# Table of Contents

## IoT Platform

- Table of contents .............................................................. 1

1. **Prolog** ............................................................................. 1  
   - *Typical units representation* ........................................... 4

2. **REST API for Stationary Version** .................................... 5  
   - 2.1. **Query data** .......................................................... 5  
     - 2.1.1. Data Aggregation ................................................. 5  
     - 2.1.2. Fault Injection .................................................. 6  
   - 2.2. **Create series** ...................................................... 7  
     - Series Semantics ...................................................... 8  
   - 2.3. **Insert data** .......................................................... 8  
   - 2.4. **AI Workflows** .................................................... 9  
     - 2.4.1. Persistency ....................................................... 12  
     - 2.4.2. Loading previous data ...................................... 13  
     - 2.4.3. Inserting new data .......................................... 13  
     - 2.4.4. Notifications .................................................. 13  
   - 2.5. **Response codes** .................................................. 14  
   - 2.6. **Plotting a dashboard with Grafana** ......................... 14  

3. **Binary API for SmartData Version 1.1** ............................ 15  
   - 3.1. **Create series (Binary)** ........................................ 15  
   - 3.2. **Insert data (Binary)** .......................................... 15  
   - 3.3. **Version format** .................................................. 16  

4. **Multi SmartData** .......................................................... 17  
   - 4.1. **MultiValueSmartData** ......................................... 17  
     - 4.1.1. Periodic Data and Uniform (or Unknown) Uncertainty ........................................ 17  
   - 4.2. **MultiDeviceSmartData** ........................................ 18  
   - 4.3. **MultiUnitSmartData** ........................................... 19  

5. **Client Authentication** .................................................... 21  

6. **Scripts** ........................................................................... 22  
   - 6.1. **Python** ............................................................... 22  
     - 6.1.1. Get Script Example .......................................... 22  
     - 6.1.2. Put Script Example ........................................ 22  
   - 6.2. **R** ........................................................................ 23  
     - 6.2.1. Get Script Example .......................................... 23  

7. **Troubleshooting** .......................................................... 25  
   - 7.1. **TLS support for Post-Handshake Authentication** .......... 25  

---

https://epos.lisha.ufsc.br/  22/09/2020
IoT Platform

Table of contents

- IoT Platform
  - 1. Prolog
    - Typical units representation
  - 2. REST API for Stationary Version
    - 2.1. Query data
      - 2.1.1. Data Aggregation
      - 2.1.2. Fault Injection
    - 2.2. Create series
      - Series Semantics
    - 2.3. Insert data
    - 2.4. AI Workflows
      - 2.4.1. Persistency
      - 2.4.2. Loading previous data
      - 2.4.3. Inserting new data
      - 2.4.4. Notifications
    - 2.5. Response codes
    - 2.6. Plotting a dashboard with Grafana
  - 3. Binary API for SmartData Version 1.1
    - 3.1. Create series (Binary)
    - 3.2. Insert data (Binary)
    - 3.3. Version format
  - 4. Multi SmartData
    - 4.1. MultiValueSmartData
      - 4.1.1. Periodic Data and Uniform (or Unknown) Uncertainty
    - 4.2. MultiDeviceSmartData
    - 4.3. MultiUnitSmartData
  - 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
  - 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 Ciber-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.

Each SmartData stored in the Platform is a data point in a 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).

A SmartData can be viewed as a data structure containing the following properties:
While a Series has the following data structure:

<table>
<thead>
<tr>
<th>SmartData</th>
<th>version</th>
<th>unit</th>
<th>value</th>
<th>uncertainty</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>t</th>
<th>dev</th>
<th>signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series</td>
<td>version</td>
<td>unit</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>r</td>
<td>t0</td>
<td>t1</td>
<td>signature</td>
<td>workflow</td>
</tr>
</tbody>
</table>

Common properties to **smartdata** and **series**:

- **version**: The SmartData version. Possible values:
  - "1.1": Version 1, Stationary (.1). Data from a sensor that is not moving.
  - "1.2": Version 1, Mobile (.2). Data from a sensor that is moving.
- **unit**: The SI (or digital) unit of the data. See typical units or the full documentation.
- **signature**: (only for mobile version) An identifier for the mobile version of SmartData (version 1.2)

