MSI contributions for S-X-AIPI

MSI is contributing with the development of:

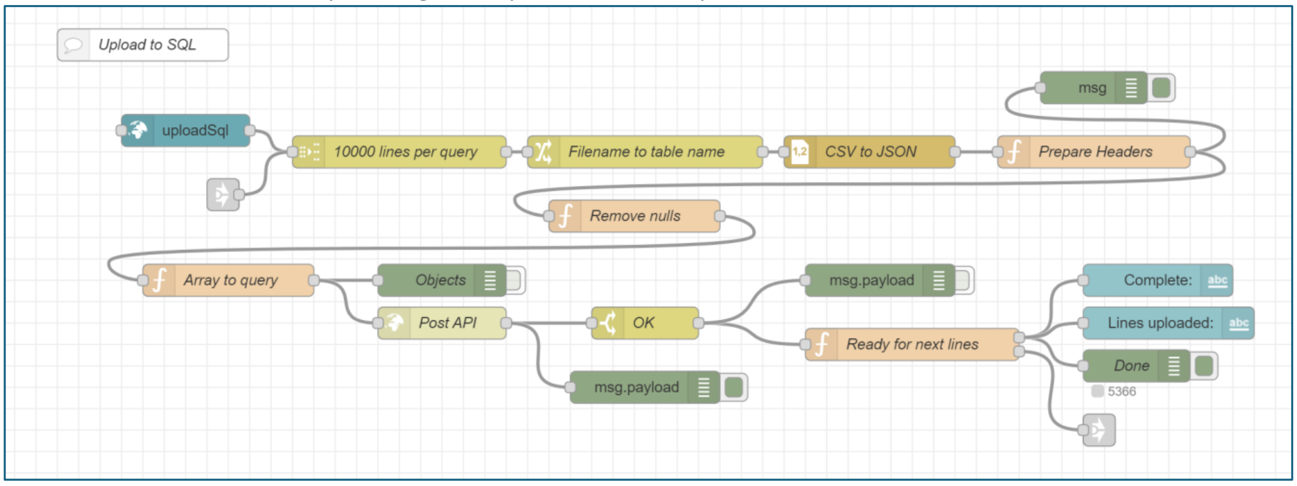
- A node-red based data flow.
- A standard API interface definition which allows to access in a simple way to the heterogenous data structures obtained from the multiple process data sources.
- A database infrastructure for time-series data storage and consumption.

The goal is to get data from a highly heterogeneous set of sources and transform them into readible and accesible data sets for the S-X-AIPI modules to be used. It covers three type of data structures being (i) Real Time data defined as *Signals*, (ii) historical time series defined as *Measurements* and (iii) structured data structures defined as *Records* (SQL).

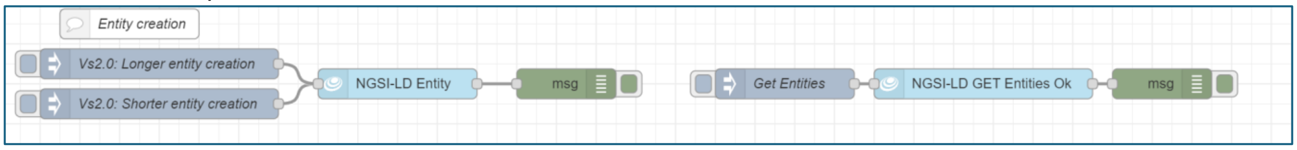
1. Node-Red flow

The node red flows executes these tasks:

