English

# Asema IoT Central
Object data parsing and serializing

ASEMA

# Table of Contents

1. Introduction ..... 1
2. Schemas ..... 2
3. Server side API: Feeding data into objects ..... 5
    3.1. Writing a feed schema ..... 5
    3.2. Creating a Server side API object ..... 6
        3.2.1. Identifier formats ..... 6
        3.2.2. Identifier placement ..... 7
    3.3. Pushing data to the feed API ..... 8
4. Client side API: Fetching data ..... 10
    4.1. Request client functionality ..... 10
    4.2. Writing a client schema ..... 10
        4.2.1. Authenticators ..... 10
        4.2.2. Serializers ..... 11
        4.2.3. Parsers ..... 14
        4.2.4. Templates ..... 15
    4.3. Creating a client object ..... 16
5. Database API: objects from raw data stored in a database ..... 17
    5.1. Objects that represent database data ..... 17
    5.2. Writing a database object schema ..... 17
    5.3. Filtering conditions ..... 19
    5.4. Raw queries ..... 20
    5.5. Triggers ..... 21
    5.6. Creating a database object ..... 21
6. Smart API: working with semantic data ..... 22
    6.1. A basic parser ..... 22
    6.2. Unit and quantity specifications ..... 22
    6.3. Application specific parsers ..... 22
7. Hardware API: Receiving data from directly connected hardware ..... 23
    7.1. Introduction ..... 23
    7.2. Bluetooth Classic ..... 23
    7.3. Bluetooth beacons - Eddystone and iBeacon ..... 24
    7.4. NFC ..... 24
    7.5. Serial port data ..... 24
        7.5.1. Writing a parser schema for serial data ..... 24
    7.6. ModBus ..... 26
        7.6.1. Writing a parser schema for ModBus ..... 26
8. How to define data parsers ..... 28
    8.1. Parsing various data formats ..... 30
        8.1.1. Parsing JSON ..... 30
        8.1.2. Parsing XML ..... 30
        8.1.3. Parsing CSV ..... 30
        8.1.4. Parsing bytestreams ..... 30
    8.2. Examples of parsing various data formats ..... 30
        8.2.1. Parsing JSON ..... 30
        8.2.2. Parsing XML ..... 31
        8.2.3. Parsing CSV ..... 31
9. Debugging and troubleshooting ..... 33

# 1. Introduction

The Asema IoT Central functions, as the name implies, as a central hub in an Internet of Things application. At the core of IoT central are objects, which can be monitored, controlled and combined with each other to form various application logic.

The automatic behavior can be augmented and changed with the use of programming APIs that allow the creation, modification, control, and monitoring of these objects. Asema IoT Central can act both as a server and as a client in such interactions over network or as a client to a database. For this purpose, objects can be created in four distinct styles:

- Server-side API. The server side listens to API calls from various clients and lets the clients monitor and control objects.
- Client-side API. The client side creates clients that access external resources at given intervals or when a specific request takes place.
- Database API. The database API reacts to changes in the database of the Asema IoT Central. In case data either in the standard or custom tables/namespaces of the database changes, those changes are automatically reflected in the values of the objects and can then trigger further action through logical rules, monitoring scripts, etc.
- Hardware API. Direct connections to connected hardware devices.

The APIs can be customized with schemas to read and write data in various formats. Schemas for instance make it possible to create custom clients that authenticate themselves using various methods and understand the data from external sources. Therefore the schema effectively defines how a particular API object behaves.

Schemas are shared between objects so that you don't need to write one for each object separately. So for instance if you have an API that receives incoming data about a car, write a schema for a car and then assign that schema for all the cars that are individual objects in the system.

# 2. Schemas

A system that acts as a data hub, essentially performs three tasks

1. Receive data. The system takes data from some source such as connected IoT hardware, a shared datasource, or an online server.

2. Store and process data. The system calculates with the data, shows it to users, and stores it to a database.

3. Serve data. The data is made available through an API to other systems for further use.

Asema IoT defines these three roles in one definition file for objects. This definition file is called a schema. It is a text document in JSON format. Understanding the format of this definition document is key to customizing the system into various use cases. The extent of the schema is dependent on your use. Schemas can be very simple and contain only a bare minimum. In this case they can be used with the standard API methods the system offers. Alternatively they can contain more processing instructions to better identify the data and to mold data from external sources into a more usable format.

The fields of the schema tell how the object behaves and can be manipulated. Some fields are purely related to receiving, some to storing, and some to serving. As you'll see in the examples below, many of these tasks are symmetric so the same field may cover all three roles.

Let's first take a look at a very simple schema:

```
{
    "type": "standard",
    "jsonapi": {
        "version": "1.0"
    },
    "attrs": [
        {
            "property": "speed"
        }
    ]
}
```

The schema above is for an object that has one property: speed. Most likely it is used for some object that moves. As the schema defines Asema IoT's JSON API as the standard, not much more is needed. The speed can be set with the `setProperty` JSON method and read with the `getProperty` method. In user interfaces, the property is visible as "speed" and automatically adjusts its values. In the database of stored values, you will find the value with the key "speed" and in-memory objects will be populated with this property automatically.

The only field needed in attribute definitions is `property`. This is the only mandatory field and it simply defines the name of the property that stores values. Because of the standard approach, this name is enough for all three roles, input, output and storage.

> **Important**
> The JSON API is a standard feature of Asema IoT. So although you may modify the schemas to work with other input and output standards, as long as the schema has an attribute definition with a `property`field, that field will be available with the getter and setter methods of the JSON API.

But what if the data is not that simple in format? If it is fed in by some other system that uses completely different naming convention? This is where dedicated input related fields come into play. Here are the main definition fields used for defining input:

- `source`. This marks the name of the field in the input data. So for instance if the speed is in the input data in a field called "velocity" while your application uses "speed", then velocity would be the `source`and speed would be the `property`. Effectively your system then maps between velocity and speed.

- `datatype`. The type of data that is in the input: string, integer, float, byte.
- `start`. Used with byte streams. The number of the byte in the stream where the value starts at.
- `len`. Used with byte streams. The lenght (in bytes) of the value.
- `conversion`. A conversion factor for changing the value as it is read in.

Later chapters in this manual show how to use these fields in various use cases. For now, here is a hort example on what a schema that parses JSON with source fields could look like

```
{
    "type": "json",
    "first": {
        "in": "movementData",
        "attrs": [
            {
                "source": "velocity",
                "property": "speed",
                "datatype": "float"
            }
        ]
    }
}
```

Notice the extra fields `first` and `in`. These are parsing instructions. They say "take the first value found inside the structure movementData and read the field velocity, convert from float and store to speed". More details on parsing instructions can be found in Chapter 8, How to define data parsers.

How about the output side then? There are two ways to interface with Asema IoT: the JSON API and the Smart API. As we saw above, the definitions for JSON API are very straightforward. All you need is the `property` field and the rest is defined and enabled by default. While very easy, the JSON API is also somewhat limited. It cannot perform more advanced things such as conversions on the fly or interpretation of data with more complex logic. It is also bad at integrating with heterogeneous external systems. For this purpose there is the Smart API.

