Cordis Machine Control Dashboard

Design Document

Author: Patrick Molenaar
Date: 24 April 2020
Version: 0.1 Draft

Table of Contents

1 Introduction....3
1.1 Document references....3
1.2 Document History....3
2 Structure....4
3 The backend....5
3.1 The configuration....5
3.2 Connection....6
3.2.1 Failures in communication with MCS....6
3.3 Data flow....6
3.4 Data handling and storage....7
4 Backend design....8
5 The frontend....9
5.1 Notification of the frontend....9
6 Database model....10
7 XML Configuration file....13
7.1 XML validation....14
8 JSON interface template....16
8.1 Request payload....16
8.2 Reply payload....16

Illustration Index

Figure 1: Global overview of architecture....4
Figure 2: Backend startup sequence diagram....5
Figure 3: Data flow sequence diagram....7
Figure 4: Backend service class diagram....8
Figure 5: Database model diagram....11

Index of Tables

Table 1: Document references....3
Table 2: Document history....3
Table 3: Database table ValueType....11
Table 4: Database table PartVariable....11
Table 5: Database table PartValue....12

1 Introduction

This document describes the design for the Cordis Machine Control Dashboard (MCD). MCD is an application that shows information from the Cordis Machine Control Server (MCS) to the user.

1.1 Document references

For the creation of this document the following documents were used.

Reference	Title	Doc. version
MCS_PROT	MCS Protocol Cordis SUITE	0.4 Draft

Table 1: Document references

1.2 Document History

Version	Author	Date	Change
0.1 Draft	Patrick Molenaar	24-04-2020	Initial version

Table 2: Document history

2 Structure

The MCS is a server running at Cordis and can be queried to get the information needed to show on the dashboard. The information that can be queried is the values from machine parts in the fabrication process. These machine parts are connected to a PLC (Programmable Logic Controller) and the MCS can read the values from the PLC and send them to the MCD.
This querying is done using HTTP with a JSON payload as that is the protocol specified by Cordis for getting the information from the MCS.

Figure 1 shows a high level overview of the architecture. In this document the design decisions are explained that lead to the shown architecture.

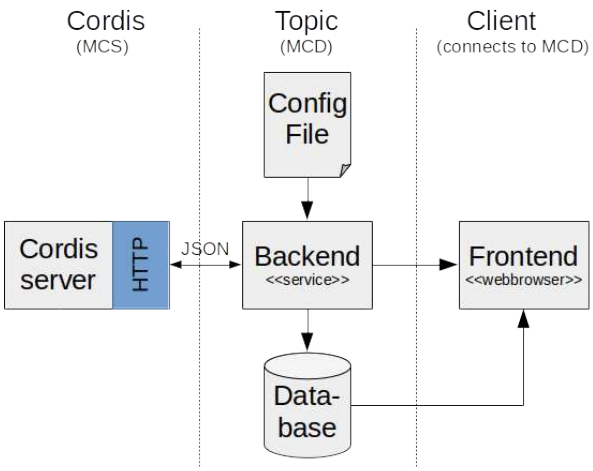

Figure 1: Global overview of architecture

The MCD can be split into several components.

1. The backend
2. The storage
3. The frontend

The Cordis server (MCS) is not part of the MCD, as this is a separate server. The MCD does however communicate with the MCS to get its information.

The backend is the client of the MCS that queries the MCS for information and stores the information in the storage. The frontend is a separate component that connects to the backend and displays the information. The frontend does not necessarily have to be running on the same machine as the backend, but can be a different machine running a web browser.

3 The backend

The backend is a service which is running constantly. On startup of this service, a configuration will be loaded. This configuration determines to which MCS to connect and which information to get from the MCS with what interval.

When the backend is started, it loads the configuration. The configuration file parser reads the configuration file from disk, validates it and processes it. The processing of the configuration file means extracting the MCS connection details and the information details (the machine parts that need to be read at which interval) from the configuration file. Extracted information is then passed to the backend.