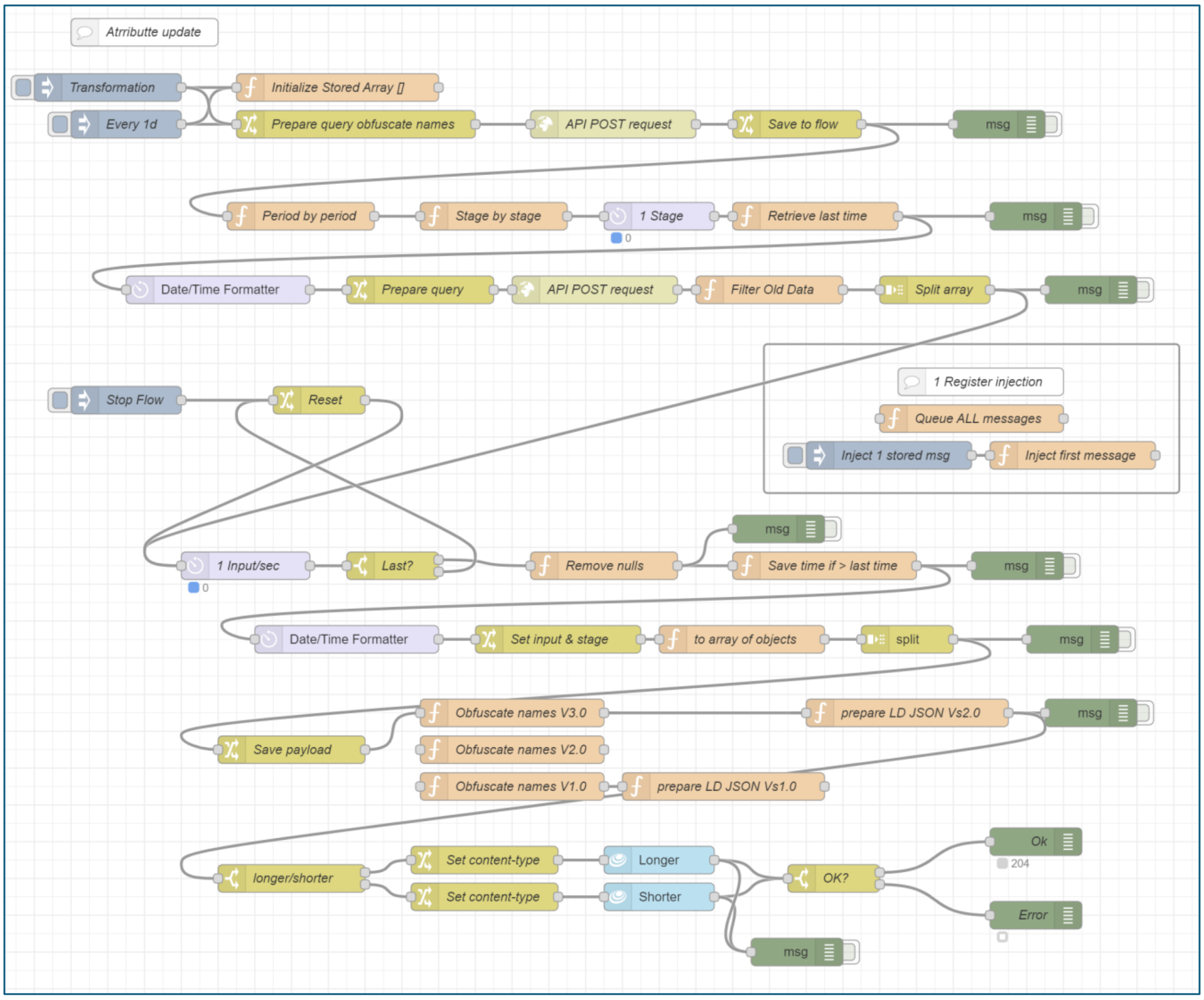
1- The Sidenor Data Uploading Flow: processes and uploads the new data from Sidenor.



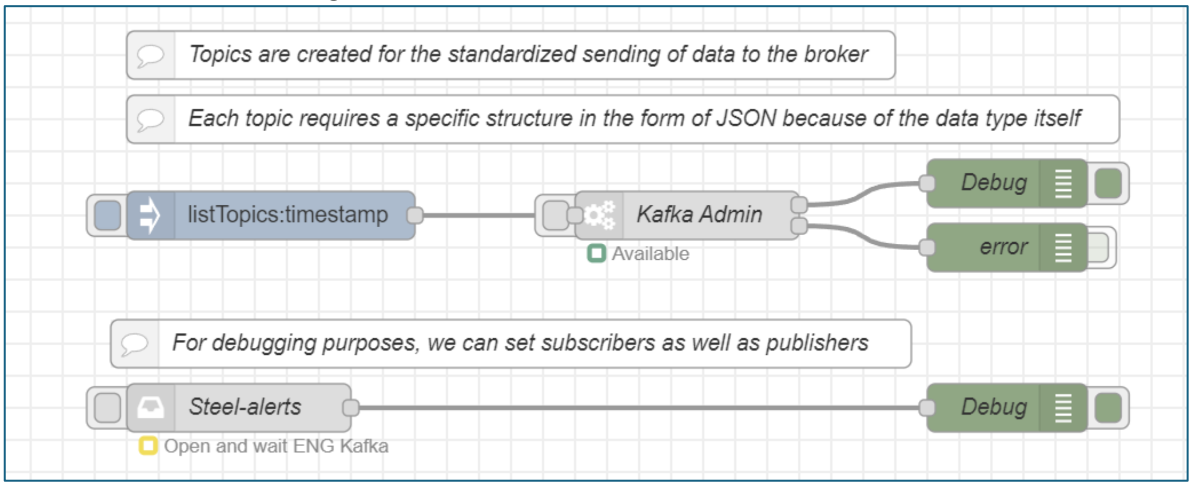
2- The Entity Creation Flow: creates the entities in the Orion Context Broker.