Let's look at the same use case with the Smart API interface definition

```
{
    "type": "standard",
    "smartapi": {
        "version": "1.0"
    },
    "attrs": [
        {
            "property": "speed",
            "identifier": "http://smart-api.io/ontology/1.0/smartapi#velocity",
            "quantity": "http://smart-api.io/ontology/1.0/smartapi#Velocity",
            "unit": "http://data.nasa.gov/qudt/owl/unit#KilometerPerHour",
            "label": "Speed",
            "unitLabel": "km/h"
        }
    ]
}
```

This definition now has more fields that help the system communicate in a globally recognizable way and define the variables according to standard definitions. We still have the mandatory `property` field in the schema but in addition there are standard definition fields which work for both input and output. Let's take a look at them in more detail:

- `identifier`. This is the global identifier of the value. Instead of our internal value, this identifier is predefined in an ontology file that defines what type of object a speed actially is. The identifier can be used to point to that specific value in a given external system.

- `quantity`. The quantity in which the value is measured. Each quantity can have various units (for example speed is measured in km/h, m/s or mph). A known quantity lets the system interpret the value and convert as necessary.
- `unit`. The unit in which we want to process the data, in this case kilometres per hour. In case incoming data is in a different unit, the API automatically converts it.
- `label`. Local label of the data. This is what user interfaces use to display the property.
- `unitLabel`. Local label of the unit. The local interfaces will append this label to the value when displaying it.

# 3. Server side API: Feeding data into objects

## 3.1. Writing a feed schema

An incoming data feed is defined by

1. an object that holds the values pushed into the system through the API; and

2. a schema that sets up a parser needed to parse the incoming data.

An example feed schema to parse incoming XML could look like this

```
{
    "type": "xml",
    "first": {
        "in": "computerData|measurementData",
        "attrs": [
            {
                "source": "system_load",
                "property": "system_cpu_load",
                "datatype": "integer",
                "conversion": "none"
            },
            {
                "source": "time",
                "property": "timestamp",
                "datatype": "datetime",
                "conversion": "yyyy-MM-dd HH:mm:ss"
            }
        ]
    }
}
```

This feed schema says that the API will receive data in XML format (defined with field `type`). This XML has a section called "computerData" and inside computerData there is a section "measurementData" (these are in the field `in` which defines in which structure values should reside). measurementData will further contain at least two tags: system_load and time that contain the actual values to extract. If there are multiple sections called measurementData in the incoming XML, only data from the first section is extracted (this is the field `first`).

Values from system_load will be stored into a property called system_cpu_load after converting to integer. Values inside the time tag are timestamps that have the format year-month-day hour:minute:second.

If on the other hand you would use this object with the standard JSON API, the schema would look like this

```
{
    "type": "standard",
    "jsonapi": {
        "version": "1.0"
    },
    "attrs": [
        {
            "property": "system_cpu_load"
        }
    ]
}
```

For a full explanation on how the parser schema works, documentation of each attribute of the schema, as well as more examples, please refer to Chapter 8, How to define data parsers.

**Important**

Note that Asema IoT Central supports multiple ways of pushing data into the system. The server side API feeds is just one of them. The other options are the

JSON API and the Smart API. Because Asema IoT Central allows you to connect to multiple systems, it also supports multiple ways for doing those connections. The choice depends on how much freedom you have in designing the interaction.

The important distinction between a custom feed and a standard feed is that of data interpretation. Both the JSON API and the Smart API use standard data formats mandated by the API. The schema in these cases does not define how the data is interpreted, only what the data is. In Smart API the actual details of the data are in the ontology file of the standard. This is where the data is defined centrally so that it can be used in a similar fashion anywhere.

With the feed schema you also have the option to define the structure of the data. Therefore you should use this method when you are integrating with an external system but do not have the option to modify the way the other end sends data. If you do have the freedom to modify the data format or are writing a client from scratch, then the JSON API or Smart API may be better options for implementation.

For details on the JSON API, see the separate JSON API manual. For details on Smart API, please refer to Smart API manuals.

## 3.2. Creating a Server side API object

Once a schema has been entered, a new object can be created from the Asema IoT Central web interface. This object must be given a name and a schema and most importantly **an identifier**. The identifier will help the API recognize to which object the data applies to when it is received. The format and placement of the identifier will depend on the type of object you create (see below for details).

**Important**

Pay attention to the format and location of the identifier as this is specific to the type of object you use. A Smart API identifier will not work with Feed API and vice versa. Asema IoT user interface will show you both options. If you use both standards for the object, you should enter values into both fields.

Note however that if you do not explicitly enter a Smart API or JSON API identifier, one is implicitly generated by the system so that there is always some identifier for the object in both APIs.

**Important**

Note that objects in Asema IoT can have one more identifier called the Component Identifier. This identifier has nothing to do with the API side, it is targeted towards user interfaces. The Component Identifier can be used to keep the access to application logic and user interfaces constant when applications are imported and exported between separate Asema IoT installations. So for example if one system receives temperature data from an object that is a "thermostat" and the other for a system called "weatherstation", the user interface can be kept standard by giving these the common Component Identifier "temperature_source", for instance. Do not confuse this concept with the API identifiers.

### 3.2.1. Identifier formats

Depending on the type of Server API object you create, the identifiers to use will differ somewhat

- In custom feed objects the identifier is a freeform string. You may give it any value. Note however that because the identifier can be placed in the URI, it is recommended not to use special characters that may interfere with URI encoding (such as ? and &).

- The JSON API always uses the GID of the object as the identifier. There is no need to specify an identifier separately (or if you do, the setting has no effect).
- With Smart API, you may either define the identifier or leave it empty. If the identifier is left empty, it will be automatically generated based on the GID and a standard URI generated from the domain name of your installation.

### 3.2.2. Identifier placement

To be readable, the identifier must be placed in the correct location in the incoming data. Naturally, it is the responsibility of the client software to do so. Depending on the type of object you create, the placement will be different

- With custom feed objects the identifier can be placed either in the HTTP/CoAP headers or in the call URI. If in headers, the header is called `x-source-id`. If in URI, it is placed right after the `/feed`path, e.g. http://myserver.org/feed/myobject/
- In JSON API the identifier will be placed in the `gid`field of the incoming payload. Please see JSON API manual for details.
- In Smart API the identifier will be placed in the identifier fields of the incoming payload. Please see Smart API manual for details.

Below are three examples of incoming payloads for one object in the three formats listed above. Let's assume this object is called "my_car" and it has a random Global Identifier (GID) generated by the system "a47792c27eec4f82fa4795cf35e947af5e22ba9". The system runs under the domain http://iot.examplecompany.org. Notice how the global uniqueness and detectability of the object improves as we move from the Feed API (worst uniqueness) towards the Smart API (best uniqueness).

Notice also how the detail of the message increases. In the Feed API it is very vague what is actually to be done with the object by whom (it is implicitly assumed this is a write operation) while the Smart API explicitly states who does what and when and by which definitions. While the latter requires much more detail in the message, it also makes it possible to be more exact in things like access control and other management. The data becomes more cumbersome for humans to handle but then again that is not really a problem as it is created and interpreted by a library anyways.