For each machine part an entry in the database is created (if not already present) and a timer is started. Then a connection with the MCS is set up. The backend is now ready for requesting information from the MCS. See chapter 6 for more information on the database model.

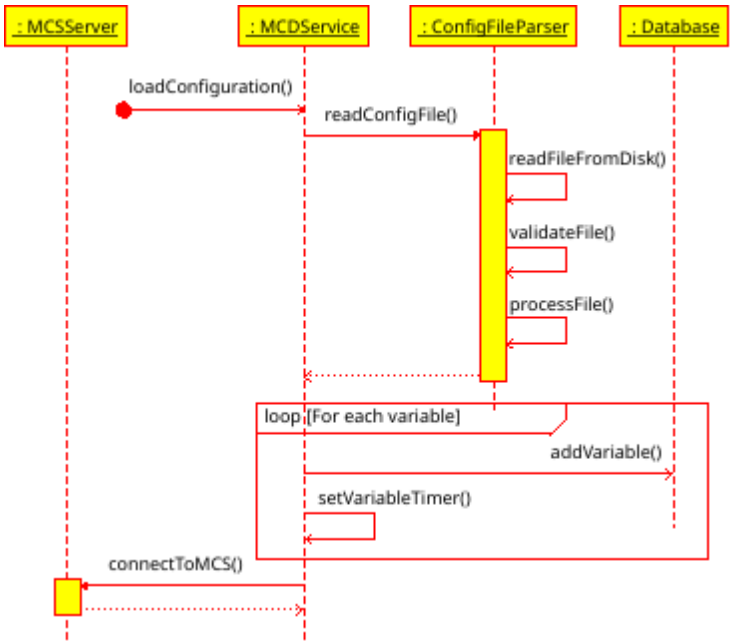

Figure 2: Backend startup sequence diagram

To make the backend flexible for other configurations a message can be sent to the backend to reload the configuration. This is to prevent the backend from a restart each time a new configuration needs to be loaded.

When the configuration need to be reloaded, all variable timers must be stopped and the connection to the MCS must be closed. Only then the new configuration can be loaded. For more information about the configuration file syntax chapter 7.

3.1 The configuration

The backend should be configurable in such a way that the information to retrieve and the query interval can be specified as well to which MCS to connect.

There are multiple options for this configuration.

1. Configuration file
2. Database
3. Startup parameters

Startup parameters are inefficient as it requires the backend to restart each time a new configuration needs to be loaded. For a user to be able to modify the configuration a frontend must be created as manually altering the parameters in the command line requires advanced knowledge of the system. And the backend might be installed in a location where a user can't access it.

A database is an option, but is less flexible as it requires a more complex frontend to load a new configuration. This frontend gives the user the capability to add, remove or change the configuration. However, this is overkill for this application.

A configuration file seems the best option. It can be loaded easily, altered offline and loaded again. For loading a remote call to the backend can be sufficient. A frontend for adding, removing or changing the configuration file can be made, but is not necessary for the backend to function properly and be dynamic with loading the configuration. XML format is easy to understand for users to be able to edit the configuration file and is widely supported in various programming languages. Including validation of the content.

3.2 Connection

The backend makes a connection to the MCS using HTTP. It sends requests for information at given intervals, as defined in the configuration file. These requests have a JSON payload as specified by Cordis. This is the protocol that the MCS understands. The MCS will reply with a JSON message containing the requested information. For more information about the configuration file syntax chapter 7.

3.2.1 Failures in communication with MCS

- *How will the MCS reply in case of an incorrect or not understood request?*

For now assume that the MCS will not respond to incorrect requests.

3.3 Data flow

When the information timer elapses, the backend requests the information for the machine part for which the timer elapsed from the MCS. When this information is received, this information is stored in the database. Afterwards a notification is sent to all connected clients to notify them which machine part is updated.

For a frontend to be able to receive notifications, they first must register themselves at the MCD service. On registration a list of connected clients is kept in the backend. Once registered, the service is aware of the presence of the frontend and will be able to sent notifications when new information from the MCS is received.

It is the responsibility of each client to retrieve the latest information of that given machine part and display that information to the user.

