IoT Platform

Software/Hardware Integration Lab at UFSC

IoT Platform

Table of contents

- IoT Platform
- 1. Prolog
- 2. IoT Platform Overview
  - 2.1. SmartData
  - 2.2. SmartData Series
  - 2.3. Authentication and Authorization
  - 2.4. Usefull SmartData Units
- 3. REST API for Stationary Objects
  - 3.1. Data Querying
    - 3.1.1. Data Aggregation
    - 3.1.2. Fault Injection
    - 3.1.3. Downsampling
  - 3.2. Series Creation
    - 3.2.1. Series Types and Modes
    - 3.2.2. Series Status
    - 3.2.3. Meaninful Types and Status
      - Time-Triggered Series
      - Event-Driven Series
  - 3.3. Data Insertion
    - 3.3.1. Bulk Data Insertion
    - 3.3.2. Series Documentation
  - 3.4. Series Termination
  - 3.5. AI Workflows
    - 3.5.1. Persistency
    - 3.5.2. Loading previous data
    - 3.5.3. Inserting new data
    - 3.5.4. Notifications
  - 3.6. Data Searching
  - 3.7. Response codes
  - 3.8. Plotting a dashboard with Grafana
- 4. Binary API for SmartData Version 1.1
  - 4.1. Create series (Binary)
  - 4.2. Insert data (Binary)
    - 4.2.1. Binary Multi SmartData
  - 4.3. Version format
- 5. Client Authentication
- 6. Scripts
  - 6.1. C++
    - 6.1.1. Get Script Example
  - 6.2. Python
    - 6.2.1. Get Script Example
    - 6.2.2. Put Script Example
  - 6.3. R
    - 6.3.1. Get Script Example
- 7. Troubleshooting

- 7.1. TLS support for Post-Handshake Authentication
- Review Log

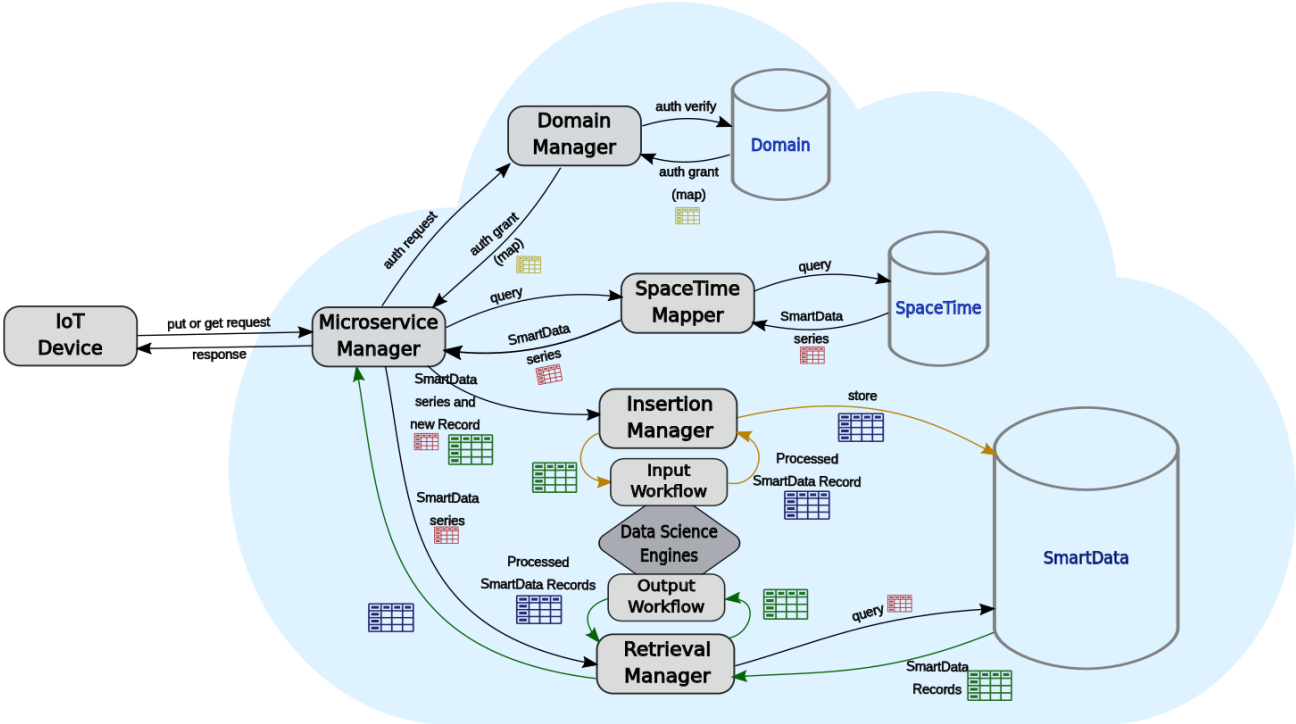
1. Prolog

LISHA's IoT Platform is an effort to support projects investigating the application of Data Science algorithms in the realm of the Internet of Cyber-Physical Systems. This document is a technical documentation of the Platform aimed at supporting real users. Before reading it, or if you just want to get a glimpse of it, you might want to visit the Platform's site for an overview of its architecture and the underlying technology. LISHA's IoT Platform is based on EPOS SmartData, so you might also want to take a look at it before continuing with this document. Finally, if you want to contribute to the development of the Platform, there is also a Guide about its Internals.

The **IoT Platform** is organized around a set of *microservices* relating to storage, processing, aggregation and visualization of data widely used in LISHA projects. The **SmartData** format tags data with spatial location, high definition temporal tagging, authentication and semantics. The microservice composition is performed through specific workflows for each application. A library of pre-processing, filters, feature selection, feature extraction, transformation, aggregation and machine learning algorithms effectively enables the creation of these workflows. The data can also be recovered through origin, time, space or semantics filtering. Workflows can also be constructed by the same libraries. The non-relational database and the workflow execution containers have been designed for scalability on the high-availability platform maintained by SETIC/UFSC (our data center). The platform also contains a real-time data visualizer, which, after configuration, shows the data through a website for monitoring and functional verification. The **Microservice Manager** acts as a front-end to IoT devices, IoT gateways, Data Analytics services, and a Visualization Engine. Microservices requests are first handled by the **Domain Manager**, which is responsible for mapping SmartData sets to projects and implementing certificate and password-based authentication (both for users and devices), access control, and secure communication. The **SpaceTime Mapper** is responsible for mapping regions of Space and Time to the associated SmartData stored or to be stored in the Platform. The **Insertion** and **Retrieval** managers are responsible for running Data Science algorithms on the SmartData flowing into and out of the Platform.

Da mesma forma, os dados podem ser recuperados em função da origem, do tempo, do espaço ou da semântica. A recuperação de dados pode também utilizar workflows construídos com as mesmas bibliotecas. Tanto o banco de dados não relacional quanto os containers de execução de workflows foram projetados para escalar sobre a plataforma de alta disponibilidade mantida pela SETIC / UFSC (nosso data center). A plataforma também conta com um visualizador de dados em tempo real que, após ser configurado, exibe os dados através de um website para monitoramento e verificação do seu funcionamento.

2. IoT Platform Overview



IoT Platform Overview

Each SmartData stored in the Platform is a data point in a SmartData time series, and it is characterized by a version, a unit, and the SpaceTime coordinates of origin (that is, where and when the SmartData was produced, created, captured, sampled, etc.).