Feed API:

```
POST /feed HTTP/1.1
Host: iot.examplecompany.org
Connection: keep-alive
Content-type: application/json
x-source-id: my_car

{
 "movementData": {
  "velocity": 24.5
 }
}
```

JSON API:

```
POST /json HTTP/1.1
Host: iot.examplecompany.org
Connection: keep-alive
Content-type: application/json

{
 "id": 1,
 "method": "set_object_property",
 "params": {
  "gid": "a47792c27eec4f82fa4795cf35e947af5e22ba9",
  "property": "speed",
  "value": 24.5
 }
}
```

Smart API:

```
POST /smart/v1.0e1.0/access HTTP/1.1
Host: iot.examplecompany.org
Connection: keep-alive
Content-type: text/turtle

@prefix smart: <http://smart-api.io/ontology/1.0/smartapi#> .
@prefix nasa: <http://data.nasa.gov/qudt/owl/qudt#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://iot.examplecompany.org/objects/Ca47792c27eec4f82fa4795cf35e947af5e22ba9> a smart:Entity ;
    smart:valueObject <http://smart-api.io/ontology/1.0/smartapi#velocity> .

<http://smart-api.io/ontology/1.0/smartapi#velocity> a smart:ValueObject ;
    nasa:quantityKind smart:Velocity ;
    nasa:unit nasa:KilometerPerHour ;
    rdf:value 2.45e+01 .

[] a smart:Request ;
    smart:activity [ a smart:Activity ;
            smart:entity <http://iot.examplecompany.org/objects/Ca47792c27eec4f82fa4795cf35e947af5e22ba9> ;
            smart:method smart:Write ] ;
    smart:generatedAt "2018-04-20T13:33:16.246050"^^xsd:dateTime ;
    smart:generatedBy <http://demo.iot.asema.com/demosystem> .
```

## 3.3. Pushing data to the feed API

Pushing data to the feed API simply means sending a HTTP POST message to `http://<server ip and port>/feed/` on your Asema IoT Central installation. The payload of the POST must conform to the structure you have defined in the schema.

**Important**

Note that the URL `http://<server ip and port>/feed/` explicitly addresses the feed parser which uses your schema to interpret the data. If you use the JSON API, the URL is `http://<server ip and port>/json/` and if you use the Smart API the URL is `http://<server ip and port>/smart/`. The data for these APIs is described in more detail in their corresponding manuals.

Below is a sample script that sends data through the Feed API. In this case the data is in CSV format. The script forms the correct headers, puts the values into the body of the message and posts it to the server.

```
#!/usr/bin/python

import urllib, urllib2

SERVER_IP = "127.0.0.1"
SERVER_PORT = 8080
FEED_ID = "myobject"

def push_csv_request(value_list):
 values = ";".join(value_list)

 url = "http://" + SERVER_IP + ":" + str(SERVER_PORT) + "/feed/"
 headers = { "content-type" : "text/plain", "x-source-id": FEED_ID }
 try:
  req = urllib2.Request(url, values, headers)

  # use a 5s timeout
  filehandle = urllib2.urlopen(req, timeout = 5)
  if filehandle is not None:
   data = filehandle.read()
   result = data
 except:
  print "Failed in contacting", url
  finally:
   return result

def send_value():
 vals = ['1', '2', '3', '4', '5', '6']
 push_csv_request(vals)

def main():
 send_value()

if __name__ == '__main__':
 main()
```

The corresponding schema that takes those values (1, 2, 3...) and puts them into object properties "one", "two", "three", etc would look like this:

```
{
    "type": "csv",
    "direction": "column",
    "first": {
        "attrs": [
            {
                "source": 1,
                "property": "one",
                "datatype": "int"
            },
            {
                "source": 2,
                "property": "two",
                "datatype": "int"
            },
            {
                "source": 3,
                "property": "three",
                "datatype": "int"
            },
            {
                "source": 4,
                "property": "four",
                "datatype": "int"
            },
            {
                "source": 2,
                "property": "five",
                "datatype": "int"
            },
            {
                "source": 6,
                "property": "six",
                "datatype": "int"
            }
        ]
    }
}
```

# 4. Client side API: Fetching data

## 4.1. Request client functionality

Request clients are, as the name implies, clients that connect to some server in order to fetch data from an API residing at that server. More specifically, a request client can

- authenticate itself to the server
- request the data based on a known URI and parameters embedded into the URI or serialized into the payload or header
- parse the incoming data and feed to into the rest of the system.

Request clients run either when scheduled to do so or when requested to do so. A scheduled client simply reruns the query after a set number of seconds. A client triggered by a request runs its query when some other query into the Asema IoT Central requires values from the object representing the client.

Similar to many other objects in Asema IoT Central, the format processed by the client is defined by schemas and then defined as an object. Note however that unlike other objects that have just one schema, clients have two because data traffic is bidirectional:

1. a serializer schema that modifies data to be sent, and
2. a parser schema that parses the data being received.

Do note that if the client does not need to send a payload or specific header when it makes the request, there is no need for a serializer schema either. In this case the serializer schema is simply an empty document.

**Important**
As with server side APIs, the client side API also supports multiple means for defining how the client operates. In addition to the basic client which takes the format from your custom schema, you can also write a Smart API client. For details on that, please refer to the Smart API manual.

## 4.2. Writing a client schema

### 4.2.1. Authenticators

Authenticators take care of authenticating ("logging in") into an external service. The forms of authentication supported currently are

- HTTP Basic authentication
- Cookie authentication / forms login
- OAuth

**Important**
Note that authentication is typically a part of the request. This is why in schemas the authenticator is written as a part of the serializer.

Authenticators do their authentication process either at each request (in the case of HTTP Basic authentication) or before the first request to the target. In case a session has expired, the authentication is redone after receiving an error of denied access.

Below is an authenticator that logs in into a PHP server and obtains a cookie from it. The components of an authenticator are

- Type. Autheticator type defines whether the authentication is done with a cookie, a HTTP Basic authentication header or OAuth token.
- The target. This tells where the authenticator is to be placed when accessing the server. In this case the cookie goes to the HTTP headers as a heades called PHPSESSID.
- The source. This says from where the cookie can be obtained. In this case it is a PHP page called login.php. The authenticator will send this page an URL encoded body which contains the fields username, password and myextradata with the corresponding values. The cookie will be returned in the body and can be parsed with the regex that is within the "in" declaration.

```
"auth": {
    "type": "cookie",
    "target": {
        "location": "header",
        "field": "PHPSESSID"
    },
    "source": {
         "url": "https://targetsystem.example.com/login.php",
         "params": {
             "location": "body",
             "format": "urlencode",
             "values": {
                 "password": "mypass",
                 "username": "myuser",
                 "myextradata": "somedata"
             }
        },
        "at": "body",
        "in": "^token:([^:]+):token$"
    }
}
```

## 4.2.2. Serializers

Serializers modify the payload sent to the server, specify the authentication needed, and list the set of requests done per each update round. One serializer can define multiple requests. So for instance in case you need to update the location and velocity of an object and both variables require a separate request, you can bundle the requests into one update round.

Below is an example of a serializer that does two queries, one for velocity, one for position for some object that has id 2. Each serializer must specify

- Location. Where will the serialized data be put. Url, header or payload.
- Format. What will the format of the data be.
- Attrs or raw. Either specify data as raw (will not be modified) or as attributes that will be taken from the object.

