],

"ph:find": {

"href": "/rest/data\_sources/540e5056e77989182bd5aad0/raw\_data/{id}",

"templated": true

},

"ph:raw": [

{

"href": "/rest/data\_sources/540e5056e77989182bd5aad0/raw\_data/04D67DBCD2401000"

},

{

"href": "/rest/data\_sources/540e5056e77989182bd5aad0/raw\_data/04D669F769001000"

}

],

"self": {

"href": "/rest/data\_sources/540e5056e77989182bd5aad0/raw\_data?derived:id\_\_gte=STRING:sensor\_1130&derived:id\_\_lt=STRING:sensor\_1130"

}

},

"count": 2,

"title": "Raw Data Collection"

}

*Query Data Response Example*

The returned JSON follows the JSON Hypertext Application Language (HAL) standard. The document contains multiple parts identified by the following keys:

**count** - the number of rows found

**\_embedded[“ph:raw”]** - an array containing the individual rows returned

**\_links** - query meta data and URI values for individual rows. The individual URIs are restricted to admin users.

In addition to the HAL information, the query response contains all the derived data where the key is the name of the derived data as specified by the Semios DPF manifest.

When an HTTP request is made against the raw data REST interface, the following HTTP status codes can be expected depending on the query result:

* **200 OK** - query successful
* **400 Bad Request** - Invalid criteria parameter value provided
* **401 Unauthorized** - the user is not authenticated
* **403 Forbidden** - the authenticated user is not authorized to perform the query.
* **404 Not Found** - a resource, such as a Datasource, associated with the query does not exist
* **500 Internal Server Error** - an internal system error has occurred.

## 

## Query Aggregate Data

The Aggregate Data REST interface provides the ability to query for aggregated sensor data. The aggregated sensor data needs to be specified and encoded into the DPF for processing.

Aggregate queries return the count, min, max, sum and the sum of squares for each requested key.

### URI

GET /rest/data\_sources/5418c1f4e7798936ea540b71/aggregations?agg\_id=id=sensor\_239a392afd8978015311bf32&bin=DAY&bin\_value=2014-06-23&embed=aggregation

### Request Parameters

* The data source id is embedded within the URL and is not technically a request parameter
* agg\_id=<attribute name>=<attributes value>
  + The identifier that specifies the entity that aggregation is being done for. For Semios, the agg\_id is the device identifier and is specified as the following:

&agg\_id=id=sensor\_539a392afd8978015311b9c9

* + Required: Yes
  + Accepted operators: \_\_eq or \_\_in, defaults to \_\_eq if not specified

Example:

&agg\_id=id=sensor\_539a392afd8978015311b9c9

&agg\_id\_\_eq=id=sensor\_539a392afd8978015311b9c9

&agg\_id\_\_in=id=sensor\_539a392afd8978015311b9c9,id=sensor\_939a392cfd89782153117e78

* bin=<bin identifier>
  + The identifier that specifies the type of aggregation. This is also referred to as the “bin type”. The values can be either: HOUR, DAY, MONTH, YEAR. For example

&bin=HOUR

* + Required: Yes
  + Accepted operators: \_\_eq, defaults to \_\_eq if not specified

Example:

&bin=HOUR

&bin\_\_eq=MONTH

* bin\_value=<day and time>
  + The identifier that specifies value for the aggregation bin type. The bin value is the day and time that works with the bin type. For example:

&bin\_value=2014-08-14T09

* + Required: Yes
  + Accepted operators: \_\_eq, \_\_lt, \_\_lte, \_\_gt, \_\_gte, defaults to \_\_eq if not specified. The bin\_value must be bounded, therefore the usage of a \_\_lt or \_\_lte must be accompanied by a \_\_gt or \_\_gte operator. If the two ranges are mutually exclusive, the query will return a status code 500.