In case a connection exception from the service to the client occurs, the client will be unregistered (removed from the list of connected clients), to prevent further errors in communication. There is also the possibility for a client to unregister itself. In that case the client is also removed from the list of connected clients at the backend.

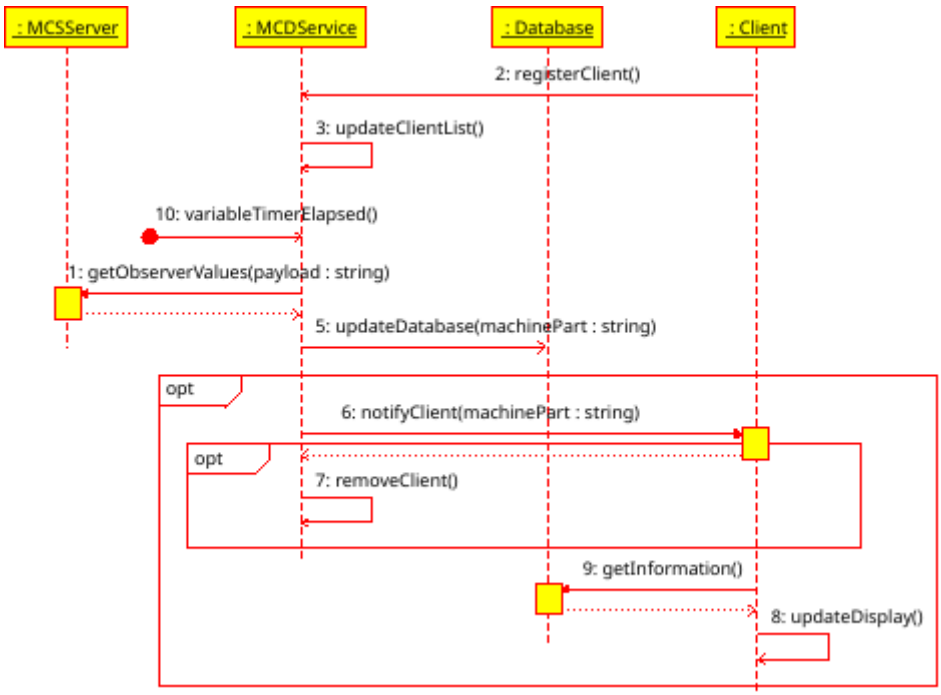

Figure 3: Data flow sequence diagram

3.4 Data handling and storage

The backend receives the information in JSON format. It will extract this information and store it on a generic storage location. Several options for storage are available.

1. In memory
2. In a file
3. In a database

Storage in memory is very fast but also very volatile. As soon as the backend stops, all information is gone. This is unwanted behavior. When storing the information in memory, custom code in the backend is required for the frontend to access this information.

Storage in a file is persistent, but difficult to access. Also, the custom format of the file limits the possibilities of standard tools to be able to access the information in the storage. When storing the information in a file, custom code in the backend is required for the frontend to access this information. Finding information in a file is much harder than finding information in a database.

Storage in a database seems the best option. The data is persistent, and easy accessible. Databases are also pretty fast as they are optimized for handling large amounts of information. Most databases have a standardized interface (SQL) through which the information can be accessed by standard tooling (e.g. Grafana). For more information on the database model see chapter 6.

4 Backend design

Here follows an initial design of the backend.