### 4.2.2.1. Raw payload serialization

Let's first take an example of raw payloads. The raw payload as a concept is simple: whatever you enter into the field `raw`, will be placed into the request as such. Here is an example:

```
{
```

```
"requests": [
    {
        "location": "url",
        "format": "urlencode",
        "raw": "/service.php/GetVelocity/?id=2"
    },
    {
        "location": "url",
        "format": "urlencode",
        "raw": "/service.php/GetPosition/?id=2"
    }
]
}
```

In the example above, there are two requests (the client will actuall make two HTTP calls to get the values). There is no payload data so the requests will be GET. The location of the data will be the URL of the call. The server address will be entered into the object itself when an object using this schema is added to the system. Let's assume for example that the server URL entered will be www.examplecor-p.com. This schema would result in two get calls:

- GET http://www.examplecorp.com/service.php/GetVelocity/?id=2
- GET http://www.examplecorp.com/service.php/GetPosition/?id=2

Whatever data those two calls return will then be parsed according to the instuctions in the response schema (the other schema of the client type object).

#### 4.2.2.2. Attribute based data serialization

Attribute data serialization takes values from the actual object properties and feeds that into the target system. So essentially while the serialization lets you request data from a target system in a dynamic fashion, you can actually feed the data into that target system while you do that. Actually if you leave the parser schema empty, the client object actually becomes a data pusher, not a requester.

There are three formats the attribute data serialization supports: raw bytes, JSON and XML. As a format, the serializer schemas use pretty much the same format and just make a mirror image of the parser process. Let's take a look at them, starting with the byte format. Here is an example:

```
{
"requests": [
    {
        "location": "body",
        "format": "byte",
        "preamble": [ 128, 0, 128 ],
        "endofframe": [ 1, 0, 16 ],
        "attrs": [
            {
                "property": "temperature",
                "length": 4
            },
            {
                "property": "humidity",
                "length": 4
            }
        ]
    }
]
}
```

First note the two common fields `location` and `format`. They say that the data should be in the main body (in the case of bytes in raw TCP stream or an HTTP message) and the format is bytes.

Next, as this is a byte stream, it most likely needs some type of frame. The frame is defined by a preamble and an end-of-frame marker. These are in the corresponding fields of the schema and the values are given in decimal.

The `attrs` list specifies the properties which will be taken from the object and transformed into bytes. If no datatype is given (as in the example), the values are assumed to be integers. The length says how many bytes will be used to encode the value. The values will be placed into the payload in the order they are in the schema.

So if the value of temperature for example is 24 decimal and humidity is 99 decimal, this example will result in the following bytestream (in hex): `0x80 0x0 0x80 0x0 0x0 0x0 0x18 0x0 0x0 0x0 0x63 0x01 0x0 0x10`. Note the preamble and end-of-frame bytes.

Next, let's look at a JSON format schema. Here's an example using the same properties as the byte format schema.

```
{
"requests": [
    {
        "location": "body",
        "format": "json",
        "in": "measurementData|environmentData",
        "attrs": [
            {
                "property": "temperature"
            },
            {
                "property": "humidity"
            }
        ]
    }
]
}
```

Again, the two common fields `location` and `format` will tell that this schema generates a JSON message and puts it into the body. As with JSON parsers, the `in` field defines the structure within which the data should be put. Finally the `attrs` tell which properties to take from the object. This example would then result in one request that has the following JSON in the body of the message:

```
{
    "measurementData": {
        "environmentData": {
            "temperature": 24,
            "humidity": 99
        }
    }
}
```

Finally XML. As you already know how a JSON schema is created, there is not much extra to learn in making an XML message. All you have to do is to change the format. Like so:

```
{
"requests": [
    {
        "location": "body",
        "format": "xml",
        "in": "measurementData|environmentData",
        "attrs": [
            {
                "property": "temperature"
            },
            {
                "property": "humidity"
            }
        ]
    }
]
}
```

As you can see, the format of the schema is otherwise exactly the same as with JSON. The resulting request would now look like this:

```
<?xml version="1.0">
<measurementData>
    <environmentData>
        <temperature>24</temperature>
        <humidity>99</humidity>
    </environmentData>
</measurementData>
```

Finally, what if the names of the properties of your object in the Asema IoT system differ from the names in the other system? For this conversion, you use the `source` field. The value of this field says what is the name of the field at the other end (while `property` is still the name at local side). Let's try this schema:

```
{
"requests": [
    {
        "location": "body",
        "format": "xml",
        "in": "measurementData|environmentData",
        "attrs": [
            {
                "property": "temperature",
                "source": "thermoReading"
            },
            {
                "property": "humidity",
                "source": "humReading"
            }
        ]
    }
]
}
```

Now the serialization would look like this:

```
<?xml version="1.0">
<measurementData>
    <environmentData>
        <thermoReading>24</thermoReading>
        <humReading>99</humReading>
    </environmentData>
</measurementData>
```

while the values of humidity and temperature are still taken from local properties called "temperature" and "humidity".

### 4.2.3. Parsers

A parser will parse the data received by the client and then store it as attributes into the corresponding client object. Below is a sample parser for CSV data. It says that the data is columnar i.e. each row represents a new dataset and each column an attribute of that dataset. In this case we want to take the values from the second row (nth, at 2) and the values we are interested in are in columns 4 and 5. After conversion to floats, the value from column 4 will be stored into "acceleration" and value from column 5 to "velocity".

```
{
    "type": "csv",
    "direction": "column",
    "nth": {
        "at": 2,
        "attrs": [
            {
                "source": 4,
                "property": "acceleration",
                "datatype": "float"
            },
            {
                "source": 5,
                "property": "velocity",
                "datatype": "float"
            }
        ]
    }
}
```

### 4.2.4. Templates

Templates are user by the preprocessor to transform the incoming data by using simple variable substitution. The template has slots for variables and bytes in the payload are used to fill these slots. Each slot has an identifier, the templating engine will seek these identifiers and replace each occurrence of the identifier with the value. For instance, a template might look like this.

```
<mydata>
    <tag1>%first</tag1>
    <tag2>%second</tag2>
    <tag3>%third</tag3>
<mydata>
```

If the incoming payload is three HEX bytes 0x01 0x02 0x03, then these three values are used in the order they are in the payload to produce this

```
<mydata>
    <tag1>1</tag1>
    <tag2>2</tag2>
    <tag3>3</tag3>
<mydata>
```

How the substitution is done is defined in the preprocessor section of the data parser. The preprocessor defines

- The template itself, inserted either as actual template data or as a link to the template
- The format of the incoming byte data

The template can be given as an URL in the `template-uri` parameter or included inside the preprocessor section itself as a base64 encoded string which is placed into the `template` parameter.

The data section then describes the byte data itself by specifying, for each variable in the data, the length of the variable in bytes, the datatype these bytes should be converted into, and the identifier in the template that should be replaced.

To create a base64 encoded template, you can use various tools. For easy online encoding, simply go to https://www.base64encode.org/, paste your template and click convert. If you're using Linux, base64 encoding is available as command line tool `base64`. You can also generate base64 encoded strings in pretty much all programming languages. In Python, use the `base64` package. In Java, use the `java.util.Base64` package and in C# do `Convert.ToBase64String`. Once you know the tool, write the template, then run it through a converter and copy-paste to your schema into the "template" field.

In case the template is on a server, insert the URI of the template. Asema IoT will fetch the template with a simple HTTP GET.

```
"preprocessor": {
    "template-uri": "http://mytemplateserver.example.com/templates/sampletemplate",
    "data": {
        "format": "byte",
        "values": [
            {
                "length": 2,
                "datatype": "float",
                "identifier": "%first"
            },
            {
                "length": 2,
                "datatype": "float",
                "identifier": "%second"
            },
            {
                "length": 4,
                "datatype": "integer",
                "identifier": "%third"
            }
        ]
    }
}
```

**Important**
If you put the template to a server it should **not**be base64 encoded.

## 4.3. Creating a client object

To create a client object, choose to Add new in Client objects and then fill in the requested data, including a name, the address of the server to connect to, the path at the server (if required), network protocol, and the schemas.