Examples:

&bin\_value=2014-08-14T09

&bin\_value\_\_eq=2014-08-14

&bin\_value\_\_lt=2014-08

&bin\_value\_\_lte=2014

&bin\_value\_\_gt=2014-08-14T09

&bin\_value\_\_gte=2014-08-14T09

-----under construction-----

* If using a single id, you can use skip and limit, but if you use multiple id’s then you can’t. And no order by at all but the results should already be sorted by decreasing timestamp.
* If searching for multiple sensor id’s, if the default limit of records is hit, then the response will be a mixture of the different id’s in no particular order….so check the number of records that you get back and maybe split your query up one for each sensor id….

### Response

{

"\_embedded": {

"ph:aggregation": [

{

"\_links": {

"self": {

"href": "/rest/aggregations/5418c1f4e7798936ea540b71|222"

}

},

"agg\_id": "id=sensor\_239a392afd8978015311bf32",

"bin": "DAY",

"bin\_value": "2014-06-23",

"data\_source": "5418c1f4e7798936ea540b71",

"data|temperature": {

"count": 1,

"max": 20,

"min": 20,

"sum": 20,

"sum\_of\_squares": 400

},

"data|wind\_speed": {

"count": 1,

"max": 65,

"min": 65,

"sum": 65,

"sum\_of\_squares": 4225

},

"row\_id": "5418c1f4e7798936ea540b71|222"

}

]

},

"\_links": {

"curies": [

{

"href": "[http://docs.phemi.com/rels/{rel}](http://docs.phemi.com/rels/%7Brel%7D)",

"name": "ph",

"templated": true

}

],

"ph:aggregation": [

{

"href": "/rest/aggregations/5418c1f4e7798936ea540b71|222"

}

],

"ph:find": {

"href": "/rest/aggregations/{id}",

"templated": true

},

"self": {

"href": "/rest/data\_sources/5418c1f4e7798936ea540b71/aggregations?agg\_id=id=sensor\_239a392afd8978015311bf32&bin=DAY&bin\_value=2014-06-23&embed=aggregation"

}

},

"count": 1,

"title": "Aggregated Data Collection"

}

*Query Aggregate Data Response Example*

The returned JSON follows the JSON Hypertext Application Language (HAL) standard. The JSON document contains multiple parts identified by the following keys:

**count** - the number of rows found

**\_embedded[“ph:raw”]** - an array containing the individual rows returned

**\_links** - query meta data and URI values for individual rows. The individual URIs are restricted to admin users.

The returned JSON document contains the result of the aggregation data query. The data is divided into groups of devices and bins. In the above example, the device has an id of “sensor\_239a392afd8978015311bf32” and a bin type of DAY. The data is further divided into aggregation groups based on the derived data information provided in the Semios DPF manifest. Only information that the Semios DPF generates as aggregated data will be returned in the aggregation query.

The following are the HTTP status codes that can be expected depending on the query results:

* **200 OK -** Request successful.
* **400 Bad Request -** An invalid request parameter was provided
* **401 Unauthorized -** The user is not logged in.
* **403 Forbidden -** The logged in user is not authorized and does not have permission to perform the aggregation operation.
* **404 Not Found -** The data source being queried by does not exist.
* **500 Internal Server Error** - An internal system error has occurred.

## 

## Query Task History

Data ingestion and derivation is managed by a task management system. The task management system is responsible for managing the lifecycle of a task such as data ingestion and derivation. Task history queries return information about a task such as the task name, action, errors, status, queued time, start time, finish time, the initiator and other information.

### URI

GET /rest/tms/task\_histories

### Request Parameters

* embed=task\_history

This parameter indicates that the task history details are expected.

Example:

embed=task\_history

* action=<action of interest>

A string value that represents the action that the task is managing. Some example actions that can be queried are:

* snapshot
* cleanup
* derive
* ingest
* chained

A chained task is a set of tasks set to run sequentially. For example, if a DPF has been configured to run data derivation immediately after ingestion, then a chained task is created containing an ingest task and a derive task.

Accepted operators: \_\_eq, \_\_lt, \_\_lte, \_\_gt, \_\_gte, defaults to \_\_eq if not specified.

