CHIPKIN

# HTTP Data Driver FS-8705-100

Chipkin - Enabling Integration

salesgroup1@chipkin.com

Tel: +1 866 383 1657

Driver Version:
Document Revision: 6

# TABLE OF CONTENTS

# 1 HTTP Data Description

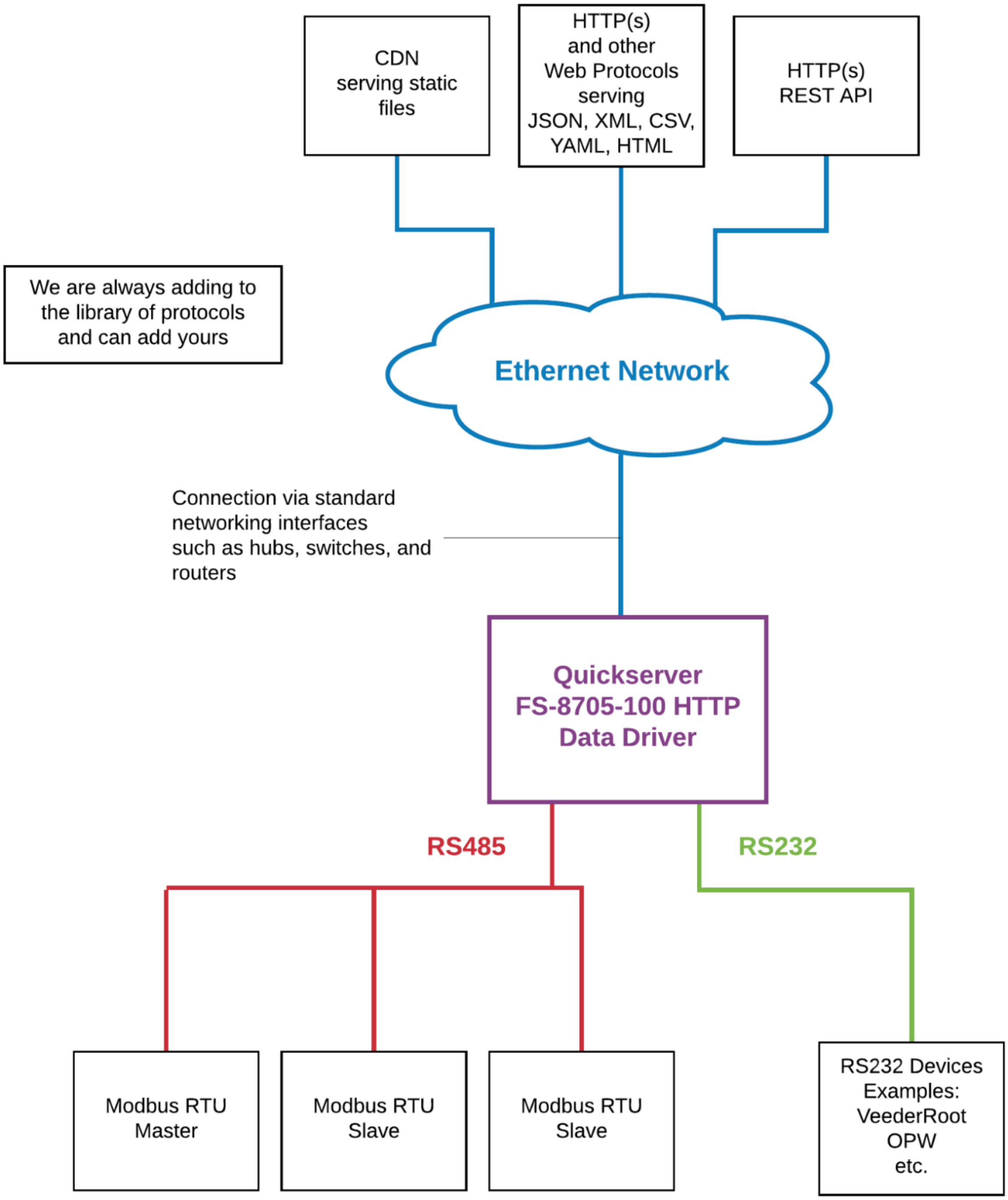
The HTTP Data Driver allows the FieldServer to poll HTTP(s) servers for data over Ethernet. The HTTP(s) server could be any HTTP(s) API, REST, CDN with static files, or any web-accessible file on the internet. The HTTP Data Driver supports polling multiple HTTP(s) servers or endpoints to retrieve data. The HTTP Data Driver supports parsing data in several formats including XML, JSON, YAML, CSV, Plain Text, or Regular Expressions.