To immediately test your newly added client, click on Analyze. This will open up a pop-up that lets you send the request to the server. The window will show

- To what URL the request was sent
- What the payload, if applicable, of the request was
- The received data, in raw format as sent by the server
- The resulting data after it has been parsed.

# 5. Database API: objects from raw data stored in a database

## 5.1. Objects that represent database data

If you have direct access to the database and would like to write data to it using e.g. SQL queries and then serve that data through an API, database objects are meant for this purpose. Database objects are useful especially in the following situations:

- You have some legacy data in a database and cannot extract that in any other way than reading it directly from there.
- You want to make statistical analysis of database data and the program used for the analysis is limited to just writing the resuls back to the database.

Database objects will monitor the data in the database and load values to itself when a change has been detected. This data is then automatically available to APIs and is updated to visualizations.

**Important**
Database objects currently work with traditional SQL relational databases only. NoSQL databases are currently not supported.

A database data object is in practice a wrapper around SQL queries that are either formed automatically or written manually. When a database data object runs, it executes that query, extracts values from it and puts those values into the object properties. Once values are stoed in properties, they can be used with the same property APIs and UI methods as any other object in Asema IoT.

A database data object can monitor / read the values in database tables in three different modes

1. Per request. The data is fetched from the database only when values are queried by some external object or API that invokes a `getProperty` method.
2. Per timer. Data is reloaded from the database at regular intervals.
3. Per trigger (available currently with Portgreql only). A database trigger is added to notify of changes so that no polling of values is required. When a change notification is received, values get reloaded.

## 5.2. Writing a database object schema

The database object schema defines how entries in a database (e.g. rows and columns in a relational database) are transformed into values of objects. To enable this the schema needs to contain information about what kind of database is used and what property names correspond to what database queries. Whenever an update is done to an object, the schema will be read and the contents are transformed to an SQL query that runs against the defined database.

Below is an example schema that shows how to define a database object. Let's first explain what that contains, starting with the basic definitions.

```
{
    "name":"weather schema",
    "database":{
        "type":"postgresql",
        "schema":"my_db_server",
        "table":"measurements",
        "identifierField":"gid",
        "timestampField":"time",
        "sortField":"time",
        "sortOrder":"desc"
    }
}
```

The name of the schema is in the `name` attribute, that's simple. The real beef is the core of the schema which is in the `database` attribute. This tells what kind of database is used and in which database schema and table data is stores. First, `type` tells the brand of the database to operate with. Valid values are `postgresql`, and `sqlite`. Second, the schema defines the namespace/schema in which the data is in the `schema` attribute as well as the table in the `table` attribute. In practice `schema` is the name of the database. So if you have a database called "animals" and data in a table called "dogs", set "animals" as the value of `schema` and "dogs" as the value of `table`.

**Important**
Note that SQLite can only host one database so the concept of a namespace has no meaning. Therefore if you set the value of `type`to `sqlite`, you can leave the field `schema`out completely.

There are two additional important attributes to set: `identifierField` and `timestampField`. `identifierField` identifies a particular record. This will be matched with the value of identifier in the actual object you will set in the admin user interface when you create an actual objecy. For example, if you have the table "dogs" and this table has a column "name", you can use this to fetch into the database object only the data of a dog with a particular name. For example, if the dog that the your object should represent is called "Eddy" and this is stored in the column "name" in your database, you would set "name" as the value of `identifierField` in the schema and then write "Eddy" into the field labeled Database identifier in the actual object settings form at the admin interface. To create another object that represents the dog "Clara", you'd use the same schema for it, just write "Clara" as the value of the Database identifier field.

`timestampField` is the field used to limit the values in a timeseries query (graph of historic values). Asema IoT will run a query of type `WHERE <timestampField> >= <start> AND <timestampField> <= <end>` with that field. Note that it is highly recommended to create an index for both `identifierField` and `timestampField` in your database as it will speed up the queries considerably if your database is large.

The query you use for fetching values into properties will use a standard select and the order of results will depend on the setup of the database you use. So if you have multiple values in a database but use those to update just one property, only one value will be selected of those multiple choices. Which one will depend on your database. If you want to affect this choice, use the `sortField` and `sortOrder` parameters. `sortField` will make the select query sort the results by the column specified by this parameter and pick the one that is first in that sort. You can define the ordering with `sortOrder`. Valid values for this parameter are `asc` and `desc` for ascending and descending ordering, respectively.

**Important**
Note that the sort order has no effect on timeseries queries (multiple values). These will always be sorted by the `timestampField`.

The database definition is followed by a list of properties. Each listed property will become a property of the object created from database data. A property needs to have a name unique from other names in the same object. The name serves as a key to the rest of the definition. The definition then contains the type of the property and its ID.

The query is entered using a parameter value that always corresponds to the type of the database in use. So if you are using a Postgresql database, you should have a parameter called `postgresql` within the definition of the property. In the definition you must have a `columnName` and `dataType` and an optional set of filters. If no filters exist, the query is performed to all values in the database and the first returned value is converted into the value of the property. For example, the schema in the example below, would result in the following SQL

```
SELECT centigrade, cloudiness from measurements where gid = '<value set to identifier>' ORDER BY time DESC;
```

```
{
    "name":"weather schema",
    "database":{
        "type":"postgresql",
        "schema":"my_db_server",
        "table":"measurements",
        "identifierField":"gid",
        "timestampField":"time",
        "sortField":"time",
        "sortOrder":"desc"
    },
    "properties":{
        "temperature":{
           "type":"float",
           "id":1,
           "postgresql":{
               "columnName":"centigrade",
               "dataType":"double"
           }
        },
        "cloudiness":{
            "type":"float",
            "id":2,
            "postgresql":{
                "columnName":"cloudiness",
                "dataType":"double"
            }
        }
    }
}
```

Note that when you are dealing with dataseries, you'll most likely be fetching and storing data that has a timestamp. If you don't specify otherwise, Asema IoT Central will assume that there is a column called "time" where this value is stored. If this is not the case and you want to use some other column name, you can define this in the schema with the `timestampColumn` field. For example like so:

```
{
    "properties":{
        "cloudiness":{
            "type":"float",
            "id":2,
            "sqlite":{
                "columnName":"cloudiness",
                "timestampColumn":"recordedDate",
                "dataType":"double"
            }
        }
    }
}
```

## 5.3. Filtering conditions

Sometimes it may be desirable to further limit the values fetched from the database. You can do this with filter conditions. Filter conditions will try to match a column value with the given filter value and return only those that do. Let's write the cloudiness property again, now with a filter.

```
"cloudiness":{
    "type":"float",
    "id":2,
    "postgresql":{
        "columnName":"cloudiness",
        "dataType":"double",
        "filters": [
            {"columnName": "precipitation", "columnValue": 10}
        ]
    }
}
```

The shema above will result in the following query:

```
SELECT cloudiness from measurements where gid = '<value set to identifier>' AND precipitation = 10;
```

## 5.4. Raw queries

Basic property queries will try to form query statements according to rules that work most of the time. And then there are times when no previous definition is simply not enough. For that purpose you can write queries manually as raw queries. Let's write a raw query for cloudiness we've used before. First start with the schema.

```
"cloudiness":{
    "type":"float",
    "id":2,
    "query":{
        "name":"getCloudiness",
        "resultIndex":"1"
    }
}
```

This tells the system to run a query called "getCloudiness" and extract the value from the first value returned by the query and put that as the value of the property. Now, how does the system know what getCloudiness looks like? For that purpose, you need to add another section into the schema called `queries`.