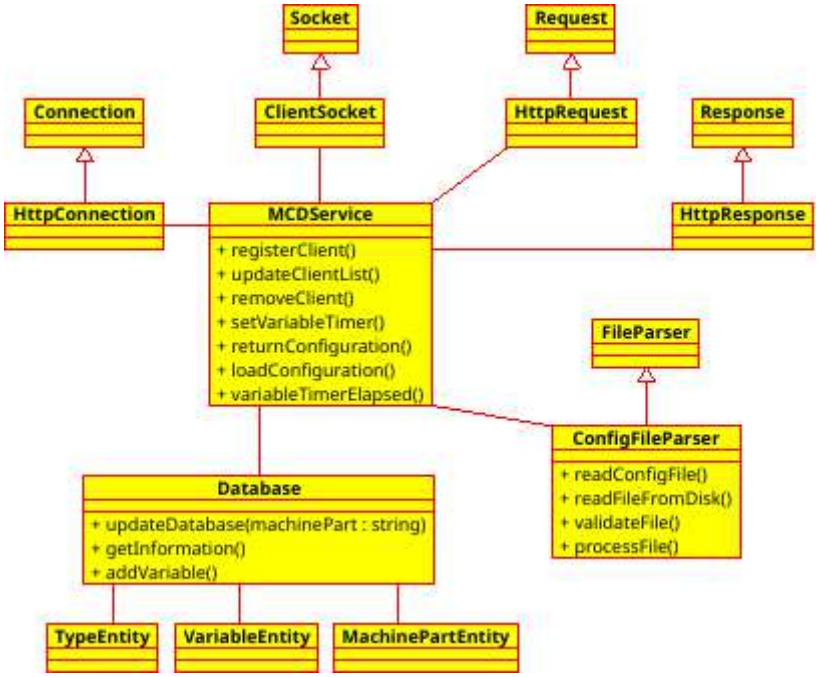

Figure 4: Backend service class diagram

MCDService is the main application. On startup of this application this class creates the ConfigFileParser to load and process in the configuration file.

After loading the configuration file, it creates the Database class which is an interface to the database. With the help of the Database class the variables and values can be written to the database.

For connection to the MCS the MCDService creates the HttpConnection class. This class takes care of the communication with the MCS. The HttpRequest class is used in communication with the MCS.

For interfacing between the backend and frontend the ClientSocket is used. The ClientSocket creates a listener for listening to the requests from the frontend. For notifying the frontend the HttpResponse class is used.

5 The frontend

The frontend can be an independent (web)application. The frontend is primarily used for viewing the information from the backend. In a later stage the frontend can also be used to control the backend. Controlling the backend means register and unregister a client (frontend) and commanding the backend to reload the configuration. The frontend design is out of scope of this document.

5.1 Notification of the frontend

The frontend can either get the information directly from the database or it can listen to update messages from the backend. As the backend knows exactly when new information is received, it has the ability to send an update message to all connected frontends (assuming there can be more than one frontend). This way the frontend will know when to update. For the backend to be able to send update messages to the frontend, the backend must know which frontends there are. This can be achieved with a registration mechanism. A frontend will then register itself at the backend, telling the backend where it can be reached and then listen for messages from the backend.

The messages from the backend to the frontend can be only a notification, something like a ping message, telling the frontend that new information is available, or it can be a more complex message passing the new information to the frontend.

In the first case where only a ping is sent, the frontend will not know what has changed and has to retrieve all information again to be up to date.

In this last case there will be two channels passing information to the frontend, the connection from the frontend to the storage and the messages from the backend. This makes the frontend more complex than necessary.

A hybrid solution is better. With this hybrid solution the backend tells the frontend which parameter has changed (but doesn't give its value). Then, the frontend can retrieve that specific information from the database, without the need to retrieve all information from the database again. This will make the frontend more efficient.

A last option is a completely independent frontend, where the frontend connects directly to the storage without interaction of the backend. In this option the frontend needs to poll the storage frequently for new information. This is less efficient, but more independent. This last option is also less flexible, as there is no interaction with the backend, so controlling the backend is not possible in this situation (e.g. sending a request to reload the configuration).

A form of an independent frontend is for instance Grafana. Grafana has an SQL plugin for various databases. This allows Grafana to connect to the storage and retrieve and display the information.

6 Database model

The database stores the information received from the MCS. This information can be accessed by the frontend for displaying. Each received value gets a time stamp so it can be displayed in chronological order.

- *How to identify variables?*

For now assume that the name is not unique, but the combination of observer name and machine part id and machine part name is. When querying for an observer name in combination with machine part (id and name), you will get all values for that particular variable.