2.1. SmartData

The SmartData stored and processed by the platform have the following structure:

SmartData

version	unit	value	uncertainty	x	y	z	t	dev	signature

- `version`: the SmartData version:
  - `"1.1"`: version 1, Stationary (.1), representing data from a device that is not moving;
  - `"1.2"`: version 1, Mobile (.2), representing data from a device that is moving;
- `unit`: the type of the SmartData (see the SmartData documentation and typical units);
- `value`: the data value (e.g., the temperature measured by a thermometer);
- `uncertainty`: a measure of uncertainty, usually transducer-dependent, expressing Accuracy, Precision, Resolution, or a combination thereof;
- `x`, `y`, `z`: the absolute coordinates of the location where the data originated;
- `t`: the time instant at which the data originated (in UNIX epoch microseconds).
- `dev`: a disambiguation identifier for multiple transducers of the same Unit and space-time coordinates (e.g., 3-axis accelerometer), "0" otherwise (i.e., if a single transducer is present);
- `signature`: a cryptographic identifier for mobile devices producing SmartData (only for version 1.2 / mobile).

SmartData can be represented in JSON as follows:

{ "version" : unsigned char "unit" : unsigned long "value" : double "uncertainty" : unsigned long "x" : long "y" : long "z" : long "t" : unsigned long long "dev" : unsigned long "signature": string } assuming the following sizes the types used in this document:

- `char`: 1 byte;
- `short`: 2 bytes;
- `long`: 4 bytes;
- `long long` : 8 bytes;

2.2. SmartData Series

The SmartData Series stored and processed by the platform have the following structure:

SmartData Series											
version	unit	x	y	z	r	t0	tf	period	count	event	workflow

- `version`: the version of the SmartData in the series (a series does not contain mixed versions SmartData);
- `unit`: the type of the SmartData in the series (see the SmartData documentation and typical units);
- `x`, `y`, `z`: the absolute coordinates of the center of the sphere containing the data points in the series (from a SmartData Interest);
- `r`: the radius of the sphere containing the data points in the series (initially from a SmartData Interest; is automatically adjusted with data point insertion);
- `t0`: (optional) a timestamp representing the time in which the series begins, in UNIX epoch microseconds;
- `tf`: (optional) a timestamp representing the time in which the series ends, in UNIX epoch microseconds;
- `type`: (optional) 'TTH' specifies high-frequency data (KHz sampling) with fixed sampling rate. Some storage optimizations are applied.
- `period`: (optional) only defined for time-triggered series representing the period of data points (usually from a SmartData Interest, but also from method `create`);
- `count`: (optional) specifies the number of data points to be captured before closing the series (-+tf+- is captured when `count` data points are collected);
- `event`: (optional) a SmartData expression designating an event that marks the beginning of the series (-+tf+- is derived from the time the expression becomes/became true, representing the occurrence of "event");
- `workflow`: (optional) specify server-side algorithms to be applied on the series (see AI Workflow Section);
    - input workflows are executed during insert operations (method `put`) to preprocess data, run machine learning algorithms, fix data points following a measurement error, generate notifications and even interact with other series.
    - output workflows are executed along with query operations (method `get`) to post process the data, for instance, performing aggregations or transformations.

SmartData series are classified based on the operation mode of the associated SmartData, either as **Time-Triggered** or **Event-Driven**. At the time of creation, series associated with time-triggered SmartData must define a `period`, whereas those not defining this attribute are assumed to be event-driven. The beginning of a series can be specified by time (giving `t0`), `event`, or manually (by not

providing `t0`, which is then assumed to be the current time). Therefore, the beginning of a time-triggered series can be an event, and the series will remain a time-triggered series. Similarly, event-driven series can start at a given time. The end of a series can be specified by time (giving `tf`), `event`, manually (with the method `finish`, which makes `tf` equal to the current time), or in terms of event counting (by giving `count`). Events are expressed as internal (stored in the platform) or external SmartData, and arithmetic and logical operators.
A SmartData Series can be represented in JSON as follows:

```
"Series" : Object { "version" : unsigned char "unit" : unsigned long "x" : long "y" : long "z" : long "r" : unsigned long "t0" : unsigned long long "tf" : unsigned long long "type" : char[3] "period" : unsigned long "count" : unsigned long "event" : string "uncertainty" : unsigned long "workflow" : unsigned long }
```

2.3. Authentication and Authorization

API methods require Authentication and Authorization, which is usually done based on digital certificate hierarchies controlled by the Platform at connection-time. This kind of primary authentication is part of the RESTfull API, and, therefore, it is not represented as JSON in any service. However, in some rare cases, access without a digital certificate can be granted based on `Credentials` appended to API method invocations and expressed in this format:

```
"Credentials" : Object { "domain" : string "username" : string "password" : string }
```

- `domain`: the domain the SmartData belongs to (usually a project or a project perspective; defaults to "public");
- `username`: a username to be used to validate access to the requested domain;
- `password`: a password used to authenticate the user requesting access to a domain.

2.4. Usefull SmartData Units

The formation rules for SmartData Units are available in the EPOS user guide, and some useful units are listed here.

3. REST API for Stationary Objects

3.1. Data Querying

Method: POST
URL: https://iot.lisha.ufsc.br/api/get.php
Body:

```
"Series" : Object { "version" : unsigned char "unit" : unsigned long "x" : long "y" : long "z" : long "r" : unsigned long "dev" : unsigned long "t0" : unsigned long long "tf" : unsigned long long "type": char[3] "period": unsigned long "workflow" : unsigned long }
```

SmartData querying is a space-time operation that is not limited or even bound to specific devices. The geographic search engine built in the Platform will promptly collect data from several devices within the specified space-time region while processing the query. The definition of `dev` in this operation must be interpreted as a filter: if multiple SmartData exists in the designated space-time

region originated from the same coordinates (i.e. `(unit, x, y, z, t)`), then only those matching `dev` are included.

The `workflow` is used to specify a post-processing function for the query by selecting an output workflow.

3.1.1. Data Aggregation

While querying data, an aggregation function can be invoked on the resulting data by appending the following structure to a `series` object in the body of a query:

```
"Aggregator" : Object { "name" : string, "parameter' : float, "offset' : unsigned long, "lenght' : unsigned long, "spacing' : unsigned long }
```

The time-related attributes `range`, `delay`, and `spacing` are expressed in *us*. Only the `name` attribute is required, other attributes are optional.
More sophisticated aggregation functions can be modeled as output workflows. An aggregator can also be combined with an output workflow, where the aggregator runs first (i.e., the workflow will handle already aggregated data).
The following is a list of the currently available aggregators. To apply one of the following aggregators simply set the `name` attribute accordingly.

- **min**: returns the minimum value among the SmartData selected by the query;
- **max**: returns the maximum value among the SmartData selected by the query;
- **mean**: returns the mean of the set of values resulting from the query;
- **filter**: filters the SmartData selected by the query, returning only those whose value is larger than `parameter` **and** smaller than `offset`, eventually returning {} if no SmartData matching the criterion is found; if any of the `parameter` or `offset` is omitted, then its corresponding criterion is ignored;
- **higherThan**: filters out SmartData whose value is less than `parameter`, eventually returning {} if no SmartData matching the criterion is found;
- **confidence**: the value of the SmartData matching the query is replaced by each SmartData confidence.

3.1.2. Fault Injection

Some aggregators have been designed to inject faults on the results of SmartData Series queries. They use the following optional attributes:

- `offset`: offset in *us* from the beginning of the query results to the first SmartData to undergo fault injection;
- `length`: length of the time window of fault injection, in *us*, starting at `offset`;
- `spacing`: time window in *us* to wait after `offset` + `length` before reapplying the fault injector.