The FieldServer is a Server, making requests to the configured URL. Upon receiving the data, the data is parsed based on the configured Tasks and the values are extracted. These values are stored on the FieldServer to be mapped to other protocols or simply to be viewed.

The information that follows describes how to expand upon the factory defaults provided in the configuration files included with the FieldServer.

## 2 Connection Description

This block diagram lists network connections that the FieldServer can poll for data over the HTTP(s) protocol, and pull into the FieldServer device using the HTTP Data Driver. This data can then be served using other protocols like Modbus® RTU/TCP, and BACnet®.


CDN serving static files
HTTP(s) and other Web Protocols serving JSON, XML, CSV, YAML, HTML
HTTP(s) REST API
We are always adding to the library of protocols and can add yours
Ethernet Network
Connection via standard networking interfaces such as hubs, switches, and routers
Quickserver FS-8705-100 HTTP Data Driver
RS485
RS232
Modbus RTU Master
Modbus RTU Slave
Modbus RTU Slave
RS232 Devices Examples: VeederRoot OPW etc.

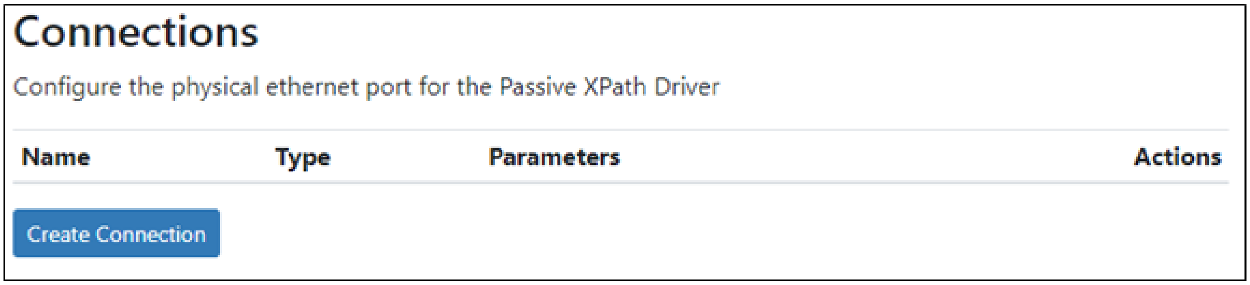
# 3 SACN Configuration

To configure the HTTP Data Driver, from the home page, visit the following link:
http://{IP_ADDRESS}/chipkin/ui/#/HTTPDataDriver

To configure the FieldServer, follow the instructions below to add a Connection (physical port), Nodes (HTTP(s) Urls and Ports to retrieve data), and finally Tasks (the data to extract from the HTTP(s) payload).

## 3.1 Create Connection

To set up the FieldServer HTTP Data Driver, first create a connection. The connection contains information about the physical port to use.



