IoT Platform
Software/Hardware Integration Lab at UFSC
Table of Contents

IoT Platform .................................................................................................................................. 1
  Table of contents .......................................................................................................................... 1
1. Prolog ........................................................................................................................................... 2
2. IoT Platform Overview ............................................................................................................... 2
  2.1. SmartData ............................................................................................................................... 3
  2.2. SmartData Series .................................................................................................................... 4
  2.3. Authentication and Authorization .......................................................................................... 5
  2.4. Usefull SmartData Units ......................................................................................................... 5
3. REST API for Stationary Objects ................................................................................................. 6
  3.1. Data Querying ......................................................................................................................... 6
    3.1.1. Data Aggregation .............................................................................................................. 6
    3.1.2. Fault Injection .................................................................................................................. 7
  3.2. Series Creation ........................................................................................................................ 7
    3.2.1. Series Types and Modes ................................................................................................. 8
    3.2.2. Series Status .................................................................................................................... 9
    3.2.3. Meaninful Types and Status ........................................................................................... 9
  3.3. Data Insertion .......................................................................................................................... 15
    3.3.1. Bulk Data Insertion ......................................................................................................... 15
  3.4. Series Termination .................................................................................................................... 17
  3.5. AI Workflows .......................................................................................................................... 17
    3.5.1. Persistency ...................................................................................................................... 21
    3.5.2. Loading previous data ..................................................................................................... 22
    3.5.3. Inserting new data .......................................................................................................... 22
    3.5.4. Notifications .................................................................................................................... 22
  3.6. Data Searching ......................................................................................................................... 23
  3.7. Response codes ........................................................................................................................ 25
  3.8. Plotting a dashboard with Grafana ......................................................................................... 25
4. Binary API for SmartData Version 1.1 ....................................................................................... 27
  4.1. Create series (Binary) ............................................................................................................. 27
  4.2. Insert data (Binary) ................................................................................................................ 27
    4.2.1. Binary Multi SmartData ................................................................................................. 28
  4.3. Version format .......................................................................................................................... 29
5. Client Authentication .................................................................................................................... 31
6. Scripts .......................................................................................................................................... 32
  6.1. Python ..................................................................................................................................... 32
    6.1.1. Get Script Example ......................................................................................................... 32
    6.1.2. Put Script Example ......................................................................................................... 32
  6.2. R .......................................................................................................................................... 33
    6.2.1. Get Script Example ......................................................................................................... 33
7. Troubleshooting ........................................................................................................................... 35
  7.1. TLS support for Post-Handshake Authentication ................................................................. 35

Review Log .................................................................................................................................... 35

https://epos.lisha.ufsc.br/  30/01/2021
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
  - 3.2. 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
  - 3.4. 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. Python
    - 6.1.1. Get Script Example
    - 6.1.2. Put Script Example
  - 6.2. R
    - 6.2.1. Get Script Example
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 implemented by the set of actors depicted in the figure below, which provide Secure Storing and Data Science Processing for Series of SmartData. 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.

2. 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:

<table>
<thead>
<tr>
<th>SmartData</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
</tr>
<tr>
<td>unit</td>
</tr>
<tr>
<td>value</td>
</tr>
<tr>
<td>uncertainty</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>z</td>
</tr>
<tr>
<td>t</td>
</tr>
<tr>
<td>dev</td>
</tr>
<tr>
<td>signature</td>
</tr>
</tbody>
</table>

- **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 at the same 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:

```json
{
  "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:

<table>
<thead>
<tr>
<th>SmartData Series</th>
<th>version</th>
<th>unit</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>r</th>
<th>t₀</th>
<th>tf</th>
<th>period</th>
<th>count</th>
<th>event</th>
<th>workflow</th>
</tr>
</thead>
</table>

- **version**: the version of the SmartData in the series (a series does not contain mixed versions of 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;
- **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, the series associated with time-triggered SmartData must define a **period**, whereas those not defining such attribute are assumed to be event-driven. Additionally, the beginning of a series can be specified in one of three ways: by time (giving t₀), by event, and manually (by not giving t₀, which is then assumed to be the current time).

Therefore, the beginning of a time-triggered series can be designated by an event and this does not make it an event-driven series. Similarly, an event-driven series can be started at a given time. The end of a series can be specified by time (giving tf), by event, manually (with the method `finish`), which makes tf equal to the current time, and, additionally, in terms of event counting (by giving...
count). Events are expressed in terms of 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
   "period" : unsigned long
   "count" : unsigned long
   "event" : string
   "accuracy" : 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 RESTful 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](https://epos.lisha.ufsc.br/) 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
    "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 there are multiple SmartData 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,
    "length" : unsigned long,
    "spacing" : unsigned long
}
```