Specific properties:

- **smartdata**: Array of one or more elements, each containing information about a particular measurement (data point).
  - **value**: The data value itself (e.g. the temperature measured by a thermometer).
  - **uncertainty**: This is usually transducer-dependent and can express Accuracy, Precision, Resolution, or a combination thereof.
  - **x, y, z**: The absolute coordinates where the measurement was taken.
  - **t**: The time when the measurement was taken, in UNIX epoch microseconds.
  - **dev**: (optional) An identifier for multiple transducers of the same unit in the same coordinate (e.g., 3-axis accelerometer), if there is only one transducer at the same unit and coordinate, it should always be 0.

- **series**: Information about data to be fetched.
  - **x, y, z**: The absolute coordinates of the center of the sphere of interest.
  - **r**: The radius of the sphere in which all data points of interest are contained.
  - **t0**: The start time of data to be fetched, in UNIX epoch microseconds.
  - **t1**: The end time of data to be fetched, in UNIX epoch microseconds.
  - **workflow**: (optional) During insert operations, specify a server-side code to be executed every insertion, generating notifications, fixing data points following a measurement error, or creating a new Digital series. For query operations (get.php), workflow specify a server-side code to be executed at the requested data, for instance, applying a data transformation. More information regarding the workflow definitions and capabilities can
be found in the AI Workflow Section.

Yet, some other properties can be useful when using the IoT Platform, the Credentials:

- credentials: Optional information about data domain and user.
  - domain: A data domain to be queried. Defaults to "public". (obligatory for non-certificated access or for certificates with multi-domain access)
  - username: Obligatory for non-certificated access to private domains.
  - password: Obligatory for non-certificated access to private domains.

Typical units representation

<table>
<thead>
<tr>
<th>Unit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x84964924</td>
<td>Length</td>
</tr>
<tr>
<td>0x8492C924</td>
<td>Mass</td>
</tr>
<tr>
<td>0x84925924</td>
<td>Time</td>
</tr>
<tr>
<td>0x84924B24</td>
<td>Current</td>
</tr>
<tr>
<td>0x84924964</td>
<td>Temperature</td>
</tr>
<tr>
<td>0x8492492C</td>
<td>Amount of Substance</td>
</tr>
<tr>
<td>0x84924925</td>
<td>Luminous Intensity</td>
</tr>
<tr>
<td>0x849A4924</td>
<td>Area</td>
</tr>
<tr>
<td>0x849E4924</td>
<td>Volume</td>
</tr>
<tr>
<td>0x84963924</td>
<td>Speed</td>
</tr>
<tr>
<td>0x84962924</td>
<td>Acceleration</td>
</tr>
<tr>
<td>0x84929924</td>
<td>Irradiance</td>
</tr>
<tr>
<td>0x848EA924</td>
<td>Pressure</td>
</tr>
<tr>
<td>0x849A9724</td>
<td>Electric Potential</td>
</tr>
<tr>
<td>0x849E3924</td>
<td>Water Flow</td>
</tr>
</tbody>
</table>

Refer to EPOS user guide to check unit value composition.
2. REST API for Stationary Version

2.1. Query data

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

```json
{
    "series": Object
    {
        "version": unsigned char
        "unit": unsigned long
        "x": long
        "y": long
        "z": long
        "r": unsigned long
        "t0": unsigned long long
        "t1": unsigned long long
        "workflow": unsigned long
    }
    "credentials": Object
    {
        "domain": string
        "username": string
        "password": string
    }
}
```

Optional properties: workflow and credentials.
Note: We assume the following sizes for each type of field

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

See that during a query operation, even though a series record does not include a dev attribute, dev is specified here to filter SmartDatas accordingly. Moreover, during a query, the workflow attribute always refers to output workflow.

2.1.1. Data Aggregation

When querying data, it is possible to execute an aggregation function over it. This can be done by changing the requesting query by adding an aggregation function. Note that the aggregation is always executed before the specified output workflow.

```json
{
    'series': {
```
Time properties range, delay, and spacing expressed in us.