Connections

Configure the physical ethernet port for the Passive XPath Driver

| Name | Type | Parameters | Actions |
|---|---|---|---|

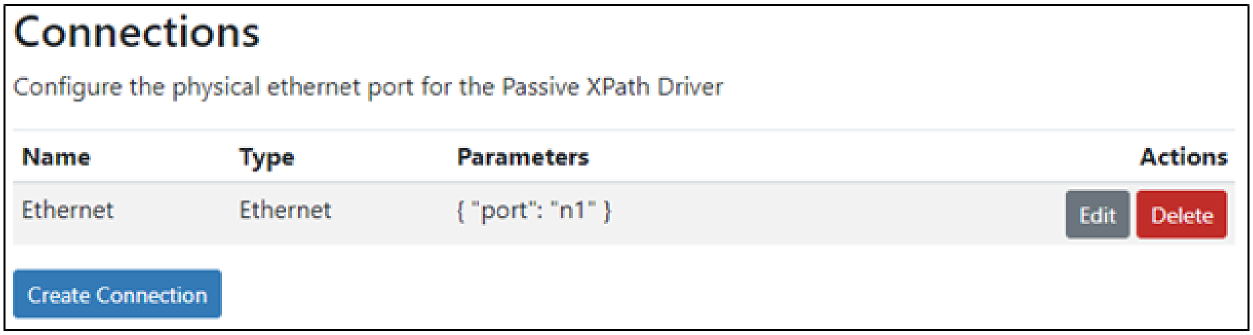
1. Click on the “Create Connection” button to open the Create Connection form.
2. Fill out the fields in the form. The fields are as follows:

| COLUMN TITLE | FUNCTION | LEGAL VALUES |
|---|---|---|
| Name | Name of the connection, used internally as an identifier for Nodes. | Text, must be unique |
| Type | The type of connection this is. Currently, only ethernet is supported. | **ethernet** |
| Parameters: Port | The physical port on the FieldServer to use | **n1** |

* Bolded values are defaults

3. Click the “Save” button to add the connection.

   If successful, the new entry will be populated in the Connections table:



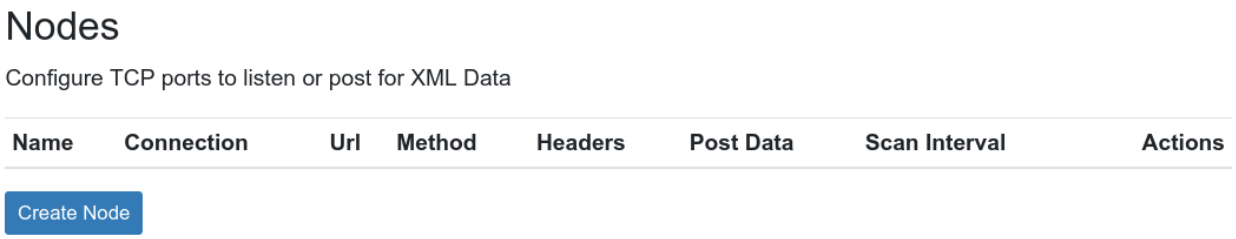
Connections

Configure the physical ethernet port for the Passive XPath Driver

| Name | Type | Parameters | Actions |
|---|---|---|---|
| Ethernet | Ethernet | { "port": "n1" } | Edit Delete |

## 3.2 Create Node

Follow the instructions below to configure the device to retrieve HTTP(s) data from the URL specified.



Nodes

Configure TCP ports to listen or post for XML Data

| Name | Connection | Url | Method | Headers | Post Data | Scan Interval | Actions |
|---|---|---|---|---|---|---|---|

1. Click on the “Create Node” button to open the Create Node form.
2. Fill out the fields in the form. The fields are as follows:

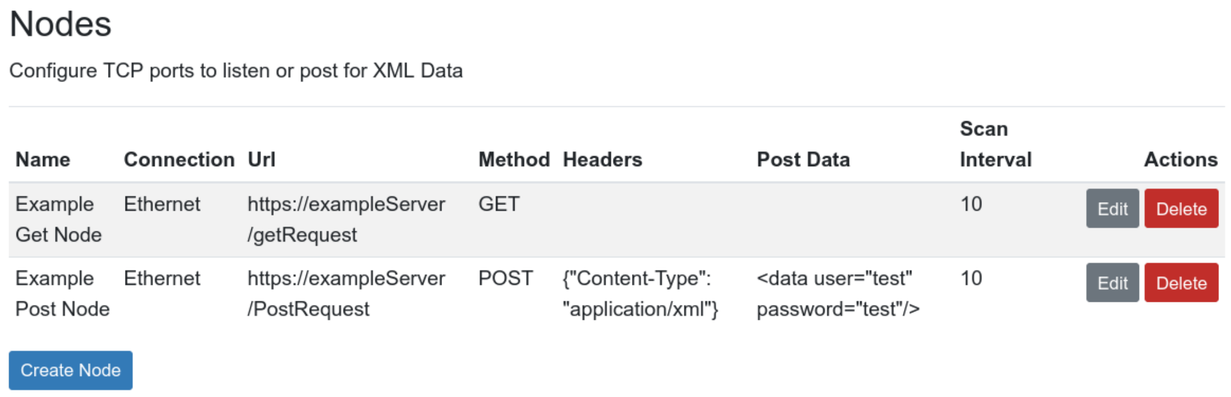
| COLUMN TITLE | FUNCTION | LEGAL VALUES |
|---|---|---|
| Name | The name of the HTTP(s) data being received. Used internally as an identifier for Tasks. | Text, must be unique |
| Connection | The name of the FieldServer’s physical port, linked via the Connection. | Text (Use the name of the Connection created in the previous section) |
| URL | The URL to retrieve data from. | Text, Any valid fully qualified URL.<br>IE: https://exampleServer/api/data |

| | | |
|---|---|---|
| Method | The HTTP(s) method to use for retrieving data<br>Only GET and POST HTTP(s) methods are currently supported | **GET** – Used to request data from the specified URL, resulting in a return payload<br>POST – Used to send data to the specified URL, such as request parameters, resulting in a return payload |
| Headers | Set request HTTP(s) headers to pass additional information to the server, in order to tailor the server's response. This can be any valid HTTP(s) header such as Authorization, Content-Type, etc. | Text, Valid HTTP headers formatted as JSON<br>EG: {“Content-Type”: “application/xml”} |
| Post Data | The data to be sent via HTTP(s) POST to the URL. This is ignored if method is set to GET | Text, the Server should dictate what this content is, as it can be anything |
| Scan Interval | How often to poll the URL in seconds. Recommended length is 60 seconds. | Integer |

*Bolded values are defaults

3. Click on the “Save” button to add the node.

   If successful, the new entry will be populated in the Nodes table:



## Nodes

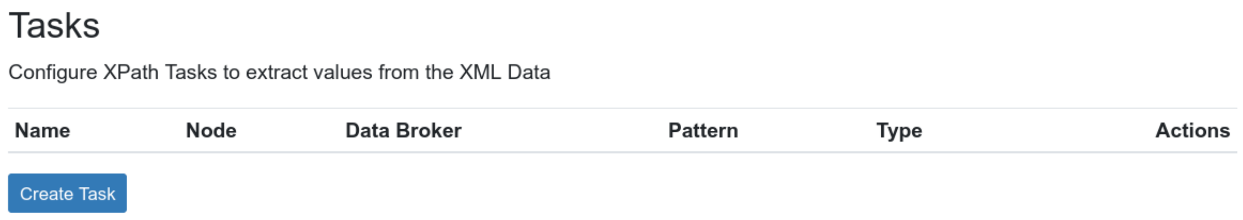
Configure TCP ports to listen or post for XML Data

| Name | Connection | Url | Method | Headers | Post Data | Scan Interval | Actions |
|---|---|---|---|---|---|---|---|
| Example Get Node | Ethernet | https://exampleServer/getRequest | GET | | | 10 | Edit Delete |
| Example Post Node | Ethernet | https://exampleServer/PostRequest | POST | {"Content-Type": "application/xml"} | <data user="test" password="test"/> | 10 | Edit Delete |

Repeat the above steps to add additional requests to receive other HTTP(s) data.