The time-related attributes range, delay, and spacing are expressed in us. All attributes but name are optional. More sophisticated aggregation functions can be modeled as output workflows. An aggregator can also be combined with an output workflow, case in which the aggregator is run before the workflow (i.e., the workflow will handle already aggregated data).
The following aggregators are available and can be applied by setting 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 `parameter` is omitted, then the lower limit is ignored and only `offset` is considered; if `offset` is suppressed, then the higher limit is ignored and only `parameter` is considered;
- **higherThan**: filters out SmartData whose value is not greater 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:  
  \[ \text{drift} = \text{parameter} \times i. \]
- **stuckAt**: the values of the SmartData selected by the query in the time windows defined by \([\text{offset}, \text{length}]\) spaced by `spacing` are 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 \([\text{offset}, \text{length}]\) spaced by `spacing`.
- **constantGain**: multiplies the value of each SmartData selected by the query by `parameter` considering the windowing mechanism described earlier.

### 3.2. Series Creation

Method: POST

URL for create: `https://iot.lisha.ufsc.br/api/create.php`

Body:

```json
"Series" : Object {
    "version" : unsigned char
    "unit" : unsigned long
    "x" : long
}```
This method creates a SmartData Series if there is no other series that already encompasses the designated unit, space-time region \((x, y, z, r, dev, t0, tf)\) for the same operating mode (e.g. time-triggered vs event-driven). If the new series intersects existing ones but is not fully contained in any of them, then a new series is indeed created with a space-time region that is the smallest one to encompass both, the given space-time region and all the preexisting series intersecting with that region. That is, the method can be used to irreversibly merge series and therefore 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 must 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 quite different types of series with quite different operating modes. As previously stated, series created 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 attributes \(t0\), \(tf\), \(count\), \(event\), and \(uncertainty\).

For sanity checking and documentation purposes, a series can have a starting time different from the time it was created. The starting time can be explicitly specified using attribute \(t0\). It can also be implicitly set from the time an event occurs, which can be 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 makes 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 method finish, 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 along its life cycle:

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waiting</td>
<td>the series is created, but <code>t0</code> is not yet set or it is set but not yet reached</td>
</tr>
<tr>
<td>Open</td>
<td>data for the series is being collected and <code>tf</code> is not yet set or it is set but not yet reached</td>
</tr>
<tr>
<td>Closed</td>
<td><code>tf</code> is defined and reached, so no further insertions are allowed</td>
</tr>
<tr>
<td>Defective</td>
<td>the series should be closed, but data counting does not match the specification</td>
</tr>
</tbody>
</table>

### 3.2.3. Meaninful Types and Status

**Time-Triggered Series**

**TT-t₀,tₖ: begin and end set at creation**

<table>
<thead>
<tr>
<th>JSON</th>
<th>Attributes</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;Series&quot; : Object {</td>
<td><code>p</code> = period</td>
<td>Waiting : <code>now</code> 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Open</strong>: <code>t₀</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Closed</strong>: <code>(now &gt; t₀) ∧ (n &gt;= c)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Defective</strong>: <code>(now &gt; tₖ) ∧ (n &gt;= c)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TT-t₀,c: begin set at creation and end set by count**

<table>
<thead>
<tr>
<th>JSON</th>
<th>Attributes</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TT-t₀.c: begin set at creation and end set by count

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

- \( p = \text{period} \)
- \( t₀ = \text{t₀} \)
- \( c = \text{count} \)
- \( tᵢ = t₀ + p \times c \)
- \( n = \text{current data count} \)
- \( \text{now} = \text{current time} \)

Waiting: \( \text{now} = 0 \)
Open: \( t₀ \)
Closed: \( (\text{now} > tᵢ) \land (n \geq c) \)
Defective: \( (\text{now} > tᵢ) \land (n \geq c) \)

TT-t₀.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,
    "t₀" : unsigned long,
    "period" : unsigned long,
    "event" : string,
    "uncertainty" : unsigned long,
    "workflow" : unsigned long
}
```