The list of available aggregators and their parameters is presented below.

- **min**: Returns the minimum value of an interval of range "range". Parameters: range.
- **max**: Returns the maximum value of an interval of range "range". Parameters: range.
- **mean**: Returns the mean of the set of values of an interval of range "range". Parameters: range.
- **lowerThan**: Returns the value if lower than a threshold defined by "parameter", else returns null. Parameters: parameter.
- **higherThan**: Returns the value if lower than a threshold defined by "parameter", else returns null. Parameters: parameter.
- **can**: Treats the data from the CAN port with an offset (represented by "range") and a given id ("parameter"). Parameters: range, parameter.
- **confidence**: SmartData confidence becomes its value. No parameters needed.

2.1.2. Fault Injection

Some aggregators have been designed in order to inject faults on SmartData time-series. They are:

- **drift**: Applies a drift of "parameter" to the values of the intervals of range "range" after an offset "delay" and repeating after every "spacing" (if this is higher than 0). The drift varies according to the number of samples it has been applied. Parameters: range, parameter, delay, and spacing.
- **stuckAt**: The values of the SmartData on the interval "range" becomes the first interval
SmartData value. Counts after an offset "delay" and repeating after every "spacing" (if this is higher than 0). Parameters: range, delay, and spacing.

- **constantBias**: Sums a value of "parameter" to every SmartData value of the intervals of range "range" after an offset "delay" and repeating after every "spacing" (if this is higher than 0). Parameters: range, delay, parameter, spacing.

- **constantGain**: Multiplies the value of "parameter" by every SmartData value of the intervals of range "range" after an offset "delay" and repeating after every "spacing" (if this is higher than 0). Parameters: range, delay, parameter, spacing.

- **allAnomalies**: Apply drift, stuckAt, constantBias and constantGain to different subsequent intervals of range "range", separated by "spacing" and counting after "delay". Parameters: range, delay, spacing, drift, stuck, bias, gain. If any of those (drift, stuck, bias, or gain) is 0, the respective aggregator is not applied.

### 2.2. Create series

- Create will only create a series if there is no other series that already encompasses its characteristics (space and time, device, unit, period and activation type). Moreover, it will merge every series that intersect with the new one, creating the smallest series that encompasses the intersecting series (modifying space and time of the oldest series). Moreover, the same unit of a specific domain is only allowed to have one Workflow for insertion, avoiding multiple insertions of the same SmartData. In this way, the method create follows the execution flow presented below.

**Method: POST**

**URL for create:** [https://iot.lisha.ufsc.br/api/create.php](https://iot.lisha.ufsc.br/api/create.php)

**Body:**

```json
{
  "series" : Object
  {
    "version" : unsigned char
    "unit" : unsigned long
    "x" : long
    "y" : long
    "z" : long
    "r" : unsigned long
    "t0" : unsigned long long
    "t1" : unsigned long long
    "workflow": unsigned long
  }
  "credentials" : Object
  {
    "domain" : string
    "username" : string
    "password" : string
  }
}
```
Series Semantics

When creating a series, it is important to specify some semantics of it, that can be recovered later when analyzing stored series. Mainly, the characteristics that define a series are:

- **activation**: if the series is created in a timely way, due to some condition regarding observed data, or (rarely) manually started;
- **data periodicity**: defines if data is collected at fixed time intervals (time-triggered), or only upon the occurrence of some events;
- **end condition**: determines if the series finishes after a specific amount of time, a number of data points, or upon some event.

The semantics of a series is extracted from the data it receives on creation. A complete description of the series semantics is available [here](https://epos.lisha.ufsc.br/).