### 3.3 Create Task

Create tasks to extract values from the chosen type of HTTP(s) data.



Tasks

Configure XPath Tasks to extract values from the XML Data

| Name | Node | Data Broker | Pattern | Type | Actions |
|---|---|---|---|---|---|

1. Click on the “Create Task” button to open the Create Task form.
2. Fill out the fields in the form. The fields are as follows:

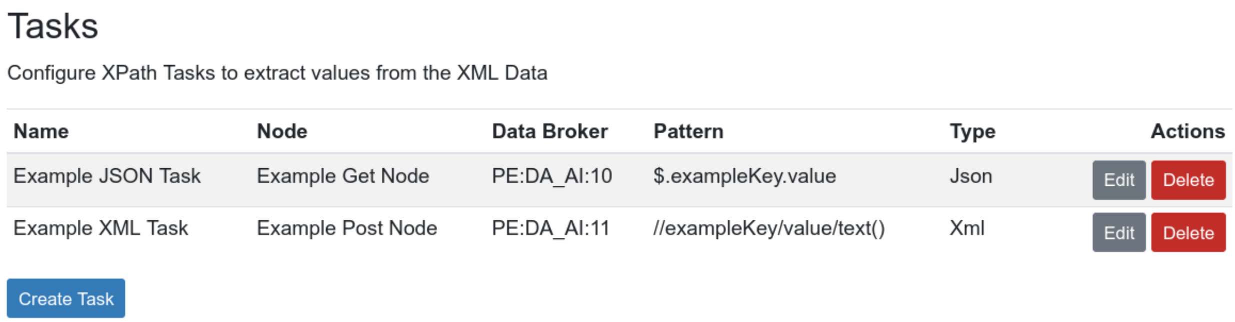
| COLUMN TITLE | FUNCTION | LEGAL VALUES |
|---|---|---|
| Name | The name of the variable to add. | Text, must be unique |
| Node | The node that this variable belongs to, linked via the Node. | Text (Use the name of a node created in the previous section) |
| DataBroker: Name | The name of a data array in the FieldServer protocol engine from where to retrieve a value. Data Array names are set up within the PE config.csv.<br>* See Section 6.2 for an example of a data array name within the PE config.csv | String - One of the Data Array names as defined in config.csv |
| DataBroker: Start | An offset in the data array to retrieve the value. This offset must be within the maximum range as defined by the Data_Array_Length of a Data_Array specified within the PE config.csv<br>* See Section 6.2 for an example of the Data_Array_Length within the PE config.csv | 0 to (“Data_Array_length” - 1) - An offset within a range up to a maximum as defined in config.csv |
| Pattern | The pattern used to extract data from the payload<br>* See Section 4 for examples | String – Dependent upon the chosen Type |
| Type | The expected payload to receive, can be any document formatted as XML, JSON, CSV, YAML, or plain text. If a Content-Type header was set in the Node, this selection should match that Content-Type. | JSON – JavaScript Object Notation<br>XML – Extensible Markup Language<br>CSV – Comma Separated Value |

| | | |
|---|---|---|
| | * See Section 4 for examples | YAML – Yet Another Markup Language<br>RegEx – Regular Expressions<br>Plain – Plain text |

* Bolded values are defaults

3. Click the "Save" button to add the task.

If successful, the new entry will be populated in the Tasks table:



Tasks

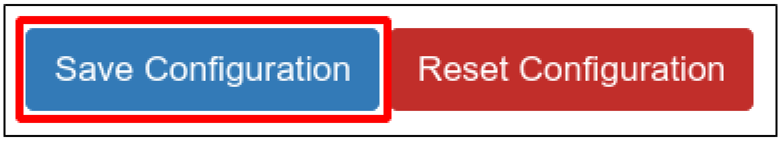
Configure XPath Tasks to extract values from the XML Data

| Name | Node | Data Broker | Pattern | Type | Actions |
|---|---|---|---|---|---|
| Example JSON Task | Example Get Node | PE:DA_AI:10 | $.exampleKey.value | Json | Edit Delete |
| Example XML Task | Example Post Node | PE:DA_AI:11 | //exampleKey/value/text() | Xml | Edit Delete |

Repeat the above steps to add additional  HTTP(s) data variables.