The available fault injectors are:

- **drift**: applies a drift of `parameter` to the values of the SmartData selected by the query. The drift varies according to the number of samples it has been applied to following this formula: ***drift*** = `parameter` * *i*.
- **stuckAt**: the SmartData selected by the query in the time windows defined by [`offset`, `length`] spaced by `spacing` have their values set to the value of the first SmartData in the interval.
- **constantBias**: sums `parameter` to the value of each SmartData selected by the query in the

time windows defined by [`offset`, `lenght`] spaced by `spacing`.
- **constantGain**: each SmartData selected by the query has its value multiplied by `parameter`, considering the windowing mechanism described earlier.

3.1.3. Downsampling

When only sparse data samples of some long or dense data series is needed, the **downSampling** aggregator can specify the spacing between the original data points that shall be returned. Uses only one attribute:

- `spacing`: the number of samples skipped between each returned value.

The series' original attributes (as `period`) aren't changed.
In the example below, samples 0, 100, 200, 3000... will be returned.

```
"series": { "version":"1.1", ... }, "aggregator":{"name":"downSampling", "spacing":100}
```

3.2. Series Creation

Method: POST
URL for create: https://iot.lisha.ufsc.br/api/create.php
Body:

```
"Series" : Object { "version" : unsigned char "unit" : unsigned long "x" : long "y" : long "z" : long "r" : unsigned long "dev" : unsigned long "t0" : unsigned long long "tf" : unsigned long long "period" : unsigned long "count" : unsigned long "event" : string "uncertainty" : unsigned long "workflow" : unsigned long }
```

This method creates a SmartData Series if there is no other existing series that already encompasses the designated unit `(unit)` and space-time region `(x, y, z, r, dev, t0, tf)` for the same operating mode (e.g., time-triggered or event-driven). If the new series intersects existing ones but is not fully contained in any of them, a new series is created with the smallest space-time region that contains both the given space-time region and **all** the preexisting series intersecting with that region. Thus, this method can be used to merge series **irreversibly** and must be used with extreme caution (it can also be very expensive).

All the series in a domain associated with the same unit **must either not use an input workflow or use the same input workflow**, thus avoiding multiple insertions of the same SmartData. The `create` method follows the execution flow presented below:

3.2.1. Series Types and Modes

The method `create` can be used to create different types of series with quite different operating modes. As previously stated, series crated with a defined `period` are assumed to contain time-triggered SmartData, whereas those not defining this attribute are assumed to contain event-driven SmartData. Additional information about the operating regimen of a series can be given through the attributes `t0`, `tf`, `count`, `event`, and `uncertainty`.

For sanity checking and documentation purposes, a series can have a starting time different from its creation time. The starting time can be explicitly specified using attribute `t0`. It can also be set implicitly using the time an event occurs. This event, if set, is documented using the `event` attribute. Therefore, if `event` is given but not `t0`, then the starting time of the series will be set by the

timestamp in the first SmartData (i.e., *series*[0]) inserted in the series (which is **assumed** to be conditioned by `event`). If neither `t0` nor `event` are given, then the starting time of the series is assumed to be the moment in which `create` was called. Note that specifying an `event` for a time-triggered series does not make it an event-driven one, nor does the association of a timestamp `t0` with an event-driven series make it time-triggered. It is solely the presence (or absence) of attribute `period` that characterizes the series as time-triggered or event-driven. Inserting data before `t0` for a series that has a defined `t0` is an error.

Similarly, the end of a series can be specified by giving `tf` along with `create` or manually through the invocation of the `finish` method, which sets `tf` to the current time. An `event` can be specified at creation-time to document the ending of a series. It can also be supplied along with `finish`. Trying to insert SmartData in a series after `tf` will return an error condition.

3.2.2. Series Status

A SmartData Series can assume the following status during its life cycle:

Status	Description
Waiting	the series is created, but `t0` is not yet set, or it is set but not yet reached
Open	data for the series is being collected and `tf` is not yet set or it is set but not yet reached
Closed	`tf` is defined and reached, so no further insertions are allowed
Defective	the series should be in status closed, but data counting does not match the specification

3.2.3. Meaninful Types and Status

Time-Triggered Series

TT-t_0.t_f: begin and end set at creation		
JSON	**Attributes**	**Status**
□□□□□□□□ "Series" : Object { "version" : 1.1 "unit" : unsigned long "x" : long "y" : long "z" : long "r" : unsigned long "t0" : unsigned long long "tf" : unsigned long long "period" : unsigned long "uncertainty" : unsigned long "workflow" : unsigned long }	**p** = `period` **t_0** = `t0` **t_f** = `tf` **c** = (t_f - t_0) / p **n** = *current data count* **now** = *current time*	**Waiting** : now < t_0 **Open**: t_0 <= now <= t_f **Closed**: (now > t_f) ∧ (n >= c) **Defective**: (now > t_f) ∧ (n < c)

TT-t_0.c: begin set at creation and end set by count		
JSON	**Attributes**	**Status**

TT-t_0.c: begin set at creation and end set by count

□□□□□□□□
"Series" : Object { "version" : 1.1 "unit" : unsigned long "x" : long "y" : long "z" : long "r" : unsigned long "t0" : unsigned long long "period" : unsigned long "count" : unsigned long "uncertainty" : unsigned long "workflow" : unsigned long } }

p = `period`
t_0 = `t0`
c = `count`
t_f = t_0 + p * c
n = *current data count*
now = *current time*

Waiting : now < t_0
Open: t_0 <= now <= t_f
Closed: (now > t_f) ∧ (n >= c)
Defective: (now > t_f) ∧ (n < c)

TT-t_0.f: begin set at creation and end set by `finish`

JSON

□□□□□□□□
"Series" : Object { "version" : 1.1 "unit" : unsigned long "x" : long "y" : long "z" : long "r" : unsigned long "t0" : unsigned long long "period" : unsigned long "event" : string "uncertainty" : unsigned long "workflow" : unsigned long } }

Attributes

p = `period`
t_0 = `t0`
t_f = `finish`.t
c = t_f → (t_f - t_0) / p
n = *current data count*
now = *current time*

`finish`.event → `series`.event = `finish`.event

Status

Waiting : now < t_0
Open: t_0 <= now
Closed: t_f ∧ (n >= c)
Defective: t_f ∧ (n < c)

TT-e.t_f: begin set by data and end set at creation

JSON

□□□□□□□□
"Series" : Object { "version" : 1.1 "unit" : unsigned long "x" : long "y" : long "z" : long "r" : unsigned long "tf" : unsigned long long "period" : unsigned long "event" : string "uncertainty" : unsigned long "workflow" : unsigned long } }

Attributes

p = `period`
t_f = `tf`
t_0 = *series*[0].t
c = t_0 → (t_f - t_0) / p
n = *current data count*
now = *current time*

Status

Waiting : ¬t_0
Open: t_0 ∧ (t_0 <= now <= t_f)
Closed: t_0 ∧ (now > t_f) ∧ (n >= c)
Defective: (¬t_0 ∧ (now > t_f)) ∨ (t_0 ∧ (n < c))

TT-e.c: begin set by data and end by count

JSON **Attributes** **Status**

TT-e.c: begin set by data and end by count

"Series" : Object { "version" : 1.1 "unit" : unsigned long "x" : long "y" : long "z" : long "r" : unsigned long "period" : unsigned long "count" : unsigned long "event" : string "uncertainty" : unsigned long "workflow" : unsigned long } }	**p** = `period` **c** = `count` **t_0** = *series*[0].t **t_f** = $t_0 \rightarrow t_0 + p * c$ **n** = *current data count* **now** = *current time*	**Waiting** : $\neg t_0$ **Open**: $t_0 \wedge (t_0 <= now <= t_f)$ **Closed**: $t_0 \wedge (now > t_f) \wedge (n >= c)$ **Defective**: $t_0 \wedge (now > t_f) \wedge (n < c)$