2.3. Insert data

Method: POST

URL: [https://iot.lisha.ufsc.br/api/put.php](https://iot.lisha.ufsc.br/api/put.php)

Body:

```json
{
    "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
        }
    ]
    "credentials" : Object
    {
        "domain" : string
        "username" : string
        "password" : string
    }
}
```

Optional properties: credentials.
2.4. 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 +++++++++++++++++
```
2.4.1. 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  
    # ...
```

2.4.2. 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>")
...
```

2.4.3. 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>")
...
```

2.4.4. 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.

```php
... $smartdata = json_decode($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.

2.5. 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.

2.6. 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 Crendential 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.
3. 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 t1;
}
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;
    unsigned long long t;
}
```

3.1. 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>t1</td>
</tr>
</tbody>
</table>

3.2. Insert data (Binary)

Method: POST

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

Body: SmartData
3.3. 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),
};
```
4. Multi SmartData

To optimize the processing of multiple data originated at the same location, a set of multi-data formats are provided by the IoT API. Currently, the following structures are provided:

4.1. MultiValueSmartData

Useful to handle high-frequency sensors data, where buffering some values to save bandwidth is possible without compromising the time constraints. The data packet has a variable length and is composed of a fixed header of 27 bytes, plus N data blocks of 16 bytes each, as described by the following tables. At arrival, the packet size is verified for consistency.

4.1.1. Periodic Data and Uniform (or Unknown) Uncertainty

If the multiple values being inserted have a constant offset from each other (an exact periodic sampling), the **period** parameter can be used in the header, and the offset can be omitted in the datapoints. Also, if a constant **uncertainty** — possibly 0 — should be assigned to all datapoints, it can be specified in the header.

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

JSON format:

```json
{
    "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 informed, ommit it in datapoints
    "period" : unsigned long //OPTIONAL. if informed, ommit offset in datapoints
    "datapoints": [       // OPTIONAL, not used if period informed in the header
        {                  // OPTIONAL, not used if informed in the header
            "offset" : unsigned long
            "value" : double
            "uncertainty" : unsigned long
        }
    ]
}
```
In the binary format, the byte flag will carry bit 0 set if the **period** is defined in the header, and bit 1 set if the **uncertainty** is defined in the header. Otherwise, flag will be 0 (zero). Therefore, the packet header can have a length of 30, 34 or 38 bytes.

Binary Format:
Packet Header (first 29 bytes):

<table>
<thead>
<tr>
<th>Byte</th>
<th>29</th>
<th>25</th>
<th>21</th>
<th>17</th>
<th>13</th>
<th>5</th>
<th>1</th>
<th>0</th>
<th>(4)</th>
<th>(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>version</td>
<td>unit</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>t0</td>
<td>dev</td>
<td>flag</td>
<td>period</td>
<td>uncertainty</td>
</tr>
</tbody>
</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 datapoint 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>12</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>offset</td>
<td>value</td>
<td>uncertainty</td>
</tr>
</tbody>
</table>

### 4.2. MultiDeviceSmartData

When a device is responsible to monitor several devices of the same SI Unit, it provides a mechanism to sent all datapoints in a single request.

**Important:**

The **device** field **must** start at 0 for each SI unit at the same location. It is used only for disambiguation. When all sensors on a specific location are all of different SI units, **all** have to use device = 0.

Method: POST
URL: https://iot.lisha.ufsc.br/api/md_put.php
Body: MultiDevice SmartData
JSON Format:
4.3. MultiUnitSmartData

Allows a monitoring station, managing several sensors of different units, to pack a set of datapoints into a single request to the IoT API.

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

JSON Format:

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

Binary Format:
Packet Header (first 21 bytes):

<table>
<thead>
<tr>
<th>Byte</th>
<th>20</th>
<th>16</th>
<th>12</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>version</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>t0</td>
</tr>
</tbody>
</table>

Packet Payload (N x 24 bytes):

<table>
<thead>
<tr>
<th>Byte</th>
<th>20</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>unit</td>
<td>offset</td>
<td>value</td>
<td>dev</td>
<td>uncertainty</td>
</tr>
</tbody>
</table>
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:

```
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),
        't1': 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
```

https://epos.lisha.ufsc.br/  22/09/2020  22
6.2. R

6.2.1. Get Script Example

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

```python
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)
```

```r
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,
    "t1":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":
  }
json_body <- capture.output(cat(json_body, time_0 - 5*60*1000000))
json_body <- capture.output(cat(json_body, ','))
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:

```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 Horstmann, Roberto Scheffel</td>
<td>Review for Insert Changes and ADEG.</td>
</tr>
</tbody>
</table>