Example:

action=derive

* queued=<queue time>

A string value that represents the time that the task was queued waiting to be started. The following is an example of the format:

2014-10-21T19:16:01.020Z

Accepted operators: \_\_eq, \_\_lt, \_\_lte, \_\_gt, \_\_gte, defaults to \_\_eq if not specified.

Examples:

queued=2014-10-21T19:16:01.020Z

queued\_\_gte=2014-10-21T19:16:01.020Z

queued\_\_lt=2014-10-21T19:16:01.020Z

* started=<task start time>

A string value that represents the time that the task was started. The following is an example of the *format*:

2014-10-21T19:16:01.020Z

Accepted operators: \_\_eq, \_\_lt, \_\_lte, \_\_gt, \_\_gte, defaults to \_\_eq if not specified.

Examples:

started=2014-10-21T19:16:01.020Z

started\_\_gte=2014-10-21T19:16:01.020Z

started\_\_lt=2014-10-21T19:16:01.020Z

* finished=<task completion time>

A string value that represents the time that the task was completed. The following is an example of the *format*:

2014-10-21T19:16:01.020Z

Accepted operators: \_\_eq, \_\_lt, \_\_lte, \_\_gt, \_\_gte, defaults to \_\_eq if not specified.

Examples:

finished=2014-10-21T19:16:01.020Z

finished\_\_gte=2014-10-21T19:16:01.020Z

finished\_\_lt=2014-10-21T19:16:01.020Z

* status=<task status>

A string value that represents the status of the task. The following is list of statuses:

* queued
* periodic
* pending
* running
* finished

Accepted operators: \_\_eq, \_\_lt, \_\_lte, \_\_gt, \_\_gte, defaults to \_\_eq if not specified.

Example:

status=finished

* name=<task name>

A string value that represents the name of the task. This is generated by the task management system. For ingest and derive tasks, the name is typically the data source name.

Accepted operators: \_\_eq, \_\_lt, \_\_lte, \_\_gt, \_\_gte, defaults to \_\_eq if not specified.

Example:

name=Semios%20Sensors

* username=<user>

A string value that represents the name of the user who initiated the task.

Accepted operators: \_\_eq, \_\_lt, \_\_lte, \_\_gt, \_\_gte, defaults to \_\_eq if not specified.

Example:

username=admin

### Response

{

"\_embedded": {

"ph:task\_history": [

{

"\_id": {

"$oid": "5446ad32e779891099693fdd"

},

"\_links": {

"self": {

"href": "/rest/tms/task\_histories/5446ad32e779891099693fdd"

}

},

"action": "cleanup",

"error": "",

"extra\_props": {},

"finished": "2014-10-21T19:00:17.208Z",

"name": "Semios Retention",

"queued": "2014-10-21T19:00:02.644Z",

"started": "2014-10-21T19:00:02.649Z",

"status": "finished",

"uniquename": "datasource-Semios Retention",

"username": "cron"

},

{

"\_id": {

"$oid": "54469f22e779891099693f9e"

},

"\_links": {

"self": {

"href": "/rest/tms/task\_histories/54469f22e779891099693f9e"

}

},

"action": "cleanup",

"error": "",

"extra\_props": {},

"finished": "2014-10-21T18:00:16.604Z",

"name": "Semios Retention",

"queued": "2014-10-21T18:00:02.756Z",

"started": "2014-10-21T18:00:02.761Z",

"status": "finished",

"uniquename": "datasource-Semios Retention",

"username": "cron"

},

{

"\_id": {

"$oid": "54469112e779891099693f5f"

},

"\_links": {

"self": {

"href": "/rest/tms/task\_histories/54469112e779891099693f5f"

}

},

"action": "cleanup",

"error": "",

"extra\_props": {},

"finished": "2014-10-21T17:00:16.263Z",

"name": "Semios Retention",

"queued": "2014-10-21T17:00:02.658Z",

"started": "2014-10-21T17:00:02.663Z",

"status": "finished",

"uniquename": "datasource-Semios Retention",

"username": "cron"