TT-e.f: begin set by data and end by `finish`

JSON	Attributes	Status
"Series" : Object { "version" : 1.1 "unit" : unsigned long "x" : long "y" : long "z" : long "r" : unsigned long "period" : unsigned long "uncertainty" : unsigned long "workflow" : unsigned long } }	**p** = `period` **t_0** = *series*[0].t **t_f** = `finish`.t **c** = $t_0 \wedge t_f \rightarrow (t_f - t_0) / p$ **n** = *current data count* **now** = *current time*	**Waiting** : $\neg t_0$ **Open**: $t_0 \wedge \neg t_f$ **Closed**: $t_0 \wedge t_f \wedge (n >= c)$ **Defective**: $t_0 \wedge t_f \wedge (n < c)$

Event-Driven Series

ED-t_0.t_f: begin and end set at creation

JSON	Attributes	Status
"Series" : Object { "version" : 1.1 "unit" : unsigned long "x" : long "y" : long "z" : long "r" : unsigned long "t0" : unsigned long long "tf" : unsigned long long "uncertainty" : unsigned long "workflow" : unsigned long }	**t_0** = `t0` **t_f** = `tf` **n** = *current data count* **now** = *current time*	**Waiting** : $now < t_0$ **Open**: $t_0 <= now <= t_f$ **Closed**: $(now > t_f) \wedge (n > 0)$ **Defective**: $(now > t_f) \wedge (n = 0)$

ED-t_0.c: begin set at creation and end set by count

JSON	Attributes	Status

ED-t_0.c: begin set at creation and end set by count

JSON	Attributes	Status
"Series" : Object { "version" : 1.1 "unit" : unsigned long "x" : long "y" : long "z" : long "r" : unsigned long "t0" : unsigned long long "count" : unsigned long "uncertainty" : unsigned long "workflow" : unsigned long } }	**t_0** = `t0` **c** = `count` **t_f** = *series*[c].t **n** = *current data count* **now** = *current time*	**Waiting** : now < t_0 **Open**: $\neg t_f \wedge$ (t <= now) **Closed**: $t_f \wedge$ (n >= c) **Defective**: $t_f \wedge$ (n < c)

ED-t_0.f: begin set at creation and end set by `finish`

JSON	Attributes	Status
"Series" : Object { "version" : 1.1 "unit" : unsigned long "x" : long "y" : long "z" : long "r" : unsigned long "t0" : unsigned long long "event" : string "uncertainty" : unsigned long "workflow" : unsigned long } }	**t_0** = `t0` **t_f** = `finish`.t **n** = *current data count* **now** = *current time*	**Waiting** : now < t_0 **Open**: $\neg t_f \wedge$ (t_0 <= now) **Closed**: $t_f \wedge$ (now > t_f) $\wedge$ (n >= 0) **Defective**: $t_f \wedge$ (now > t_f) $\wedge$ (n = 0)

ED-e.t_f: begin set by data and end set at creation

JSON	Attributes	Status
"Series" : Object { "version" : 1.1 "unit" : unsigned long "x" : long "y" : long "z" : long "r" : unsigned long "tf" : unsigned long long "event" : string "uncertainty" : unsigned long "workflow" : unsigned long } }	**t_f** = `tf` **t_0** = *series*[0].t **n** = *current data count* **now** = *current time*	**Waiting** : $\neg t_0$ **Open**: $t_0 \wedge$ (t_0 <= now <= t_f) **Closed**: $t_0 \wedge$ (now > t_f) $\wedge$ (n > 0) **Defective**: (now > t_f) $\wedge$ (n = 0)

ED-e.c: begin set by data and end by count

JSON	Attributes	Status

ED-e.c: begin set by data and end by count		
"Series" : Object { "version" : 1.1 "unit" : unsigned long "x" : long "y" : long "z" : long "r" : unsigned long "count" : unsigned long "event" : string "uncertainty" : unsigned long "workflow" : unsigned long } }	**c** = `count` $\mathbf{t_0}$ = *series*[0].t $\mathbf{t_f}$ = *series*[c].t **n** = *current data count* **now** = *current time*	**Waiting** : $\neg t_0$ **Open**: $t_0 \wedge \neg t_f$ **Closed**: $t_0 \wedge t_f$

ED-e.f: begin set by data and end by `finish`		
JSON	**Attributes**	**Status**
"Series" : Object { "version" : 1.1 "unit" : unsigned long "x" : long "y" : long "z" : long "r" : unsigned long "event" : string "uncertainty" : unsigned long "workflow" : unsigned long } }	$\mathbf{t_0}$ = *series*[0].t $\mathbf{t_f}$ = `finish`.t **n** = *current data count* **now** = *current time* `finish`.event → `series`.event = `finish`.event	**Waiting** : $\neg t_0$ **Open**: t_0 **Closed**: $t_0 \wedge t_0 \wedge (n > 0)$ **Defective**: $t_0 \wedge t \wedge (n = 0)$

3.3. Data Insertion

Method: POST
URL: https://iot.lisha.ufsc.br/api/put.php
Body:

```
"SmartData" : Array [ { "version" : unsigned char "unit" : unsigned long "value" : double "uncertainty" : unsigned long "x" : long "y" : long "z" : long "t" : unsigned long long "dev" : unsigned long } ]
```

This method is used to insert SmartData into the existing SmartData Series. The series is implicitly determined from the given unit and space-time coordinates. If the data point does not fit in any existing series, then the operating fails, and error `400` is returned. Multiple data points can be inserted at once (hence the Array in the JSON).

3.3.1. Bulk Data Insertion

To optimize the processing of multiple SmartDate originated at the same location **(i.e. r=0)**, the `put` can receive alternative payloads (body). Currently, the following structures are supported:

Periodic SmartData with Constant Uncertainty
If the multiple values being inserted have a constant time rate (e.g., they result from a regular periodic sampling), the **period** attribute can be used in the header, and the offset is omitted in the data points. Additionally, if a constant **uncertainty** — possibly 0 — is to be assigned to all data

points, it can also be specified in the header.

Body: MultiValueSmartData

```
"MultiValueSmartData" : Object { "version" : unsigned char "unit" : unsigned long "x" : long "y" : long "z" : long "r" : 0 "t0" : unsigned long long "dev" : unsigned long "type" : char(3), // 'TTH', 'TTL', 'ED'. 'OLD' assumed if not present "period" : unsigned long, "uncertainty" : unsigned long // OPTIONAL: if given, then ommit it in data points "period" : unsigned long // OPTIONAL: if given, then ommit offset in data points "datapoints": Array [ { "offset : unsigned long // OPTIONAL, not used if period is informed in the header "value" : double "uncertainty" : unsigned long // OPTIONAL, not used if informed in the header } ] }
```

MultiDeviceSmartData
When a node or datalogger is regularly capturing several variables of the same type, with the same SI unit, a `MultiDeviceSmartData` can be used to spare the space-time coordinates.
Body: MultiDeviceSmartData

```
"MultiDeviceSmartData" : Object { "version" : unsigned char "unit" : unsigned long "x" : long "y" : long "z" : long "r" : 0 "t0" : unsigned long long "datapoints": Array [ { "offset : unsigned long "value" : double "dev" : unsigned long; "uncertainty" : unsigned long } ] }
```

Note that the **device** field must start from 0 since it is only used for disambiguation for multiple same-type sensors.

MultiUnitSmartData
Allows multiple variables from a single space-time coordinate to be inserted without repeating such coordinate.