- \( p = \text{period} \)
- \( t₀ = \text{t₀} \)
- \( tᵢ = \text{finish.t} \)
- \( c = tᵢ - (tᵢ - t₀) / p \)
- \( n = \text{current data count} \)
- \( \text{now} = \text{current time} \)

Waiting: \( \text{now} = 0 \)
Open: \( t₀ \)
Closed: \( tᵢ \land (n \geq c) \)
Defective: \( tᵢ \land (n \geq c) \)

TT-e.t: 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,
    "t₀" : unsigned long long,
    "t₀" : unsigned long,
    "period" : unsigned long,
    "event" : string,
    "uncertainty" : unsigned long,
    "workflow" : unsigned long
}
```

- \( p = \text{period} \)
- \( t₀ = \text{t₀} \)
- \( tᵢ = \text{finish.t} \)
- \( c = tᵢ - (tᵢ - t₀) / p \)
- \( n = \text{current data count} \)
- \( \text{now} = \text{current time} \)

Waiting: \( \text{now} = 0 \)
Open: \( t₀ \)
Closed: \( tᵢ \land (n \geq c) \)
Defective: \( tᵢ \land (n \geq c) \)

- finish.event →
- series.even = finish.event

https://epos.lisha.ufsc.br/  30/01/2021
### TT-e.t: 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
    "event" : string
    "uncertainty" : unsigned long
    "workflow" : unsigned long
}
```

- \( p = \text{period} \)
- \( t_i = tf \)
- \( t_0 = \text{series}[0].t \)
- \( c = t_0 \rightarrow (t_i - t_0) / p \)
- \( n = \text{current data count} \)
- \( \text{now} = \text{current time} \)

#### Statuses:
- **Waiting**: \( \neg t_0 \)
- **Open**: \( t_0 \land (t_i \land t_0) \land (n \geq c) \)
- **Closed**: \( t_0 \land (\text{now} > t_i) \land (n \geq c) \)
- **Defective**: \( t_0 \land (\text{now} > t_i) \land (n \geq c) \)

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

#### JSON

```json
"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
}
```

- \( p = \text{period} \)
- \( c = \text{count} \)
- \( t_0 = \text{series}[0].t \)
- \( t_i = t_0 \rightarrow t_0 + p \times c \)
- \( n = \text{current data count} \)
- \( \text{now} = \text{current time} \)

#### Statuses:
- **Waiting**: \( \neg t_0 \)
- **Open**: \( t_0 \land (t_i \land t_0) \land (n \geq c) \)
- **Closed**: \( t_0 \land (\text{now} > t_i) \land (n \geq c) \)
- **Defective**: \( t_0 \land (\text{now} > t_i) \land (n \geq c) \)

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

#### JSON

```json
"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
}
```

- \( p = \text{period} \)
- \( c = \text{count} \)
- \( t_0 = \text{series}[0].t \)
- \( t_i = t_0 \rightarrow t_0 + p \times c \)
- \( n = \text{current data count} \)
- \( \text{now} = \text{current time} \)

#### Statuses:
- **Waiting**: \( \neg t_0 \)
- **Open**: \( t_0 \land (t_i \land t_0) \land (n \geq c) \)
- **Closed**: \( t_0 \land (\text{now} > t_i) \land (n \geq c) \)
- **Defective**: \( t_0 \land (\text{now} > t_i) \land (n \geq c) \)
TT-e.f: begin set by data and end by finish

```json
"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
}
```

\[ p = \text{period} \]
\[ t_0 = \text{series[0].t} \]
\[ t_f = \text{finish.t} \]
\[ c = t_0 \land t_f \rightarrow (t_f - t_0) / p \]
\[ n = \text{current data count} \]
\[ \text{now} = \text{current time} \]

Waiting: \( \neg t_0 \)
Open: \( t_0 \land \neg t_f \)
Closed: \( t_0 \land t_f \land (n \geq c) \)
Defective: \( t_0 \land t_f \land (n = 0) \)

Event-Driven Series

**ED-t_0.t:** begin and end set at creation

**JSON**

```json
"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
}
```

\[ t_0 = t_0 \]
\[ t_f = t_f \]
\[ n = \text{current data count} \]
\[ \text{now} = \text{current time} \]