},

{

"\_id": {

"$oid": "54468bb0e779891099693f45"

},

"\_links": {

"self": {

"href": "/rest/tms/task\_histories/54468bb0e779891099693f45"

}

},

"action": "derive",

"error": "",

"extra\_props": {},

"finished": "2014-10-21T16:37:19.683Z",

"name": "Semios Retention",

"queued": "2014-10-21T16:37:04.488Z",

"started": "2014-10-21T16:37:04.583Z",

"status": "finished",

"uniquename": "datasource-Semios Retention",

"username": "admin"

},

{

"\_id": {

"$oid": "54468bb0e779891099693f44"

},

"\_links": {

"self": {

"href": "/rest/tms/task\_histories/54468bb0e779891099693f44"

}

},

"action": "ingest",

"error": "",

"extra\_props": {

"data\_source\_id": "543d87d8e779897cfc5d3330",

"data\_source\_name": "Semios Retention",

"total\_metasize": 212,

"total\_objects": 1,

"total\_rawsize": 780

},

"finished": "2014-10-21T16:37:04.576Z",

"name": "Semios Retention",

"queued": "2014-10-21T16:37:04.464Z",

"started": "2014-10-21T16:37:04.498Z",

"status": "finished",

"uniquename": "datasource-Semios Retention",

"username": "admin"

},

{

"\_id": {

"$oid": "54468b68e779891099693f40"

},

"\_links": {

"self": {

"href": "/rest/tms/task\_histories/54468b68e779891099693f40"

}

},

"action": "derive",

"error": "",

"extra\_props": {},

"finished": "2014-10-21T16:36:08.528Z",

"name": "Semios Retention",

"queued": "2014-10-21T16:35:52.932Z",

"started": "2014-10-21T16:35:53.071Z",

"status": "finished",

"uniquename": "datasource-Semios Retention",

"username": "admin"

},

{

"\_id": {

"$oid": "54468b68e779891099693f3f"

},

"\_links": {

"self": {

"href": "/rest/tms/task\_histories/54468b68e779891099693f3f"

}

},

"action": "ingest",

"error": "",

"extra\_props": {

"data\_source\_id": "543d87d8e779897cfc5d3330",

"data\_source\_name": "Semios Retention",

"total\_metasize": 212,

"total\_objects": 1,

"total\_rawsize": 780

},

"finished": "2014-10-21T16:35:53.067Z",

"name": "Semios Retention",

"queued": "2014-10-21T16:35:52.907Z",

"started": "2014-10-21T16:35:52.940Z",

"status": "finished",

"uniquename": "datasource-Semios Retention",

"username": "admin"

}

]

},

"\_links": {

"curies": [

{

"href": "http://docs.phemi.com/rels/{rel}",

"name": "ph",

"templated": true

}

],

"ph:find": {

"href": "/rest/tms/task\_histories/{id}",

"templated": true

},

"ph:task\_history": [

{

"href": "/rest/tms/task\_histories/5446ad32e779891099693fdd",

"title": "Semios Retention"

},

{

"href": "/rest/tms/task\_histories/54469f22e779891099693f9e",

"title": "Semios Retention"

},

{

"href": "/rest/tms/task\_histories/54469112e779891099693f5f",

"title": "Semios Retention"

},

{

"href": "/rest/tms/task\_histories/54468bb0e779891099693f45",

"title": "Semios Retention"

},

{

"href": "/rest/tms/task\_histories/54468bb0e779891099693f44",

"title": "Semios Retention"

},

{

"href": "/rest/tms/task\_histories/54468b68e779891099693f40",

"title": "Semios Retention"

},

{

"href": "/rest/tms/task\_histories/54468b68e779891099693f3f",

"title": "Semios Retention"

}

],

"self": {

"href": "/rest/tms/task\_histories?finished\_\_gte=2014-10-21T16:22:26.680Z&name=Semios%20Retention&embed=task\_history"