Body: MultiUnitSmartData

```
"MultiUnitSmartData" : Object { "version" : unsigned char "x" : long "y" : long "z" : long "r" : 0 "t0" : unsigned long long "datapoints": Array [ { "unit" : unsigned long "offset : unsigned long "value" : double "dev" : unsigned long; "uncertainty" : unsigned long } ] }
```

3.3.2. Series Documentation

The information present in the series creation data, determining the data type (SI Unit), position, and time interval, isn't enough to completely describe the meaning of data. Therefore, the API was extended to support the insertion and the querying of a human-readable description of the data. The API offers two methods: `describe` and `list`. To describe the data of a series, the `describe` method accepts a JSON name **series_description**. If the **dev** field contains 0, the description applies to all devices of the defined **unit** at this position. Information about a specific device, or specific devices with different **type** and **period** fields can also be inserted.

Method: POST
URL: https://iot.lisha.ufsc.br/api/describe.php
BODY:

"series_description" : { "version" : unsigned char "unit" : unsigned long "x" : long, "y" : long, "z" : long, "type" : char(3), "period" : unsigned long, "dev" : unsigned long, "description" : string }

The `describe` method also supports an array of descriptions, as illustrated below:

```
"series_descriptions" : [ { "version" : unsigned char, "unit" : usigned long, "x": long, ....,"description": string}, { "version" : unsigned char, "unit" : usigned long, "x": long, ....,"description": string}, ....]
```

The `list` method supports queries the descriptions of devices of a specific region. It uses the `series` JSON. Several fields are optional for this method, The **unit**, **dev**, **type** and **period** parameters can be used to filter specific information.

Method: POST
URL: https://iot.lisha.ufsc.br/api/list.php
BODY:

```
"series" : { "version" : unsigned char, "unit" : unsigned long, "x" : long, "y" : long, "z" : long, "r" : long, "dev": unsigned long, "type" : char(3), "period" : unsigned long, }
```

Both methods, like all others, require authentication through certificate or username/password and are restricted to the user's domain.

3.4. Series Termination

Method: POST
URL: https://iot.lisha.ufsc.br/api/finish.php
Body:

```
"Series" : Object { "version" : 1.1 "unit" : unsigned long "x" : long "y" : long "z" : long "r" : unsigned long "tf" : unsigned long long "event" : string "uncertainty" : unsigned long }
```

This method is used to finish a SmartData Series. It adjusts the series final time stamp t_f and, if `event` is given, this method concatenates it with the previous value of that attribute. Inserting new SmartData by invoking `put` after having invoked `finish` is an error and will return `400`.

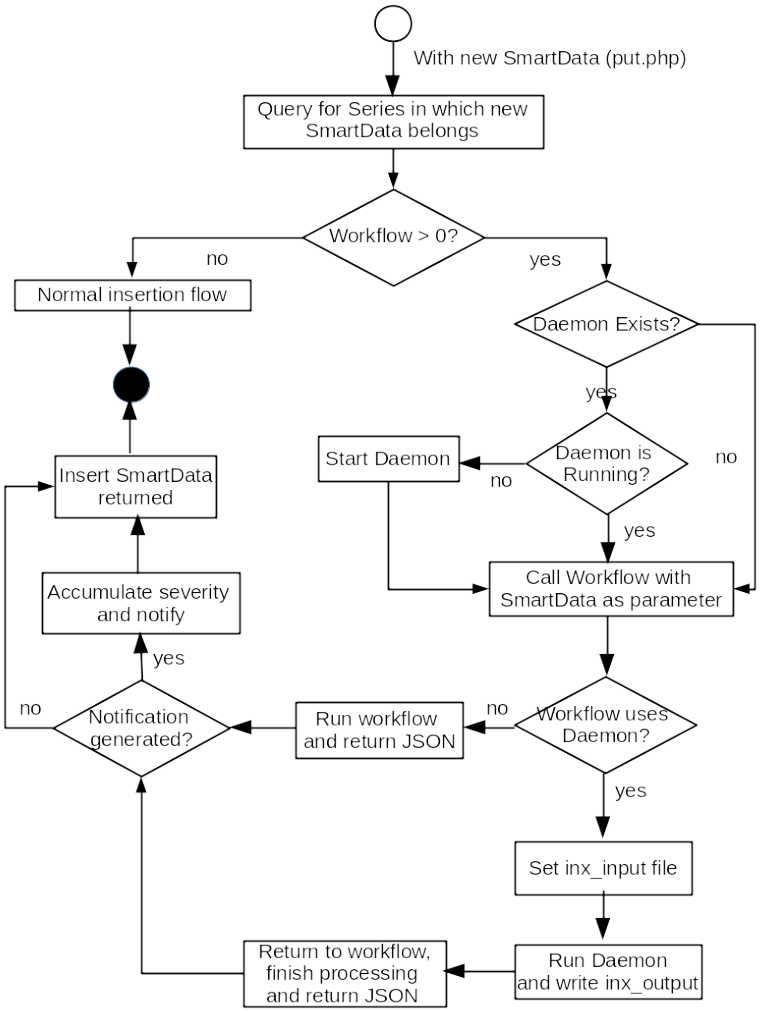
3.5. AI Workflows

SmartData on the platform can be submitted to specific workflows to process data before its proper insertion (e.g., fix known sensors error and notifying anomalies) or by applying a transformation on requested data (e.g., Fast Fourier Transform), called Input and Output Workflows, respectively. An Input Workflow can be specified during Series creation, denoting the "ID" of an existing workflow at the respective Series domain. In this way, its execution takes place during SmartData insertions on this Series (see Section Overview of the Platform for more details of this relation). Input workflows are applied to each SmartData individually, and persistency is achieved using daemons. Moreover, an Input Workflow can store useful meta-data inside the SmartData record (using the uncertainty remaining bits), or a new Series, or a file in the same folder as the workflow code. An Output

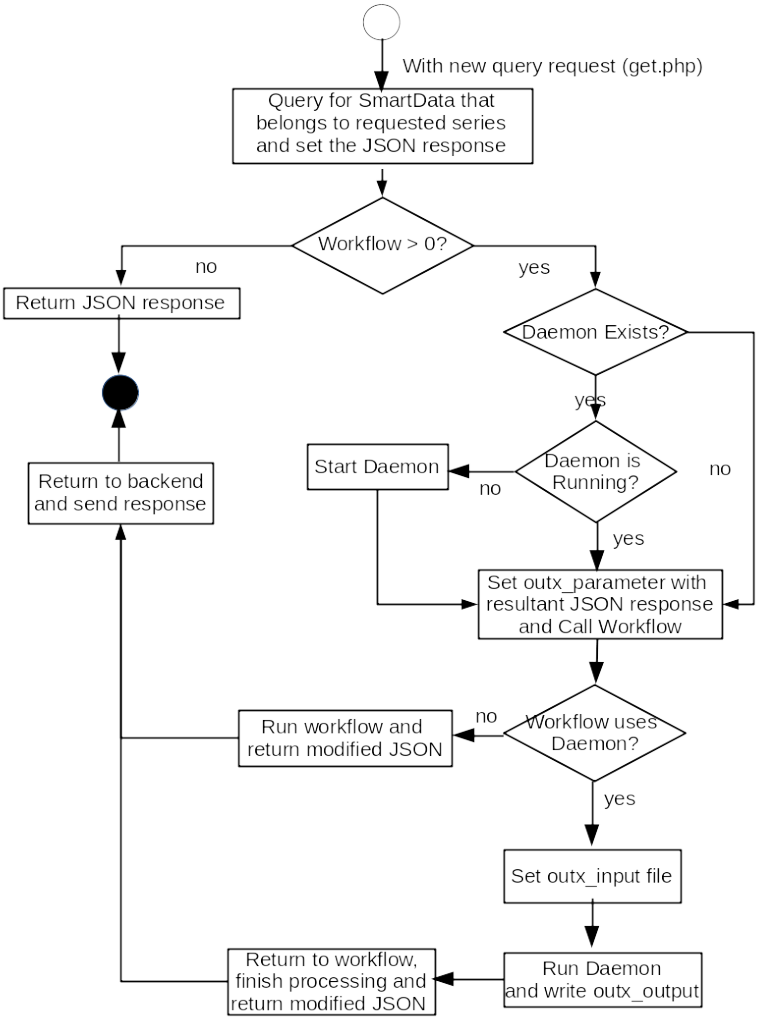
Workflow can be specified during a query request, denoting the "ID" of an existing Output Workflow at the respective domain. In this way, its execution is applied at the end of a query process to consider all SmartData records returned. For both Workflow types, if no `workflow` "ID", or 0 (default), is specified in the Series, no Workflow is executed. The same applies if the specified "ID" is not available in the current domain.