When information is received from the MCS, the backend will read the information and convert it to fit in the database. The database is optimized to hold all information in such a way that quick querying is possible.

Some information that is received is redundant and will occur on every request for a certain machine part and never change. This information is put in a separate table which will act as a lookup table. This makes the expansion of the information easier and keeps the size of the database small. There is also some fixed information, namely the value type. This can only be a specific set of values. See table Type.

For storing the values there were several options.

1. Storing values of each machine part in a separate table.
2. Storing values of all machine parts in only one table with the value as a string representation.
3. Storing values and machine parts information in one single table (flat database model).
4. Storing values of all machine parts in only one table with the value as is.

Storing the values of each machine part in a separate table is not efficient. The information is separated, but this construction is not very flexible. When new variables are to be explored, new tables need to be created, which changes the integrity of the database.

Storing all values as a string will overcome the difficulty of how to store each type of variable. But this requires more processing from the frontend, as the frontend must convert each variable from string to its actual type. If you don't want this, then the only way is for the backend to convert the value, but that means than you can only get the values by requesting them from the backend. This makes the frontend very dependable on the backend.

Storing all values in a single table (flat database model), results in very much redundant information being stored in the database. This is not an optimal solution.

The best solution would be to separate the type, machine part and values in separate tables. Also create a separate field for each type of variable (Boolean, Integer, Sting etc.). Splitting the information over multiple tables reduces the information redundancy and makes the database flexible for storing new variables. Splitting the value by type in a separate field preserves that actual value in it original format and makes querying by the frontend easier. Another advantage of this is that in case of a Boolean value, there is the possibility to store the string representation of that value too. A Boolean value of 'true' can then have a string representation of 'on', or 'running' or something similar. This is a supported feature of the JSON reply message from the MCS.

Here a representation of the database tables. The bold fields represent the key or combined key of that table.

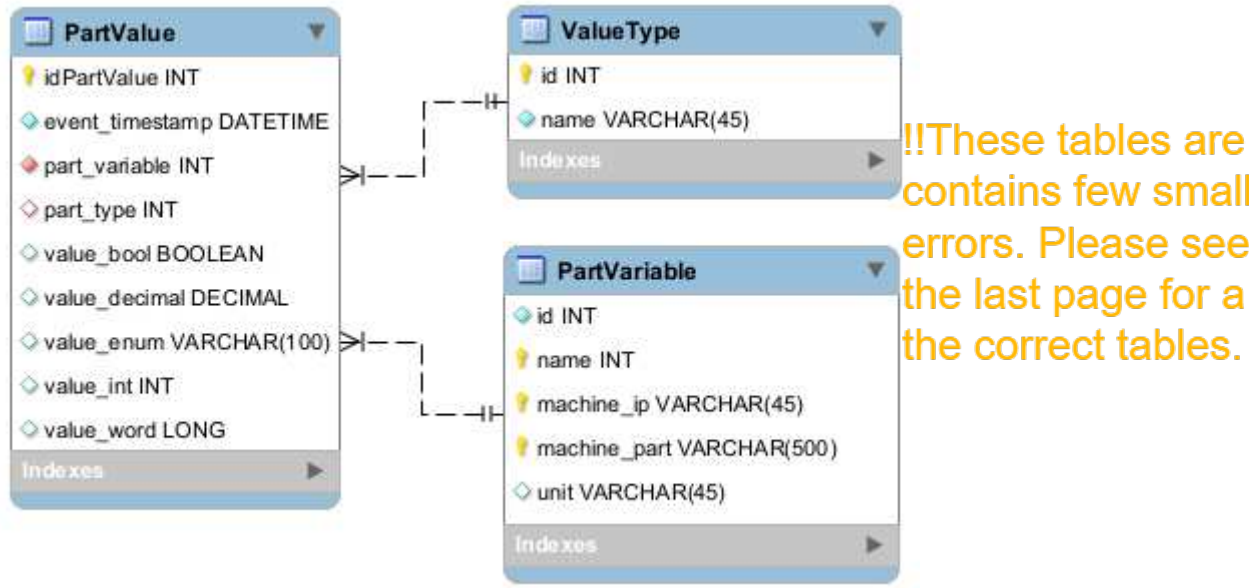

Figure 5: Database model diagram

Field	**Type**	**Description**
id	Integer	Auto generated ID
name	String	Fixed values, can be one of BOOLEAN_OBSERVER, DECIMAL_OBSERVER, ENUM_OBSERVER, INT_OBSERVER or WORD_OBSERVER