}

*Query Task History Response Example*

The returned JSON follows the JSON Hypertext Application Language (HAL) standard. The document contains multiple parts identified by the following keys:

**count** - the number of rows found

**\_embedded[“ph:task\_history”]** - an array containing the individual rows returned

**\_links** - query meta data and URI values for individual rows. The individual URIs are restricted to admin users

The following are the HTTP status codes that can be expected depending on the query results:

* **200 OK -** Request successful.
* **202 Accepted -** The task has been accepted but hasn't been finished.
* **401 Unauthorized -** The user is not logged in.
* **403 Forbidden -** The logged in user is not authorized and does not have permission to perform the query operation.
* **404 Not Found -** If the individual task history row does not exist.
* **405 Method Not Allowed -** If a POST, DELETE or PUT is issued instead of a GET.
* **500 Internal Server Error** - An internal system error has occurred.

# 

# 

# Appendix

## DPF Meta Data

The following is a list of meta data generated when a log is processed:

|  |  |
| --- | --- |
| **Name** | **Type** |
| datasource | admin |
| document\_status | admin |
| source\_of\_data | admin |
| timestamp\_of\_creation | admin |
| data:chunks | data |
| encoding | data |
| encryption\_key | data |
| encryption\_type | data |
| hash | data |
| size | data |
| type | data |
| asset\_type | descriptive |
| name | descriptive |
| metadata\_checksum | header |
| process\_information | header |
| system\_identifier | header |
| version | header |

## 

## DPF Processed Tags

The following is a list of the derived data that the DPF extracts from a log message:

|  |  |
| --- | --- |
| **Name** | **Type** |
| id | STRING |
| log\_timestamp | STRING (*see note*) |

*Note:* PHEMI Central will receive and persist log timestamps from Semios as STRINGs (as-is). Semios will send all date/timestamps in ISO 8601 format (e.g., “2014-06-23T19:35:00-07:00”). PHEMI Central will utilize this date format for aggregations. For aggregations, local times are necessary in-order-to align the aggregations with the local time clock (so a day goes from midnight to midnight local time).

## DPF Processed Data

|  |  |
| --- | --- |
| **Name** | **Type** |
| latitude | DOUBLE |
| longitude | DOUBLE |
| mac | STRING |
| timezone | STRING |

|  |  |
| --- | --- |
| **Name** | **Type** |
| ALGAD | LONG |
| ack | LONG |
| app\_version | STRING |
| awake | LONG |
| battery | LONG |
| bridge\_reboot | LONG |
| conf | LONG |
| dev\_installed | LONG |
| device\_model | STRING |
| dew\_point | DOUBLE |
| env\_ambient\_temperature\_celsius | DOUBLE |
| env\_ambient\_temperature\_farhenheit | DOUBLE |
| env\_humidity\_percent | DOUBLE |
| error\_code | LONG |
| errors | LONG |
| event\_error | LONG |
| firmware\_version | STRING |
| humidity | LONG |
| image\_count | LONG |
| ir\_off | LONG |
| ir\_on | LONG |
| last\_error | LONG |
| lithium | LONG |
| model | LONG |
| modem\_reboot | LONG |
| moisture\_0\_15 | LONG |
| moisture\_105\_120 | LONG |
| moisture\_15\_30 | LONG |
| moisture\_30\_45 | LONG |
| moisture\_45\_60 | LONG |
| moisture\_60\_75 | LONG |
| moisture\_75\_90 | LONG |
| moisture\_90\_105 | LONG |
| node\_missed | LONG |
| node\_type | LONG |
| node\_unknown | LONG |
| os\_reboot | LONG |
| os\_version | STRING |
| rain | LONG |
| raw\_ambient\_temperature | LONG |
| raw\_board\_temperature | LONG |
| raw\_humidity | LONG |
| rte\_depth | LONG |
| signal | LONG |
| solar | LONG |
| solar\_radiation | LONG |
| spray\_count | LONG |
| spray\_on | LONG |
| temperature | LONG |
| temperature\_15 | LONG |
| temperature\_45 | LONG |
| temperature\_75 | LONG |
| temperature\_90 | LONG |
| thinkness | LONG |
| ttl\_left | LONG |
| upload\_time | LONG |
| uptime | LONG |
| uv\_radiation | LONG |
| vm\_error | LONG |
| wetness | LONG |
| wet\_bulb | DOUBLE |
| wind\_direction | LONG |
| wind\_speed | LONG |