Workflows are stored on directories according to the domain they belong to (i.e., "smartdata/bin/workflow/<domain>/"). Input Workflows are named "in" followed by the workflow number ("ID"). For instance, the first Input Workflow of a domain, ID = 1, must be named "in1". Output Workflows are named "out" followed by the workflow number (e.g., "out1"). Currently, for security purposes, installing a Workflow code requires a system admin to intermediate the operation, but the code itself can be user-defined for the specific domain of interest.

Input Workflow Diagram



Output Workflow Diagram



A simple example of python workflow

```
#!/usr/bin/env python3 import sys import json if __name__ == '__main__': #+++++++++++++++++ DO NOT CHANGE THIS LINE +++++++++++++++++ smartdata = json.loads(sys.argv[1]) # Load json from argv[1] #+++++++++++++++++ DO NOT CHANGE THIS LINE +++++++++++++++++ # ... # DO SOMETHING HERE smartdata['value'] = 2*smartdata['value'] # example # ... #+++++++++++++++++ DO NOT CHANGE THIS LINE +++++++++++++++++ print(json.dumps(smartdata)) # Send smartdata back to API #+++++++++++++++++ DO NOT CHANGE THIS LINE +++++++++++++++++
```

3.5.1. Persistency

Input Workflows are executed for each instance of SmartData. Workflows that require persistence (e.g., requiring information of more than one SmartData) should implement a daemon. Daemons are meant to be separated processes that receive data from the workflow, do the processing, and either return this to the workflow or insert the processed data on a new Series, preserving the original data. Before a workflow execution, the platform checks for the existence of a demon for this workflow. If so, the platform Backend assures its execution, initializing it whenever necessary. Additionally, each workflow must manage its data, including the daemon's input and output.

Daemons are placed on the same directory as their respective workflows (i.e., "smartdata/bin/workflow/<domain>/"), and each workflow can have only one daemon. The daemons are named with the same name of the workflow plus the word "daemon" (e.g., "in1_daemon"). Files that receive daemon inputs or outputs are named with the same name of the workflow plus the word

"input" or "output" accordingly (e.g., "in1_input" and "in1_output").

The platform Backend manages the execution of a daemon. The Backend creates two support files, one for the process pid and the other for the execution log. The pid and log file are named with the workflow name plus the word "pid" or "log" accordingly (e.g., "in1_pid" and "in1_log").

Daemons also have a life cycle, finishing their execution after the current SmartData processing. To achieve this, one can implement a watchdog implementation over the input file content.

The following example of workflow writes its input into a file to be processed by the daemon, keeping data persistency

import sys, json from process_verify import process_is_alive if __name__ == '__main__': ''' This dummy workflow is used to calculate the average of the last 10 inserted SmartData ''' if len(sys.argv) != 2: exit(-1) #++++++++++++++++++ DO NOT CHANGE THIS LINE ++++++++++++++++++ smartdata = json.loads(sys.argv[1]) # Load json from argv[1] #++++++++++++++++++ DO NOT CHANGE THIS LINE ++++++++++++++++++ # ... # DO SOMETHING HERE IF IT WILL CHANGE DATA # ... #++++++++++++++++++ DO NOT CHANGE THIS LINE ++++++++++++++++++ print(json.dumps(smartdata)) # Send smartdata back to API #++++++++++++++++++ DO NOT CHANGE THIS LINE ++++++++++++++++++ # ... # DO SOMETHING HERE IF IT WILL NOT CHANGE DATA (INCREASES PARALELLISM BY UNBLOCKING PHP) with open('in1_input', 'a') as fifo: fifo.write(json.dumps(smartdata)+"\n") fifo.close() # ...

3.5.2. Loading previous data

Daemon receives its entry from its respective input file.
However, a Daemon can also request historical data executing an importer to complete its input file.

The following piece of code represents an example of a daemon calling a data importer

... if not os.path.exists("in1_input"): os.system("./get_data.py <parameters>") ...

3.5.3. Inserting new data

In case the workflow does not change the SmartData, it may insert the processed SmartData on other time Series through a data inserter. This script must create a different Series for the new data. A possibility is to use the very same series configuration but with another dev.

The following piece of code represents a daemon calling an export script to create the new series

... if os.path.exists("put_data.py"): os.system("./put_data.py <parameters>) ...

3.5.4. Notifications

Workflows on the platform can produce notifications on the processed data. These notifications carry information related to faults and abnormalities on the data. The SmartData, in JSON format, has a notification field ("notify") appended to its structure before returning to the insertion API. The following PHP code snippet depicts the process of adding the notify information to the SmartData JSON.

```
... $smartdata = json_decode($argv[1],false); // 0x84924964 == 2224179556 == temperature if ($smartdata->unit == 2224179556 && $smartdata->value < 0) { $smartdata->notify = array( 'severity' => 100, 'description' => 'Invalid value for temperature in SI unit (Kelvin)'); } echo json_encode($smartdata); //Send smartdata back to API ...
```

Whenever a notify structure is attached to the returned SmartData JSON, the platform will log the information on the API log files. A notification severity control is recommended to avoid excessive notification entries on the API logs. A severity threshold can be specified during the workflow building so that a notify structure would only be added to the processed SmartData when the notifications reach the workflow severity threshold.
The daemon process handles the current severity level and the severity threshold. The verification is done during insertions and can be logged to auxiliary files to maintain persistency through multiple executions.
Additionally, a workflow can be customized to communicate notifications to the domain owners, for instance, by sending an email to the domain mail group. This information can also be brought to the platform so that the API would perform the communication. The following is a PHP code snippet depicting an example of the notifications handling on the API.

```
private function _notify(\stdClass $json) { if(isset($json->notify)) { $notification = "Domain: {$this->_domain}. Data irregularity."; if(isset($json->notify->description)) { $notification .= " Description: {$json->notify->description}."; } if(isset($json->notify->severity)) { //threshold can be either defined by the workflow or by a standard value (100% in this case) // isset($json->notify->severity_threshold) $notification .= " Severity level: {$json->notify->severity}."; if ($json->notify->severity > 100) { //send mail or message bus } } //log notification self::debug_X($notification); } }
```

Output workflows and Search workflows can also produce notifications. However, the API will not handle those notifications since data processed by them is directly returned to the user.