Waiting: \( \text{now} 0 \)
Open: \( t_0 \land \neg t_f \)
Closed: \( \text{now} > t_f \land (n > 0) \)
Defective: \( \text{now} > t_f \land (n = 0) \)

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

**JSON**

```json
"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
}
```

\[ t_0 = t_0 \]
\[ t_f = t_f \]
\[ n = \text{current data count} \]
\[ \text{now} = \text{current time} \]

Waiting: \( \text{now} 0 \)
Open: \( t_0 \land \neg t_f \)
Closed: \( \text{now} > t_f \land (n > 0) \)
Defective: \( \text{now} > t_f \land (n = 0) \)
### ED-t₀.c: begin set at creation and end set by count

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₀</td>
<td>t₀</td>
</tr>
<tr>
<td>c</td>
<td>count</td>
</tr>
<tr>
<td>tᵣ</td>
<td>series[c].t</td>
</tr>
<tr>
<td>n</td>
<td>current data count</td>
</tr>
<tr>
<td>now</td>
<td>current time</td>
</tr>
</tbody>
</table>

#### Status
- **Waiting**: now = 0
- **Open**: ¬tᵣ ∧ (t₀ ≡ Closed: tᵣ ∧ (n >= c))
- **Defective**: tᵣ ∧ (n = 0)

### ED-t₀.f: begin set at creation and end set by finish

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₀</td>
<td>t₀</td>
</tr>
<tr>
<td>tᵣ</td>
<td>finish.t</td>
</tr>
<tr>
<td>n</td>
<td>current data count</td>
</tr>
<tr>
<td>now</td>
<td>current time</td>
</tr>
</tbody>
</table>

#### Status
- **Waiting**: now = 0
- **Open**: ¬tᵣ ∧ (t₀ ≡ Closed: tᵣ ∧ (now > tᵣ) ∧ (n >= 0))
- **Defective**: tᵣ ∧ (now > tᵣ) ∧ (n = 0)

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

<table>
<thead>
<tr>
<th>JSON</th>
<th>Attributes</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>t₀ = t₀</td>
<td><strong>Waiting</strong>: now = 0</td>
</tr>
<tr>
<td></td>
<td>tᵣ = finish.t</td>
<td><strong>Open</strong>: ¬tᵣ ∧ (t₀ ≡ Closed: tᵣ ∧ (now &gt; tᵣ) ∧ (n &gt;= 0))</td>
</tr>
<tr>
<td></td>
<td>n = current data count</td>
<td><strong>Defective</strong>: tᵣ ∧ (now &gt; tᵣ) ∧ (n = 0)</td>
</tr>
<tr>
<td></td>
<td>now = current time</td>
<td></td>
</tr>
</tbody>
</table>

---

https://epos.lisha.ufsc.br/ 30/01/2021 13
ED-e.t: begin set by data and end set at creation

```
"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
}
```

- \( t_i = tf \)
- \( t_0 = series[0].t \)
- \( n = \text{current data count} \)
- \( \text{now} = \text{current time} \)

**Waiting**: \( \neg t_0 \)

**Open**: \( t_0 \land (t_0 < tf) \)

**Closed**: \( t_0 \land (\text{now} > t_i) \land (n > 0) \)

**Defective**: \( (\text{now} > t_i) \land (n = 0) \)

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
}
```

- \( c = \text{count} \)
- \( t_0 = series[0].t \)
- \( t_f = series[c].t \)
- \( n = \text{current data count} \)
- \( \text{now} = \text{current time} \)

**Waiting**: \( \neg t_0 \)

**Open**: \( t_0 \land \neg t_f \)

**Closed**: \( t_0 \land t_f \)

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

```
"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
}
```

**Waiting**: \( \neg t_0 \)

**Open**: \( t_0 \land \neg t_f \)

**Closed**: \( t_0 \land t_f \)
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" : 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 operation 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 SmartData originated at the same location, 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 omitted in the data.
points. Also, 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
  "t0": unsigned long long
  "dev": unsigned long
  "uncertainty": unsigned long // OPTIONAL: if given, then omit it in data points
  "period": unsigned long // OPTIONAL: if given, then omit 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
  "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*

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

### 3.4. Series Termination

**Method:** POST

**URL:** https://iot.lisha.ufsc.br/api/finish.php

**Body:**