3- The Orion Attributte Updating flow: process the data it, cleaning and formatting it and uploads it to the Orion Context Broker.



4- The Kafka Flow: is subscribed to the Orion Context Broker to capture the information from the Autonomous Manager.


Topics are created for the standardized sending of data to the broker

Each topic requires a specific structure in the form of JSON because of the data type itself

For debugging purposes, we can set subscribers as well as publishers

2. API Rest Interface

To facilitate access to data and accelerate different module integrations, two services have been created for each data type: one to know the parameters included on the available datasets and another one to access the values of these parameters.

Authorization

All the following methods are secured against different types of cyberattacks and unauthorized access so, to make use of them, the data consumer must know the endpoint URL and must have a valid access token. All API calls must contain in the header an access-token with the value of the TOKEN that identifies the receiver as an authorized user.

Signals (Real Time data)

Signals are the process parameters. They are defined in the database as master data.
By accessing **https: // [ENDPOINT URL] / api / signals (GET)** the consumer will be able to know which signals has access to and with what permission (R-Read, W-Write or RW-Read write). This call will return a list in JSON format, in which each element will correspond to the definition of a signal.
Example structure:

```
[
        {
                "name": "Descriptive name of the signal",
                "field": "Technical name of the signal",
                "unit": "Physical unit of measurement",
                "access": "R-Read, W-Write or RW-Read write",
                "type": "Data type: String, Integer, Decimal, Boolean, ..."
        }
]
```

By accessing **https: // [Endpoint URL] / api / signals-get (POST)** the consumer will be able to access to a value of one signal. The call to the method must be done sending the technical name of the signal from which the consumer wants to obtain its value in a JSON format at the body of the call.
Example structure:

```
[
        "Technical name of the signal"
]
```

Example structure:

```
[
        "Actual value of the signal"
]
```

To change an actual value of a signal (already with RW-Read Write permissions), the consumer must post the value next to the technical name to the endpoint **https: // [ENDPOINT URL] / api / signals-set (POST)**
Example structure:

```
[
        {
                "field":"Technical name of the signal",
                "value":"Actual value of the signal"
        }
]
```

Measurements (Time Series Data)

Measurement is understood as a group of signals which contains values of each of the signal over time.

In addition to being able to access raw historical data, the system allows access to preprocessed data using different aggregation functions over time periods such as:

- Mean
- Maximum
- Minimum
- Sum
- Standard deviation

To access to the historical raw or aggregated values, the consumer will have to know to which measurements they belong.

By accessing **https: // [ENDPOINT URL] / api / measurement (GET)** the consumer will be able to know which measurements and which signals has access to and with what permission. This call will return a list in JSON format, in which each element at top level will correspond to the definition of a measurement, and below each measurement the signals that are included in it.
Example structure:

```
[
        {
                "name": " Name of the measurement ",
                "description": "Description of the measurement",
                "access": "R-Read, W-Write or RW-Read write",
                "fields": [
                        {
                        "name": "Descriptive name of the signal",
                        "field": "Technical name of the signal",
                        "unit": "Physical unit of measurement",
                        "type": "Data type: String, Integer, Decimal, Boolean, ..."
                        }
                ]
        }
]
```

Obtaining historical values of the signals is done by requesting the measurement that contains the desired signal and by indicating some parameters as defined hereunder.
Accessing **https: // [ENDPOINT URL] / api / measurements-get (POST)** the consumer will have to provide in the body an object (in JSON format) with the following options that will apply to the requested dataset:

- Aggregate function
- Conditions
- Grouping for aggregate function
- Order
- Limit
- Offset

```
{
        "measurement": "Name of the requested measurement to obtain data from",
        // Name of the fields of those measurements to obtain data from
        "fields": [
                {
                        "field": "Technical name of the signal",
                        "function": "Data aggregation function to be applied",
                        //The following aggregation functions can be applied: COUNT (), //DISTINCT (),
                        MEAN (), SUM (), FIRST (), LAST (), MAX (), MIN ()
```

```
                    "as": "New name (ALIAS) of the field after applying the aggregate function"
                }
        ],
// Filters to apply on the select of the data
        "where": {
                // Limit result by dates
                "condition": "AND",
                "fields": [
                    // Measurements after a date (greater than)
                    {
                    "condition": ">",
                    "fields": [
                    "time",
                    "'2020-03-13T19:00:00Z'"
                ]
        },
        // Measurements before a date (less than)
                {
                "condition": "<",
                "fields": [
                    "time",
                    "'2020-03-13T21:00:00Z'"
                ]
                }
        ]
        },
        // Data aggregation by time: d (Days), h (Hours), m (Minutes) or s (Seconds)
        "groupBy": "time(10m)",
        // Signals values sorted descending (DESC) or ascending (ASC)
        "order": "DESC",
        // Limit the number of results
        "limit": 10,
        // Offset to define from which record to start obtaining the results. This is very useful for paging
        systems
        "offset": 0
}
```