3.6. Data Searching

Method: POST
URL: https://iot.lisha.ufsc.br/api/search.php

A Search AI Workflow is a server-side code capable of searching for specific data patterns using one or multiple data matching policies and AI algorithms. Similar to the IoT Platform, a Search AI Workflow is a space-time operation that, instead of using the typical geographic search engine built in the Platform to collect data, runs a query using the specified space-time region as the parameter.

This method queries for data following, but not limited to, a SmartData Series and the specified domain, defined through the IoT Platform procedure. The query and data handling processes are specific to the search code related to the domain.

The `workflow` attribute specifies the "ID" of the selected Workflow. A secondary JSON object, named `parameter`, is a customizable object to provide additional information to the Search Workflow. The semantics of the received parameters are directly related to the Search Workflow. For instance, a pattern searching algorithm can interpret the parameters as a list of SmartData that represents the desired pattern. However, parameters are not necessarily SmartData objects and are free-form key/value pairs. The parameters are made available to the Search Workflow by the Backend API when parsing the Search request. The following is an example of a Search request body with a

customizable `parameter` object:

```
"series": Object { "version" : 1.1, "unit" : unsigned long, "x" : long, "y" : long, "z" : long, "r" : unsigned long, "t0" : unsigned long long, "tf" : unsigned long long, "uncertainty" : unsigned long, "workflow" : unsigned long }, "parameter" : Object { ... } }
```

Search Workflows are stored on the same directory of regular Workflows from the same domain (i.e., "bin/workflow/<domain>/"). Search Workflows are named "search" followed by the Workflow number ("ID"). For instance, "search1" stands for Search Workflow 1. Similar to regular workflows, for security purposes, installing a Search Workflow code requires a system admin to intermediate the operation, but the code itself can be user-defined for the specific domain of interest.

The Series and the parameter objects are made available as arguments 1 and 2 from the argument vector, where argument 1 corresponds to the provided Series and argument 2 corresponds to the parameter object. Search Workflows return data to the API through console prints. Thus, the Search Workflow can only print the final JSON object during its execution. The output of a Search Workflow must be a set of SmartData, possibly with multiple devs and units.

An example of a search algorithm is presented below:

```
#!/usr/bin/php <?php require_once( __DIR__ . '/../../smartdata/SmartAPI.php'); use SmartData\SmartAPI\Internals\{JsonAPI, BinaryAPI}; use SmartData\{Series, Backend_V1_1, Credentials, Config}; function get_data($json) { $json_aux = json_decode($json); list($credentials,$series,$aggregator,$options) = JsonAPI::parse_get($json_aux); $DOMAIN = $credentials->domain; $cred = new Credentials($DOMAIN, $username, $password); $backend = new Backend_V1_1($cred, true); $response = $backend->query($series); return json_encode($response); } $series_param = json_decode($argv[1]); $options_param = json_decode($argv[2]); $series1 = array( 'series' => array( 'version' => "1.1", 'unit' => 2224179556, 'x' => $series_param->x, 'y' => $series_param->y, 'z' => $series_param->z, 'r' => $series_param->r, 't0' => $series_param->t0, 't1' => $series_param->t1, 'dev' => $series_param->dev, 'workflow' => 0 ) ); $data = json_decode(get_data(json_encode($series1))); $series = $data->series; $response_json = array('series' => array()); $index = 0; foreach ($series as &$smartdata) { $temp_celsius = $smartdata->value - 273.15; // kelvin to celsius degrees if ($temp_celsius < 0 && $temp_celsius > 45) $response_json[$index++] = $smartdata; } unset($smartdata); echo json_encode($response_json);
```

3.7. Response codes

The HTTP response codes are used to provide a response status to the client.

Possible response codes for an API request:

- `200`:
  - `get.php`: it means that a query has been successfully completed, and the response contains the result (which may be empty)
- `204`:
  - `create.php`: it means that the series has been created successfully (there is no content in the response).
- `400`: it means there is something wrong with your request (bad format or inconsistent field).

- `401`: it means that you are not authorized to manipulate the domain specified.

3.8. Plotting a dashboard with Grafana

To plot a graph, do the following:

- 1. Inside Grafana's interface, go to `Dashboards` => `Create your first dashboard` => `Graph`.
- 2. Now you should be seeing a cartesian plane with no data-points, click on `Panel Title` => `Edit`.
- 3. This should take you to the `Queries` tab. Now you can choose your Data Source and put its due information.
- 4. If you are using `SmartData UFSC` Data Source, fill the `Interest` and `Credential` fields with the information used for insertion (see ((IoT Platform|#Create_series|Section Create]).
- 5. You can tweak your plotting settings by using the `Visualization` tab. Save your Dashboard by hitting Ctrl+S.

After doing these steps, the information should be shown instantly.

4. Binary API for SmartData Version 1.1

To save energy on the IoT wireless, battery-operated network, the platform also accepts SmartData structures, encoded as binary, considering 32-bit little-endian representation. Each data point is sent as a concatenation of the Series and the SmartData structures in binary representation, totaling 78 bytes.

struct Series { unsigned char version; unsigned long unit; long x; long y; long z; unsigned long r; unsigned long long t0; unsigned long long tf; } struct SmartData { unsigned char version; unsigned long unit; double value; unsigned long uncertainty; long x; long y; long z; unsigned long dev; unsigned long long t; }

4.1. Create series (Binary)

- Create follows the same semantic presented in Create Series.

Method: POST
URL for create: https://iot.lisha.ufsc.br/api/create.php
Body: Series

Byte	36	32	28	24	20	16	8	0
	version	unit	x	y	z	r	t0	tf

4.2. Insert data (Binary)

Method: POST
URL: https://iot.lisha.ufsc.br/api/put.php
Body: SmartData

Byte	40	36	28	24	20	16	12	8	0
	version	unit	value	uncertainty	x	y	z	dev	t

4.2.1. Binary Multi SmartData

The binary version of Multi SmartData uses different URLs to access the API. Each URL handles a specific type of data repetition. Thus, the method can attend to the specificities of each binary format. There are three cases:

MultiValue SmartData
In the binary format, the **flag**'s bit 0 shall be set if the **period** is defined in the header, or unset if the **offset** is defined for each data point. The **flag**'s bit 1 shall be set if the **uncertainty** is defined in the header, or unset if it is transmitted with each data point. Therefore, the packet header can have a length of 30, 34, or 38 bytes.
Method: POST
URL: https://iot.lisha.ufsc.br/api/mv_put.php
Body: MultiDevice SmartData
Binary Format:
Packet Header (first 30 bytes):

Byte	29	25	21	17	13	5	1	0	(4)	(4)
	version	unit	x	y	z	t0	dev	flag	period	uncertainty

The payload will also vary from 16 to 8 bytes. If **period** and **uncertainty** are informed in the header, only the value of each data point will be included in the payload. Otherwise, each value is sent along with both attributes following the table below.

Packet Payload (N x 16 bytes):

Byte	12	4	0
	offset	value	uncertainty

MultiDeviceSmartData
Method: POST
URL: https://iot.lisha.ufsc.br/api/md_put.php
Body: MultiDevice SmartData
Packet Header (first 25 bytes):

Byte	24	20	16	12	8	0
	version	unit	x	y	z	t0

Packet Payload (N x 20 bytes):

Byte	16	8	4	0
	offset	value	dev	uncertainty

MultiUnitSmartData
Method: POST
URL: https://iot.lisha.ufsc.br/api/mu_put.php
Body: MultiUnit SmartData

Binary Format:
Packet Header (first 21 bytes):

Byte	20	16	12	8	0
	version	x	y	z	t0

Packet Payload (N x 24 bytes):

Byte	20	16	8	4	0
	unit	offset	value	dev	uncertainty

4.3. Version format

The version field has 8 bits and is composed of a major and a minor version. The major version is related to API compatibility. On the other hand, the minor version defines some properties of the SmartData. For instance, minor version 1 defines a stationary SmartData, while minor version 2 a mobile SmartData.

```
enum { STATIONARY_VERSION = (1 << 4) | (1 << 0), MOBILE_VERSION = (1 << 4) | (2 << 0), };
```

5. Client Authentication

The EPOS IoT API infrastructure supports authentication with client certificates. To implement it, you should request a client certificate to LISHA through the Mailing List.