Table 3: Database table ValueType

Field	**Type**	**Description**
id	Integer	Auto generated ID
name	String	Name of the variable (combined key)
machine_ip	String	The IP of the machine (combined key)
machine_part	String	The name of the machine part (combined key)
unit	String	The unit of the variable

Table 4: Database table PartVariable

Field	Type	Description
id	Integer	Auto generated ID
event_timestamp	DateTime	Date and time when the event was received
part_variable	Integer	The ID of the part variable from table PartVariable (reference)
part_type	Integer	The ID of the type of the variable from table Type (reference)
value_bool	Boolean	Value of the variable (or null, depending on Type)
value_decimal	Float	Value of the variable (or null, depending on Type)
value_enum	String	Value of the variable (or null, depending on Type)
value_int	Integer	Value of the variable (or null, depending on Type)
value_word	Long	Value of the variable (or null, depending on Type)

Table 5: Database table PartValue

7 XML Configuration file

The configuration file must contain the information for the backend to know where to connect to and what information to retrieve from the MCS and with what interval.

Information to know where to find the MCS requires the IP address, port and base-path.

The connection block of the configuration file can look as follows.

```
<connection>
   <address>127.0.0.1</address>
   <port>80</port>
   <path>JsonServices/com.cordis.explorer.connectors.MachinePart.MachinePartConnector/</
path>
</connection>
```

Information about the variables require the controller name, machine part tree and optionally the observer name. Also, the poll interval should be specified. How often should this value be requested from the MCS. As there can be multiple variables for which the information need to be requested, this block of can occur multiple times in the configuration file. Once for each variable.

To be most flexible with the interval I suggest using the same syntax as is used for cron schedules (cron job scheduler in Linux). As this is a proven technology for scheduling.
The specification of the syntax can be found at https://crontab.guru/
A variable block of the configuration file can look as follows.

```
<data>
   <variable>
      <controllername>BeckhoffOPC</controllername>
      <machinename>Machine</machinename>
      <machinepart>ObserverTesterB</machinepart>
      <observername>Int16Var</observername>
      <schedule>5 * * * *</schedule>
   </variable>
</data>
```

For better readability can the schedule be refined. By splitting the sections of the schedule.

```
<schedule>
   <minute>5</minute>
   <hour>*</hour>
   <dayofmonth>*</dayofmonth>
   <month>*</month>
   <dayofweek>*</dayofweek>
</schedule>
```

Maybe this is also easier for parsing.

With the example information a request payload may look as follows. The request payload and reply payload have been created using document [MCS_PROT].

```
[
   {
      "SecurityToken":"ADMIN"
   },
   {
      "InputArgs":[
      {
```

```
        "criteria":{
           "machinePartRegEx":"BeckhoffOPC/Machine/ObserverTesterB",
           "observerNameRegEx":"Int16Var",
           "timeStampFrom":"",
           "timeStampTo":"",
           "rowLimit":0
        }
     }
     ]
   }
]
```

The MCS will then reply payload will look as follows.

```
{
   "type":"MULTI_OBJECT",
   "value":[
   {
      "name":"Int16Var",
      "unit":"",
      "type":"INT_OBSERVER",
      ""ObserverTesterB"":{
         "fullPath":"IP 192.168.10.1.1.1 | /TestProject/Main",
         "name":null
      },
      "currentValue":125,
      "trueText":null,
      "falseText":null
   }
   ]
}
```

- *Are these payloads correct?*

Have Cordis validate the request and reply payloads and correct them if necessary.

7.1 XML validation

Because the configuration file is in XML format, it can be validated using an XSD file. The validation file for the above XML (with split up schedule) will look as follows.