```json
"Series": Object {
   "version": 1.1
   "unit": unsigned long
   "x": long
   "y": long
   "z": long
   "r": unsigned 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, also the 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 at the series creation, denoting the "ID" of an existing workflow at the respective series domain. Thus, it's execution takes place during SmartData insertions on this series (The SmartData relation to the Series is presented previously on the Overview of the Platform). Input workflow is applied to each SmartData individually. Persistency is provided with the support of daemons that are executed whenever available. Moreover, an input workflow can store useful metadata inside the SmartData record (using the uncertainty remaining bits), on a new series, or on 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 of the requested series. Thus, it's execution is applied at the end of a query process in order to consider all SmartData records returned. For both Input and Output Workflows, if no workflow has been defined or is defined as 0 (default), data remains in its original form. Moreover, the same applies if the specified workflow is not available in the current domain.

Workflows are stored on directories according to the domain they belong (i.e., "smartdata/bin/workflow/<domain>/"). Input Workflows should be named as "in" followed by the workflow number (identifier), for instance, the input workflow 1 must be named as "in1". Output Workflows should be named as "out" followed by the workflow number (e.g. "out1"). Currently, the code to be executed can be user-defined but needs to be installed by system administrators on the specific domain of interest.

**Input Workflow Diagram**
Output Workflow Diagram
A simple example of python workflow

```python
#!/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

The Input Workflows are executed for each instance of SmartData, i.e., the SmartData are processed by the workflow individually. For the workflows that need some persistence and must execute during all the SmartData processing (Input or Output), a daemon should be created. 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 time series, preserving the original data. Whenever a workflow execution is required, the platform checks for the existence of the demon for this workflow. If the daemon does exist, the platform Backend assures its execution, initializing it whenever necessary. Each workflow must manage its own data, including daemon's input and output.

The daemons must be placed on the same directory as their respective workflows (i.e., "smartdata/bin/workflow/<domain>/"). Each workflow can have only one daemon. The daemon of the workflow must be named with the name of the workflow, plus the word "daemon", e.g., "in1_daemon". Common names for the files that receive daemon inputs and outputs are composed by the name of the workflow, plus the words "input" or "output", e.g., "in1_input".

The daemon execution is managed by the platform with the help of two files, one holding the process pid, and the other holding the execution log. The pid file is named with the workflow name, plus the word "pid", e.g., "in1_pid". The execution log file is named with the workflow name, plus the word "log", e.g., "in1_log".

Daemons are also meant to have a life cycle and must be finalized after the end of the SmartData processing. This can be managed with 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

```python
import sys, json
from process_verify import process_is_alive

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 IF IT WILL CHANGE DATA
    # ...
```
3.5.2. Loading previous data

The daemon usually receives its entry from its respective input file. On the first SmartData to be processed, however, this input file would be empty. In this case, an importer would be executed to get historic data from the very same series to fulfill the input file.

This daemon piece of code imports data if the input file is not available yet

```python
...  
  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 time series, for the new data. A possibility is to use the very same series configuration but with another dev.

This daemon piece of code exports the calculated SmartData if the put script is available

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

3.5.4. Notifications

To notify the detection of an anomaly on the processed data, the AI Input Workflow can add information to the SmartData JSON returned to the API. This information must be declared under a "notify" vector inside JSON. The PHP code snippet below depicts the process of adding the notify information to the JSON.

```python
...  
  $smartdata = json_decode($argv[1],false); 
  // 0x84924964 == 2224179556 == temperature 
  if ($smartdata->unit == 2224179556 && $smartdata->value < 0) { 
    $smartdata->notify = array(
```
Notifications received by the platform are verified. If the severity reaches a certain threshold, an email is sent to the domain mail group. The severity threshold can be either defined by the platform or informed on the notify array as the attribute "severity_threshold". Whenever receiving a notification, the platform will log the information on the platform log files.

3.6. Data Searching

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

The search AI Workflow algorithm is a server side code capable of searching for specific data patterns using a specific data matching policy or a combination of policies and AI algorithms. Similar to the IoT Platform, it is a space-time operation, however, instead of using the typical geographic search engine built in the Platform to collect data, the search API will execute 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, which is defined through the IoT Platform procedure. The query and data handling process are specific to the search code related to the domain.