The `queries` section should be placed at the top level of the schema and contain a dictionary of queries to run. Each entry is identified by the query name and has the SQL of the query as its value. Like so

```
{
    "name":"weather schema",
    "database":{
        ...
    },
    "properties":{
        ...
    },
    "queries":{
        "getCloudiness": {
            "query": "SELECT cloudiness FROM stats WHERE gid = '<some value>'"
        }
    }
}
```

Now, the raw query definition above if fine except that it has `gid` hardcoded in it. This may be fine in some queries but in many cases you'd like to make such conditions object dependent. This is done by including dynamic elements into the raw query. For example:

```
{
    "name":"weather schema",
    "database":{
        ...
    },
    "properties":{
        ...
    },
    "queries":{
        "getCloudiness": {
            "query": "SELECT cloudiness FROM stats WHERE gid = '%1' AND name = '%2'"
        }
    }
}
```

When the system recognizes a schema like the above, it will automatically add a new entry table for the object. You can enter the values of the parameters into this table for each object. These values will replace the placeholders %1 and %2 in the above query.

## 5.5. Triggers

**Important**
The methods in this chapter work with Postgresql databases only.

A trigger is a database method that can send an event from the database to the system using the database when some event happens. This is great for real-time applications because the system can react immediately to changes in database values without having to wait for a request or a poll cycle.

A trigger has a notifier name that is programmed to it. In the example below, this is done with the code `pg_notify('mytrigger' ...`. This trigger will therefore be called "mytrigger". When you set a database object to reach to triggers, you should enter a value that matches the name of the trigger to the settings of the object (in this case the value would be "mytrigger").

The trigger itself does not send any values and does not need to query them. It simply notifies the IoT system that an event has happened. As a response, the recipient will then perform a query to fetch those values.

In Postgresql triggers are programmed as plpgsql functions. This function is tied to the desired table and event (INSERT, UPDATE, DELETE) that should take place. The actual dirty work is done by the `pg_notify` function which is captured by the database driver of Asema IoT Central.

Below is an example trigger that sends a notification every time an UPDATE is done to the table "stats".

```
CREATE OR REPLACE FUNCTION add_measurement_trigger_notify_function()
RETURNS trigger AS
$BODY$
BEGIN
IF (TG_TABLE_NAME = 'stats' AND TG_OP = 'UPDATE') THEN
 RAISE NOTICE 'TRIGGER called on %', TG_TABLE_NAME;
 PERFORM pg_notify('mytrigger', CAST(NEW AS varchar));
 RETURN NEW;
END IF;

RETURN null;
END;
$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
ALTER FUNCTION add_measurement_trigger_notify_function() OWNER TO mydatabaseuser;

CREATE TRIGGER add_measurement_trigger_notify_trigger
BEFORE INSERT OR UPDATE OR DELETE
ON stats
FOR EACH ROW
EXECUTE PROCEDURE add_measurement_trigger_notify_function();
```

For more info on how to program triggers, please refer to the Postgresql user manual.

## 5.6. Creating a database object

To create a database object, simply click on Add new at Database objects. Then fill in the name of the object, choose how it is to be updated, and pick a schema from the list of available database object schemas.

# 6. Smart API: working with semantic data

## 6.1. A basic parser

```
{
  "type": "standard",
  "smartapi": {
    "version": "1.0",
  }
}
```

## 6.2. Unit and quantity specifications

```
{
  "type": "standard",
  "smartapi": {
    "version": "1.0",
  },
  "attrs": [
    {
      "property": "temperature",
      "identifier": "http://smart-api.io/ontology/1.0/smartapi#valueObject",
      "quantity": "http://data.nasa.gov/qudt/owl/quantity#ThermodynamicTemperature",
      "unit": "http://data.nasa.gov/qudt/owl/unit#DegreeCelsius",
      "label": "Temperature",
      "unitLabel": "degC"
    }
  ]
}
```

## 6.3. Application specific parsers

```
{
  "type": "standard",
  "smartapi": {
    "version": "1.0",
    "parser": "mycustomapplicationparser"
  }
}
```

Note that the plugin must be placed under the correct version number of plugins. In this case that would be `plugins/smartapi/1.0/`.

# 7. Hardware API: Receiving data from directly connected hardware

## 7.1. Introduction

By assumption nearly all data in IoT applications originates from some measurement hardware. But the methods described ealier in this document don't actually deal with a direct hardware protocol connection. With server and client side APIs, the assumption is that the hardware is connected over some IP connection. Either the device is capable of talking IP directly or there is a gateway device that does the hardware connection and converts it into some IP packets with data.

So what if there is no such capability? If no gateway is available and the device to connect to does not talk over IP? This is where direct hardware objects come into play. They are made to interface with raw hardware communication protocols directly without IP conversion in between.

## 7.2. Bluetooth Classic

Asema IoT Central supports serial data over Bluetooth. To easily create a Bluetooth network between two devices, you add a Bluetooth server on one side and a Bluetooth client on the other. Then define the data to transfer with a schema. When the server and the client have the same schema, the data will be automatically and correctly packed and extracted by the systems on each side.

As the Bluetooth data is binary over a serial connection, the schema for the data is very similar to any serial object. You'll find the description of such data in the section for serial communication in this manual.

So let's get directly to it. Here's an example schema for Bluetooth Classic objects:

```
{
    "type": "byte",
    "cutout": {
         "length": 12
    },
    "preamble": [ 128, 10, 128 ],
    "single": {
        "attrs": [
            {
                "datatype": "float",
                "len": 4,
                "property": "pressure",
                "start": 0
            },
            {
                "datatype": "float",
                "len": 4,
                "property": "altitude",
                "start": 4
            },
            {
                "datatype": "float",
                "len": 4,
                "property": "humidity",
                "start": 8
            }
        ]
    }
}
```

The capacity of Bluetooth Classic is sufficient to transfer data also in richer format, say JSON, but that is in most cases waste of capacity. As both sides of communication use the same schema anyways, a pure binary protocol packs the data in the most efficient way.

## 7.3. Bluetooth beacons - Eddystone and iBeacon

Eddystone and iBeacon are standard formats for data contained in Bluetooth Smart (aka Bluetooth LE, Bluetooth 4.0) beacons. The beacon is one directional: there is no connection nor any way to command the device. But the beacon is excellent for small long battery life senders.

Both Eddystone and iBeacon contain data in the form of a UUID field. Additionally, Eddystone has the so called URL message, that contains a free-form URL. Because the devices are often recognized by their MAC addresses, which are independent of UUIDs, those fields are available for programming an use. You can program the data in the beacon signal with most of the beacon models. Often manufacturers offer a mobile phone app that uses the Bluetooth on the mobile to make an actual connection to the beacon and pushes the data in to the beacon device

Because this is standard data that is programmed as a byte sequence, you can also parse this beacon data with a corresponding byte parser (see serial connections for examples) and assign property values with it.