This call will return a list in JSON format, in which each element will correspond to the value of the signals in an instant of time.

Example structure:

```
[
        {
                "time": "Time stamp of the measurement",
                "field": "Value of signal 1 (or Alias)",
                "field": "Value of signal 2 (or Alias)",
                // …
        },
        {
                // …
        }
]
```

To set a new values of fields of a measurement (already with RW-Read Write permissions), the consumer must post a structured JSON to **https: // [ENDPOINT URL] / api / measurements-set (POST)** . The consumer will be able to specify the fields and tags. The consumer can add more than one value for each field, creating an array of objects with a specific timestamp for each object of the array. If no timestamp is provided, the service will store the data next to the actual timestamp.

Example structure:

```
{
        "measurement": "Name of the measurement ",
        "fields":["Technical name of the signal 1","Technical name of the signal 2"],
        "tags":["Technical name of the tag 1"],
        "values":[
                {
                "time": 1607105101682,
                "fields": ["Value of signal 1","Value of signal 1"],
                "tags":["Value of tag 1"]
                },
                {
                "fields": ["Value of signal 1","Value of signal 1"],
                "tags":["Value of tag 1"]
                }
        ]
}
```

Records (SQL Data)

A record is a line in a table in the SQL database.
By accessing **https: // [ENDPOINT URL] / api / records (GET)** the consumer will be able to know which tables, and which fields has access to and with what permission. This call will return a list in JSON format, in which each element will correspond to the definition of a table. For each table the consumer will be able to obtain the fields it contains.

Example structure:
```
[
	{
	"name": "Name of table",
	"description": "Description of table",
	"access": "Access type: R-Read, W-Write or RW-Read write",
	"fields": [
			{
			"name": "Name of field",
			"description": "Description of field",
			"type": "Data type: String, Integer, Decimal, Boolean, ... ",
			"required": true, // Indicate if it is a required field
			"key": true // Indicate if it is a key field
			},
			{
			//...
			}
		]
	}
]
```

As with historical time series data, SQL data can be requested from the service in an aggregated form using different functions.
Accessing **https: // [ENDPOINT URL] / api / records-get (POST)** the consumer will have to provide in the body an object (in JSON format) with the following options that will apply to the requested dataset:

- Aggregate function
- Conditions
- Grouping for aggregate function
- Order
- Limit
- Offset

Example structure:
```
{
	"table": "Name of the table to obtain data from",
	"fields": [
		{
			"name": "Name of field",
			"function": " Function to be applied to get only one value from the group",
			"as": "New name (ALIAS) of the field after applying the aggregate function"
		},
		{}
	],
	"where": {
		"condition": "AND",
		"fields": [
			{
				"condition": ">",
				"fields": [
					"CAMP_1",
					1
				]
			},
			{
```

```
                        "condition": "LIKE",
                        "fields": [
                                "CAMP_2",
                                "%A%"
                        ]
                }
        ]
},
"groupBy": [
        "CAMP_1",
        "CAMP_2"
],
"order": [
        {
                "field": "CAMP_2",
                "direction": "DESC"
        }
],
"limit": 5,
"offset": 0
}
```

This call will return a list in JSON format, in which each element will correspond to the values of fields in one record.
Example structure:

```
[
        {
                "field": "Value of field 1 (or Alias)",
                "field": "Value of field 2 (or Alias)"
                //...
        },
        {
                //...
        }
]
```

To write record values to a structured table, the consumer must post a structured JSON to the endpoint to **https: // [ENDPOINT URL] / api / records-set (POST)** specifying the fields and the values of the record and the table to write to (already with RW-Read Write permissions).
Example structure:

```
{
        "table": "Name of table",
        "fields":["Name of field 1","Name of field 2","Name of field 3","Name of field 4"],
        "values":[
                [
                        "Value of field 1","Value of field 2","Value of field 3","Value of field 4"
                ]
        ]
}
```