The method that searches for data is selected by the request through the workflow field of the SmartData Series. A secondary JSON object named parameter can be used to give the search algorithm parameters that are not taken from the specified space-time region. The semantics of the received parameters are completely related to the filter algorithm. For instance, a pattern searching algorithm can interpret the parameters as a list of SmartData that represents the desired pattern. The parameters can be passed to the search algorithm by appending the following structure to a series object in the body of a search query:

```json
"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 algorithms are stored on directories according to the domain they belong to (i.e., "bin/workflow/<domain>/"). Search algorithms should be named as "search" followed by the algorithm number (identifier), for instance, the search algorithm 1 must be named as "search1". Currently, the code to be executed can be user-defined but needs to be installed by system administrators on the specific domain of interest.

The parameters provided to the search algorithm can be accessed as arguments 1 and 2 from the argument vector. Argument 1 corresponds to the provided series. Argument 2 corresponds to the parameter object. The output of a search algorithm is expected to be any not null JSON object.

An example of a search algorithm is presented below:

```php
#!/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 your domain.

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 that you are seeing a cartesian plane with no data-points, click on Painel 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 Credentential 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 to be inserted into the database is sent as the 78-byte concatenation of the Series with the SmartData structures:

```c
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 on Create Series.

Method: POST

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

Body: Series

<table>
<thead>
<tr>
<th>Byte</th>
<th>36</th>
<th>32</th>
<th>28</th>
<th>24</th>
<th>20</th>
<th>16</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>version</td>
<td>unit</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>r</td>
<td>t0</td>
<td>tf</td>
</tr>
</tbody>
</table>

4.2. Insert data (Binary)

Method: POST

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

Body: SmartData
4.2.1. Binary Multi SmartData

The binary version uses different URLs to access the API, one for each type of data "repetition", due to the specificity of the 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** if defined in the header, or unset if it is transmitted with each datapoint. 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):

<table>
<thead>
<tr>
<th>Byte</th>
<th>version</th>
<th>unit</th>
<th>value</th>
<th>uncertainty</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>dev</th>
<th>t</th>
</tr>
</thead>
</table>

The payload will also vary from 16 to 8 bytes. If **period** and **uncertainty** were informed in the header, only the value will be added of each data point has to be added to the payload. Or it can be combined with one or both of the values, respecting the order and size specified in the next table.

Packet Payload (N x 16 bytes):

<table>
<thead>
<tr>
<th>Byte</th>
<th>offset</th>
<th>value</th>
<th>uncertainty</th>
</tr>
</thead>
</table>

**MultiDeviceSmartData**

Method: POST

URL: https://iot.lisha.ufsc.br/api/md_put.php

Body: MultiDevice SmartData

Packet Header (first 25 bytes):

<table>
<thead>
<tr>
<th>Byte</th>
<th>version</th>
<th>unit</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>t0</th>
<th>dev</th>
<th>flag</th>
<th>period</th>
<th>uncertainty</th>
</tr>
</thead>
</table>
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 the 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 the minor version 2 defines a mobile SmartData.

```c
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. In order to implement it, you should request a client certificate to LISHA in 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 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:

```bash
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. Python

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

```python
#!/usr/bin/env python3
import time, requests, json

get_url = 'https://iot.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))
```

6.1.2. Put Script Example
The following python code inserts a json with a certificate.

```python
#!/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)

6.2. R

6.2.1. 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,
}
The following code gets Temperature data at LISHA from the last 5 minutes.

```r
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":
    },
    "credentials":{
        "domain":"smartlisha"
    }
}
''

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))
```

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

```python
import ssl

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

Review Log

<table>
<thead>
<tr>
<th>Ver</th>
<th>Date</th>
<th>Authors</th>
<th>Main Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Feb 15, 2018</td>
<td>Caciano Machado</td>
<td>Initial version</td>
</tr>
<tr>
<td>1.1</td>
<td>Apr 4, 2018</td>
<td>César Huegel</td>
<td>Rest API documentation</td>
</tr>
<tr>
<td>1.2</td>
<td>Apr 4, 2020</td>
<td>Leonardo Horstmann</td>
<td>Review for EPOS 2.2. and ADEG</td>
</tr>
<tr>
<td>1.3</td>
<td>Jun 27, 2020</td>
<td>José Luis Hoffmann, Leonardo</td>
<td>Review for Insert Changes and ADEG</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Horstmann, Roberto Scheffel</td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td>Sep 30, 2020</td>
<td>Guto</td>
<td>Major revision</td>
</tr>
</tbody>
</table>