## Semios Log Samples

There are different types of logs that the DPF needs to understand in order to perform the correct processing. In particular, the DPF needs to assign an identifier to the device where the log came from so that aggregations and queries can be performed against the processed log data.

Each of the log types have a different set of data and an identifier to specify the device the log originated from, such as gateway, sensor, node. The following are examples of the different types of logs.

### Semios Log Structure

A Semios log structure contains the following keys in the JSON data structure:

* version: Contains a message version identifier. This allows a DPF to differentiate between different message formats in the future.
* id: Contains the identifier of the Semios device.
* data: Contains the set of data generated by the Semios device.
* ~tag: Specifies a set of tags that is applied to the ingested data. Tags can be queried to quickly return sets of data.

### Gateway

The following example shows a gateway:

{  
 "~tags": {  
 "timezone": "America\/Vancouver",  
 "mac": "001C2C1AD6048652",  
 "longitude": -122.29351,  
 "latitude": 49.00657  
 },  
 "log\_timestamp": "2014-04-24T08:20:00-07:00",  
 "id": "gateway\_52cc5addc6c0872b2d260ab3",  
 "data": {  
 "uptime": 0,  
 "firmware\_version": "2.1.1",  
 "env\_ambient\_temperature\_farhenheit": 32,  
 "env\_ambient\_temperature\_celsius": 0,  
 "device\_model": "node\_9",  
 "conf": 0,  
 "bridge\_reboot": 0,  
 "battery": 320,  
 "awake": 0,  
 "last\_error": 0,  
 "lithium": 232,  
 "modem\_reboot": 0,  
 "node\_missed": 0,  
 "node\_unknown": 0,  
 "os\_reboot": 0,  
 "raw\_ambient\_temperature": 0,  
 "solar": 319  
 }  
}

* The “data” key contains the set of data generated by the Semios sensors.
* The “id” key contains the gateway identifier.
* The “log\_timestamp” key contains the timestamp when the log is generated.
* The “~tags” key specify a set of tags that is applied to the ingested data. Tags can be queried to quickly return sets of data.

### Generic Node

The following example shows a generic node:

{  
 "data": {  
 "ack": 14,  
 "app\_version": "2.0.51",  
 "awake": 5999,  
 "battery": 273,  
 "dev\_installed": 64,  
 "device\_model": "E10",  
 "event\_error": 0,  
 "lithium": 231,  
 "node\_type": 3,  
 "os\_version": "2.4.34",  
 "raw\_board\_temperature": 207,  
 "rte\_depth": 0,  
 "signal": 52,  
 "solar": 581,  
 "ttl\_left": 4,  
 "vm\_error": 0  
 },  
 "id": "node\_537bd50bfd8978450f702b73",  
 "log\_timestamp": "2014-10-11T17:20:00-07:00",  
 "~tags": {  
 "latitude": 49.00651,  
 "longitude": -122.29065,  
 "mac": "001C2C1AD60561CE",  
 "timezone": "America\/Vancouver"  
 }  
}

* The “data” key contains the set of data generated by the Semios sensors.
* The “id” key contains the node identifier.
* The “log\_timestamp” key contains the timestamp when the log is generated.
* The “~tags” key specify a set of tags that is applied to the ingested data. Tags can be queried to quickly return sets of data.