## 7.4. NFC

NFC (Near Field Communication) tags are inexpensive radio tags that work with magnetic induction at short range (usually a couple of centimeters). NFC tags have no battery, the power is generated through magnetic induction through the antenna. You find NFC nowadays almost everywhere from keycards, stickers on items at stores, and credit cards.

NFC tags can be programmed to send a certain byte sequence when contacted. Asema IoT running on a PC with an NFC card reader or running in a mobile device equipped with an NFC chip, can be used to request this data from the card. As a byte sequence, it needs a byte parser to assign the values in the sequence to object properties.

## 7.5. Serial port data

### 7.5.1. Writing a parser schema for serial data

Data from serial ports is received as arrays of bytes i.e. characters. The purpose of the parser is to extract from this array of bytes those byts that are relevant, convert them to the proper format and then store to properties.

A parser definition for serial data therefore looks for example like this

```
{
    "type": "byte",
    "mode": "readonly",
    "cutout": {
         "length": 12
    },
    "preamble": [ 128, 0, 128 ],
    "single": {
        "attrs": [
            {
                "datatype": "float",
                "len": 4,
                "property": "color_red",
                "start": 0
            },
            {
                "datatype": "float",
                "len": 4,
                "property": "color_green",
                "start": 4
            },
            {
                "datatype": "float",
                "len": 4,
                "property": "color_blue",
                "start": 8
            }
        ]
    }
}
```

Let's interpret that. First, parser type is set to "byte" as we will be parsing incoming data byte by byte. Data is extracted as single values to three properties: color_red, color_green and color_blue (this could be an output from an RGB lamp for instance). Each value is a floating point number represented by four consecutive bytes (i.e. len = 4). The first value starts from byte position 0, the second from byte position 4, and the final one from byte position 8.

The serial data protocol can operate in three different `modes`: read only, write only or read/write. This mode is determined in the "mode" setting, which can take values `readonly`, `writeonly` and `read-write`. The default is read only. In this mode the serial interface will only try to parse the data that comes in. Write only is naturally the opposite, no reading is performed. In read/write mode the driver will attempt to work syncronously with the same packet structure. So if the schema defines a message of five bytes, the driver will first read five bytes and then write a package of five bytes back. The point where the writing is done is based on the syncronization of packets from a connecting device.

The trick with serial data is that it is received as a stream of arrays of bytes of undefined length, and not in neat packages. If dat transfer is really slow, you may be able to read the data in distinct packages but more often than not this is not the case. To deal with the situation, you somehow need to tell the system which bytes of a long stream belog to which package (or "frame" as it is called in serial data). This is the process of syncronizing the stream.

The most common means of syncronizing serial data is with preambles (or "syncwords"). These are sequences of bytes that start the frame. Asema IoT will read the incoming data and wait until a sync word is found. The rest is then assumed to be the frame. As more frames arrive, the sync word is tested on them to ensure that the data remains in sync. In case of disconnects, this will resync the flow eventually once a reconnect happens.

The preamble is defined in the schema with a list of values, each representing one byte, So for example in the example above, the message is assumed to start with bytes 128, 0, 128 in decimal (or 0x800080 in hex). Note that because the rest of the frame is parsed as a full frame, bytes equaling the preable bytes do not need to be escaped in the bytestream that follows.

A second option (or an additional measure) is to sync by the end of the message, as opposed to the beginning. This is harder to synchronize in a fast stream but does work in protocols that use some kind of end of line marker for instance. determines the byte positions is of course where a message starts and where it ends. Frame ends are defined with an additional parameter called `cutout`. Three types of cutouts can exist: `length`, `delimiter` and `bytes`.

A length cutout means that we are essentially working on fixed length arrays. In the example the length is 12 i.e. the parser assumes that a message has exactly 12 bytes and 13th received byte is in position 0 of the next message.

The delimiter means that there is some end of line character after each message. In many serial protocols this is a newline, carriage return, or both. delimiters are given by convention as characters. In the example case the schema would say `"delimiter": "\n"` for a newline delimiter, `"delimiter": "\r"` for a carriage return delimiter and `"delimiter": "\r\n"` for a combination of both.

Finally, the cutout bytes are similar to the preamble. They are the bytes appended at the end of each message. Effectively they are the same as a delimiter but defined in number values instead of characters.

Note that there is a shortcut to define the end of frame bytes with a field called `endofframe`. So the two following schema snippets have an equivalent end result

```
ALTERNATIVE 1:
{
    "type": "byte",
    "mode": "readwrite",
    "preamble": [ 128, 0, 128 ],
    "cutout": {
         "bytes": [ 1, 0, 16 ]
    },
    "single": {
       ...
    }
}

ALTERNATIVE 2:
{
    "type": "byte",
    "mode": "readwrite",
    "preamble": [ 128, 0, 128 ],
    "endofframe": [ 1, 0, 16 ],
    "single": {
       ...
    }
}
```

> **Important**
> Note that if you set the `mode`into `readwrite`or `write`, you need to define the cutout bytes (either as cutout or endofframe). This is the only way the system knows how to end a message frame it writes.

## 7.6. ModBus

### 7.6.1. Writing a parser schema for ModBus

Asema IoT Central can act as either a client or a server in ModBus communication, though not both simultaneously. Not surprisingly, the schema for ModBus contains this as a primary setting.

Once the ModBus driver knows in which mode it needs to operate, the schema will tell is which kind of ModBus data it will receive or fetch. Below is an example for a client

```
{
    "mode": "client",
    "poll": 2500,
    "type": "byte",
    "single": {
        "attrs": [
            {
                "datatype": "integer",
                "coils": {
                     "address": 16,
                     "length": 8
                },
                "property": "color_intensity"
            },
            {
                "datatype": "float",
                "inputs": {
                     "address": 3200,
                     "length": 16
                },
                "property": "color_red"
            },
            {
                "datatype": "float",
                "readregisters": {
                     "address": 40000,
                     "length": 1
                },
                "property": "color_green"
            },
            {
                "datatype": "float",
                "inputregisters": {
                     "address": 80000,
                     "length": 2
                },
                "property": "color_blue"
            }
        ]
    }
}
```

First, `mode` sets the interface as client. Because in ModBus clients fetch the values from the server, the next parameter `poll` tells the polling interval in milliseconds.

The parser then follows the structure familiar from other parsers, meaning that the parser is to find single values for a list of attributes. How the attributes are defined is what differentiates ModBus from other definitions. In a ModBus device data can be in

· Coils (bits). ModBus code 0x01

· Discrete inputs (bits). ModBus code 0x02

· Read holding registers (words). ModBus code 0x03

· Input registers registers (words). ModBus code 0x04

These locations are specified in the schema with the keywords `coils, inputs, readregisters, inputregisters`, respectively. After the keyword in the schema specifies the address of the data and the length of read in the appropriate units.

So the example above would read every 2.5 seconds

· 8 bits (one byte) from coils and convert that to an integer (MSB) into property color_intensity.

· 16 bits (two bytes) from discrete inputs and convert it to float into property color_red

· 1 word (two bytes) from read registers and convert to float into property color_green

· 2 words (four bytes) from input registers and convert to float into property color_blue

# 8. How to define data parsers

IoT Central relies on parser schema to interpret received data from external resource. Example parser schemas are listed in Section 3.1, “Writing a feed schema”, 4.2.3, “Parsers” and Chapter 8.2, “Examples of parsing various data formats”. This chapter will give an overview of parser schema, explaining its structure and some of its key attributes.