If you are using the eposiotgw script to send SmartData from a TSTP network to IoT API infrastructure, you should do the following steps to authenticate with the client certificate.

- 1. Use eposiotgw available on EPOS GitLab
- 2. Copy the files .pem and .key provided by LISHA to the same directory of the eposiotgw script
- 3. Call eposiotgw using the parameter -c with the value equal to the name of the certificate file WITHOUT the extension. Both files (.pem and .key) should have the same basename.

If you are using esp8266 with axTLS library, you should convert the certificates to a suitable format, with two .der files. To do this, follow the instructions below:

```
openssl pkcs12 -export -clcerts -in client-CERT.pem -inkey client-CERT.key -out client.p12 openssl pkcs12 -in client.p12 -nokeys -out cert.pem -nodes openssl pkcs12 -in client.p12 -nocerts -out key.pem -nodes openssl x509 -outform der -in cert.pem -out cert.der openssl rsa -outform der -in
```

key.pem -out key.der

6. Scripts

6.1. C++

6.1.1. Get Script Example

This script is based on httplib.

```
#define CPPHTTPLIB_OPENSSL_SUPPORT #include "httplib.h" #include <iostream> using namespace std; int main(void) { httplib::SSLClient cli("iot.lisha.ufsc.br", 443); cli.enable_server_certificate_verification(false); const char * series = "{\"series\":{\"version\":\"1.2\",\"unit\":XXXXXXXX,\"t0\":XXXXXXXX,\"t1\":XXXXXXXX,\"dev\":XXXXXXXX,\"signature\":XXXXXXXX},\"credentials\":{\"domain\":\"XXXXXXXX\",\"username\":\"XXXXXXXX\",\"password\":\"XXXXXXXX\"}}"; auto res = cli.Post("/api/get.php", series, strlen(series) , "text/plain"); if (res) { cout << res->status << endl; cout << res->get_header_value("Content-Type") << endl; cout << res->body << endl; } else { cout << "error code: " << res.error() << std::endl; } return 0; }
```

6.2. Python

6.2.1. Get Script Example

The following python code queries luminous intensity data at LISHA from the last 5 minutes.

```
#!/usr/bin/env python3 import time, requests, json get_url ='https://iot.lisha.ufsc.br/api/get.php' epoch = int(time.time() * 1000000) query = { 'series' : { 'version' : '1.1', 'unit' : 2224179493, //equivalent to 0x84924925 = luminous intensity 'x' : 741868770, 'y' : 679816011, 'z' : 25285, 'r' : 10*100, 't0' : epoch - (5*60*1000000), 'tf' : epoch, 'dev' : 0 }, 'credentials' : { 'domain' : 'smartlisha', 'username' : 'smartusername', 'password' : 'smartpassword' } } session = requests.Session() session.headers = {'Content-type' : 'application/json'} response = session.post(get_url, json.dumps(query)) print("Get [", str(response.status_code), "] (", len(query), ") ", query, sep='') if response.status_code == 200: print(json.dumps(response.json(), indent=4, sort_keys=False))
```

6.2.2. Put Script Example

The following python code inserts a JSON with a certificate.

```
#!/usr/bin/env python3 # To get an unencrypted PEM (without passphrase): # openssl rsa -in certificate.pem -out certificate_unencrypted.pem import os, argparse, requests, json,ssl from requests.adapters import HTTPAdapter from requests.packages.urllib3.poolmanager import PoolManager parser = argparse.ArgumentParser(description='EPOS Serial->IoT Gateway') required = parser.add_argument_group('required named arguments') required.add_argument('-c','--certificate', help='Your PEM certificate', required=True) parser.add_argument('-u','--url', help='Post URL', default='https://iot.lisha.ufsc.br/api/put.php') parser.add_argument('-j','--json', help='Use JSON API', required=True) args = vars(parser.parse_args()) URL = args['url'] MY_CERTIFICATE = [args['certificate']+'.pem', args['certificate']+'.key'] JSON = args['json'] session = requests.Session() session.headers = {'Content-type' : 'application/json'} session.cert =
```

```
MY_CERTIFICATE try: response = session.post(URL, json.dumps(JSON)) print("SEND", str(response.status_code), str(response.text)) except Exception as e: print("Exception caught:", e)
```

6.3. R

6.3.1. Get Script Example

The following python code queries Temperature data at LISHA from an arbitrarily defined time interval.

```
library(httr) library(rjson) library(xml2) get_url <- "https://iot.lisha.ufsc.br/api/get.php" json_body <- '{ "series":{ "version":"1.1", "unit":0x84924964, "x":741868840, "y":679816441, "z":25300, "r":0, "t0":1567021716000000, "tf":1567028916000000, "dev":0, "workflow": 0 }, "credentials":{ "domain":"smartlisha" } }' res <- httr::POST(get_url, body=json_body, verbose()) res_content = content(res, as = "text") print(jsonlite::toJSON(res_content))
```

The following code gets Temperature data at LISHA from the last 5 minutes.

```
library(httr) library(rjson) library(xml2) get_url <- "https://iot.lisha.ufsc.br/api/get.php" time <- Sys.time() time_0 <-as.numeric(as.integer(as.POSIXct(time))*1000000) json_body <- '{ "series":{ "version":"1.1", "unit":0x84924964, "x":741868840, "y":679816441, "z":25300, "r":0, "t0":' json_body <- capture.output(cat(json_body, time_0 - 5*60*1000000)) json_body <- capture.output(cat(json_body, ',"tf":')) json_body <- capture.output(cat(json_body, time_0)) end_string <- ', "dev":0, "workflow": 0 }, "credentials":{ "domain":"smartlisha" } }' json_body <- capture.output(cat(json_body, end_string)) res <- httr::POST(get_url, body=json_body, verbose()) res_content = content(res, as = "text") print(jsonlite::toJSON(res_content))
```

7. Troubleshooting

7.1. TLS support for Post-Handshake Authentication

TLS 1.3 has the Post-Handshake Authentication disabled by default. However, the IoT platform requires PHA to securely connect with clients. This issue can be easily worked around with a custom SSLContext forcing the use of TLS 1.2, which has PHA enabled by default. An example in Python follows:

```
import ssl ctx = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2) connection = HTTPSConnection("iot.lisha.ufsc.br", 443, context=ctx);
```

Review Log

Ver	Date	Authors	Main Changes
1.0	Feb 15, 2018	Caciano Machado	Initial version
1.1	Apr 4, 2018	César Huegel	Rest API documentation
1.2	Apr 4, 2020	Leonardo Horstmann	Review for EPOS 2.2. and ADEG
1.3	Jun 27, 2020	José Luis Hoffmann, Leonardo Horstmann, Roberto Scheffel	Review for Insert Changes and ADEG

1.4	Sep 30, 2020	Guto	Major revision
1.5	Mar 16, 2022	Roberto Scheffel	Series Documentation update
1.6	May 24, 2022	Roberto Scheffel	Downsample "aggregator" documentation added
1.7	December 16, 2022	Mateus Lucena	Updated Prolog
1.8	March 19, 2026	Leonardo Passig Horstmann	Removing reference to accuracy or error in a series