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="serviceconfig" type="serviceconfigType"/>
  <xs:complexType name="connectionType">
    <xs:sequence>
      <xs:element type="xs:string" name="address"/>
      <xs:element type="xs:short" name="port"/>
      <xs:element type="xs:string" name="path"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="scheduleType">
    <xs:sequence>
      <xs:element type="xs:string" name="minute"/>
      <xs:element type="xs:string" name="hour"/>
      <xs:element type="xs:string" name="dayofmonth"/>
      <xs:element type="xs:string" name="month"/>
      <xs:element type="xs:string" name="dayofweek"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="variableType">
    <xs:sequence>
      <xs:element type="xs:string" name="controllername"/>
      <xs:element type="xs:string" name="machinename"/>
```

```
      <xs:element type="xs:string" name="machinepart"/>
      <xs:element type="xs:string" name="observername"/>
      <xs:element type="scheduleType" name="schedule"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="dataType">
    <xs:sequence>
      <xs:element type="variableType" name="variable" maxOccurs="unbounded"
minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="serviceconfigType">
    <xs:sequence>
      <xs:element type="connectionType" name="connection"/>
      <xs:element type="dataType" name="data"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

8 JSON interface template

8.1 Request payload

```
[
   {
      "SecurityToken":"ADMIN"
   },
   {
      "InputArgs":[
      {
         "criteria":{
            "machinePartRegEx":"<ControllerName>/<machineName>/<machinePart>",
            "observerNameRegEx":"<observerName>",
            "timeStampFrom":"",
            "timeStampTo":"",
            "rowLimit":0
         }
      }
      ]
   }
]
```

8.2 Reply payload

```
{
   "type":"MULTI_OBJECT",
   "value":[
   {
      "name":"<observerName>",
      "unit":"<variableUnit>",
      "type":"<variableType>",
      ""<machinePart>"":{
         "fullPath":"IP <controller address> | <path>",
         "name":null
      },
      "currentValue":<value>,
      "trueText":<textRepresentation of Boolean true value>,
      "falseText":<textRepresentation of Boolean false value>
   }
   ]
}
```

Table PartValue:

```
mysql> show columns from partvalue;
+-----------------+-------------+------+-----+---------+----------------+
| Field           | Type        | Null | Key | Default | Extra          |
+-----------------+-------------+------+-----+---------+----------------+
| idPartValue     | int         | NO   | PRI | NULL    | auto_increment |
| event_timestamp | datetime    | YES  |     | NULL    |                |
| part_variable   | int         | YES  |     | NULL    |                |
| part_type       | int         | YES  |     | NULL    |                |
| value_bool      | int         | YES  |     | NULL    |                |
| value_float     | float       | YES  |     | NULL    |                |
| value_enum      | varchar(100)| YES  |     | NULL    |                |
| value_int       | int         | YES  |     | NULL    |                |
| value_word      | mediumtext  | YES  |     | NULL    |                |
| value_string    | varchar(100)| YES  |     | NULL    |                |
+-----------------+-------------+------+-----+---------+----------------+
```

Table PartVariable:

```
mysql> show columns from partvariable;
+--------------+--------------+------+-----+---------+----------------+
| Field        | Type         | Null | Key | Default | Extra          |
+--------------+--------------+------+-----+---------+----------------+
| id           | int          | NO   | PRI | NULL    | auto_increment |
| name         | varchar(100) | YES  |     | NULL    |                |
| machine_ip   | varchar(45)  | YES  |     | NULL    |                |
| machine_part | varchar(45)  | YES  |     | NULL    |                |
| unit         | varchar(45)  | YES  |     | NULL    |                |
| path         | varchar(255) | YES  |     | NULL    |                |
+--------------+--------------+------+-----+---------+----------------+
6 rows in set (0.08 sec)
```

Table ValueType

```
mysql> show columns from valuetype;
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| id    | int         | NO   | PRI | NULL    | auto_increment |
| name  | varchar(45) | YES  |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
```