### Semios Sensor

The following example shows a Semios sensor that comes with a node and sensor identifier:

{  
 "data": {  
 "error\_code": 0,  
 "errors": 0,  
 "image\_count": 7,  
 "upload\_time": 48  
 },  
 "id": "sensor\_5356a89d2fea215cb9668ea6",  
 "log\_timestamp": "2014-04-30T09:00:00-07:00",  
 "~tags": {  
 "latitude": 49.006721,  
 "longitude": -122.293657,  
 "mac": "001C2C1AD604B4CC",  
 "timezone": "America\/Vancouver"  
 }  
}

* The “data” key contains the set of data generated by the Semios sensors.
* The “id” key contains the sensor identifier.
* The “log\_timestamp” key contains the timestamp when the log is generated.
* The “~tags” key specify a set of tags that is applied to the ingested data. Tags can be queried to quickly return sets of data.

### 

### DPF Processing

The DPF also calculates dew-point and wet-bulb based on standard formulas[[1]](#footnote-1). For all calculations, we used the following constants:

\begin{align}a&=6.112\ \mathrm{millibar};\quad\;b&= 17.67;\quad\;c&= 243.5^\circ \mathrm{C};\end{align}

For the dew point, we used the Magnus Formula:

1. From the temperature measurement T, compute
2. From and the relative humidity, compute
3. Compute dew point from

Where

= measured temperature

saturation water vapour pressure

”actual” water vapour pressure

= measured relative humidity

For the wetbulb temperature , we used the approximate relation

To compute the value of that satisfies this relation, we used an iterative method[[2]](#footnote-2). Given measurements of temperature, relative humidity, and air pressure, the initial estimate for the wetbulb temperature is chosen as the dewpoint temperature calculated for the measured temperature.

Then, the corresponding value of the pressure is calculated with the above equation. The resulting value is compared to the vapour pressure at the measured temperature. For each iteration, we add or subtract a fixed increment to the wet bulb estimate and check the pressure difference again. Any time the sign flips on the pressure difference, we assume we are getting closer to the correct temperature, and decrease the size of the temperature increment and continue until the pressure difference is within the specified tolerance.

This method can be more concisely expressed with the following pseudocode:

inputs:

T = temperature

RH = relative humidity

P\_air = air pressure

T\_wetbulb(T, RH, P\_air):

{

increment = 10

tolerance = 0.005

T\_guess = T\_dew\_point

compute P\_vapour, pressure at T using above formulas

Pressure\_difference = 1

While(pressure\_difference > tolerance)

{

compute P\_vapour\_guess at T\_guess using

P\_vapour\_guess = Ps(T\_guess) - P\_air\*(T - T\_guess)\* 0.00066 \*

(1.0 + (0.00115 \*T\_guess))

Pressure\_difference = P\_vapour - P\_vapour\_guess

if sign is opposite sign from last iteration then

increment = increment/10

end if

T\_guess = T\_guess + increment \* sign

}

return T\_guess

}

If the air pressure measurement is not provided, a wetbulb temperature is estimated if the dewpoint is above 0°C by approximating the wetbulb pressure:

Then computing the dewpoint temperature with this approximate pressure as the input.

## 

## Derivation Completion

There is currently no REST interface that will let the user know when a derivation of a specific ingested data has been completed. The following is the known workaround to determine when a derivation of a specific ingested data is complete:

* Use the [**Query Sensor Data**](#h.q1wsbqp20m80) REST call to find the specific ingested data meta data and capture the meta|admin:timestamp\_of\_creation.
* Use the [**Query Task History**](#h.aeuk76e8oa0e) REST call specifying:
  + That the finished parameter is less than the *meta:admin:timestamp\_of\_creation*
  + The action is *derive*
  + Specify the name of the task which is the *name of the data source*

For example:

/rest/tms/task\_histories?finished\_\_lt=2014-10-21T16:22:26.680Z&name=Semios%20Sensors&action=derive&embed=task\_history

If there is a derive task that finished after the time the ingested data was created in the data source it was ingested into, then the derived data has been created.

1. http://en.wikipedia.org/wiki/Dew\_point [↑](#footnote-ref-1)
2. view-source:http://www.srh.noaa.gov/epz/?n=wxcalc\_rh?n=wxcalc\_rh [↑](#footnote-ref-2)