Parser schema is written in JSON format. Following is the simplest parser schema.

```
{
    "type": "json",
    "category": "current",
    "single": {
        "in": "",
        "attrs": [
            {
                "source": "value",
                "property": "system_cpu_load",
                "datatype": "integer"
            }
        ]
    }
}
```

In this schema,

- `"type"`— specifies the format of input data. Acceptable values include `"json"`, `"xml"`, `"csv"`, `"byte"`.

- `"category"`— signals whether input data can be treated as time-series values. For time-series data, category value has to be `"history"`or `"forecast"`. For non-timeseries data, category value should be `"current"`. Note `"category"`attribute is optional. If this attribute is absent, then by default data are considered as `"current"`category.

- `"single"`— this attribute serves two purposes. First, its name is Data Capture Mode, which indicates how IoT Central will capture input value if the HTTP message received from external resource contains multiple input. For instance, in case of JSON format input, the message may contain an array of JSON objects instead of just one object. There are four types of Data Capture Mode: (1) `"single"`— only the first input will be saved, the rest is ignored. (2) `"first"`— same as `"single"`mode. (3) `"each"`— all the input will be saved. (4) `"nth"`— only keeps the `n`th input.

  If the received HTTP message contains only one input, e.g., just one JSON object in the case of JSON input, then you can use any aforementioned mode as the name of this attribute, result will be the same.

  Second, the value of this attribute is a JSON object, which encapsulates attributes `"in"` and `"attrs"`. Besides, one additional attribute `"at"` will be included in this object if Data Capture Mode is `"nth"`. The value of `"at"` is an integer starting from 1, tells what the `n` is. For example usage of `"at"` attribute, see Section 4.2.3, “Parsers”.

- `"in"`— this attribute is optional. It is needed when input data are `"json"`or `"xml"`format, and the actual values are enclosed inside several layers of structures.

  For example, the schema in Section 3.1, “Writing a feed schema” has `"in":"computerData|measurementData"` . This indicates the input XML data has a hierarchical structure like this:

```
<computerData>
  <measurementData>
  </measurementData>
</computerData>
```

and from section `measurementData` the actual values can be located. Another example, the schema in Section 8.2.1, “Parsing JSON” has `"in":"weatherData|weather|loc|fc"` . This means the input JSON data has a hierarchical structure like this:

```
{
    "weatherData": {
     "weather": {
      "loc": {
       "fc": {
        ...
       }
      }
     }
    }
}
```

and inside object `"fc"`, the actual values can be located and extracted.

- `"attrs"`— its value is an array of JSON objects. Each object contains detailed information about how to extract and intepret each value from raw input data. It has at least three attributes `"source"`, `"property"`and `"datatype"`.

  - `"source"`— indicates the location from where the actual data can be extracted. Its value depends on input format.

    For XML input, the value of `"source"` is XML tag, inside this XML tag contains the actual input value that needs to be extracted. For example, the schema at Section 3.1, “Writing a feed schema” has `"source":"system_load"` and `"source":"time"`, so the actual input value are in tags `system_load` and `time` (these two tags are inside `measurementData` as indicated by `"in"` attribute).

    For JSON input, the value of `"source"` is key string, the value of this key is the actual input data that needs to be extracted. In the sample schema at the beginning of this chapter, the `"source":"value"` indicates the actual input value is the value of key "value". For more example, see the schema at Section 8.2.1, “Parsing JSON”.

    For CSV input, the value of `"source"` is an integer number (>=1). It indicates which column contains the actual input data. For its example usage, see Section 4.2.3, “Parsers”.

    For raw byte input, there is no `"source"` attribute.

  - `"property"`— after extracting value from raw input data, IoTC need to give a name to this value for presentation purpose. This name is provided by `"property"`attribute. In the sample schema at the beginning of this chapter, input value will be named as "system_cpu_load".

  - `"datatype"`— specifies the data type for the extracted value. Acceptable values include float, double, integer, string, bool, datetime, etc.

## 8.1. Parsing various data formats

### 8.1.1. Parsing JSON

### 8.1.2. Parsing XML

### 8.1.3. Parsing CSV

### 8.1.4. Parsing bytestreams

## 8.2. Examples of parsing various data formats

### 8.2.1. Parsing JSON

```
{
 "type": "json",
 "each": {
  "in": "weatherData|weather|loc|fc",
  "attrs": [
    {
     "source": "pp",
     "property": "precipitation_prob_percent",
     "datatype": "integer",
     "conversion": "none"
    },
    {
     "source": "pr",
     "property": "precipitation_mm",
     "datatype": "integer",
     "conversion": "none"
    },
    {
     "source": "c",
     "property": "cloudiness_percent",
     "datatype": "integer",
     "conversion": "none"
    },
    {
     "source": "t",
     "property": "temperature_celsius",
     "datatype": "integer",
     "conversion": "none"
    },
    {
     "source": "rh",
     "property": "relative_humidity_percent",
     "datatype": "integer",
     "conversion": "none"
    },
    {
     "source": "wn",
     "property": "wind_direction_enum",
     "datatype": "integer",
     "conversion": "none"
    },
    {
     "source": "ws",
     "property": "wind_speed_ms",
     "datatype": "integer",
     "conversion": "none"
    },
             {
     "source": "dt",
     "property": "timestamp",
     "datatype": "datetime",
     "conversion": "yyyy-MM-dd HH:mm"
    }
  ]
 }
}
```

### 8.2.2. Parsing XML

```
{
 "type": "xml",
 "each": {
  "in": "weatherData|weather|loc|fc",
  "attrs": [
    {
     "source": "pp",
     "property": "precipitation_prob_percent",
     "datatype": "integer",
     "conversion": "none"
    },
             {
     "source": "dt",
     "property": "timestamp",
     "datatype": "datetime",
     "conversion": "yyyy-MM-dd HH:mm"
    }
  ]
 }
}
```

### 8.2.3. Parsing CSV

Consider the following CSV which presents temperature, humidity and pressure readings for five days at noon each day:

```
     2015-06-06 12:00;28.7;56.2;1201
     2015-06-07 12:00;31.1;42.7;1256
     2015-06-08 12:00;32.0;41.7;1240
     2015-06-09 12:00;32.1;55.8;1233
     2015-06-10 12:00;29.6;55.2;1259
```

```
{
    "type": "csv",
    "direction": "column",
    "first": {
        "attrs": [
            {
                "source": 2,
                "property": "temperature",
                "datatype": "float",
                "conversion": "none"
            },
            {
                "source": 3,
                "property": "humidity",
                "datatype": "float",
                "conversion": "none"
            }
        ]
    }
}
```

```
{
 "type": "csv",
 "direction": "column",
 "each": {
  "attrs": [
   {
    "source": 2,
    "property": "temperature",
    "datatype": "float",
    "conversion": "none"
   },
   {
    "source": 3,
    "property": "humidity",
    "datatype": "float",
    "conversion": "none"
   }
  ]
 }
}
```

```
{
    "type": "csv",
    "direction": "row",
    "nth": {
        "at": 2,
        "attrs": [
            {
                "source": 1,
                "property": "periodic_forward_active_energy",
                "datatype": "float",
                "conversion": "none"
            },
            {
                "source": 3,
                "property": "timestamp",
                "datatype": "float",
                "conversion": "none"
            }
        ]
    }
}
```

# 9. Debugging and troubleshooting