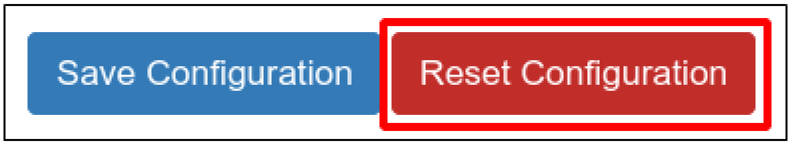
### 3.4 Saving the Configuration

When the configuration is complete, click on the "Save Configuration" button to save all of the updates and changes. For the configuration to take effect, reboot the system.



### 3.5 Resetting the Configuration

To clear the configuration and start over, click the "Reset Configuration" button.  Then follow the instructions in the sections above to create new connections, nodes, and tasks.



# 4 Data and Pattern Examples

## 4.1 JSON

### 4.1.1 Data

```
{
 "store": {
  "sensor": [
   {
    "name": "Lobby light sensor",
    "unit": "lux",
    "value": "99.9"
   },
   {
    "name": "Lobby Temperature sensor",
    "unit": "F",
    "value": "78.3"
   },
   {
    "name": "Lobby Humidity sensor",
    "unit": "RH%",
    "value": "56"
   }
  ],
  "fuel": {
   "tank": [
    {
     "name": "Tank 1",
     "unit": "%",
     "value": "12",
     "installYear": "2020"
    },
    {
     "name": "Tank 2",
     "unit": "%",
     "value": "100",
     "installYear": "2019"
    },
    {
     "name": "Tank 3",
     "unit": "%",
          "value": "80",
          "installYear": "2018"
        }
      ]
    }
  }
}
```

### 4.1.2 Task Patterns

| JSON PATTERNS | DATA |
|---|---|
| $.STORE.SENSOR[0].VALUE | 99.9 |
| $.STORE.SENSOR[2].VALUE | 56 |
| $.STORE.SENSOR[3].VALUE | ERROR=NO ELEMENTS FOUND |
| $.STORE.FUEL.TANK[1].VALUE | 100 |
| $.STORE.FUEL.TANK[@NAME=’TANK 1’].VALUE | 12 |
| $..TANK[@INSTALLYEAR=’2019’].VALUE | 100 |
| $..SENSOR[2].VALUE | 56 |

### 4.1.3 Resources

#### 4.1.3.1 https://jsonpath.com/ - JSON Path Evaluator

#### 4.1.3.2 https://goessner.net/articles/JsonPath/index.html#e2 - Overview and examples of JSON Path expressions

## 4.2 XML

### 4.2.1 Data

```
<store>
  <sensor>
      <name>Lobby light sensor</name>
      <unit>lux</unit>
      <value>99.9</value>
  </sensor>
  <sensor>
      <name>Lobby Temperature sensor</name>
      <unit>F</unit>
      <value>78.3</value>
  </sensor>
  <sensor>
      <name>Lobby Humidity sensor</name>
      <unit>RH%</unit>
      <value>56</value>
  </sensor>
  <fuel>
      <tank installYear='2020'>
      <name>Tank 1</name>
      <unit>%</unit>
      <value>12</value>
      </tank>
      <tank installYear='2019'>
      <name>Tank 2</name>
      <unit>%</unit>
      <value>100</value>
      </tank>
      <tank installYear='2018'>
      <name>Tank 3</name>
      <unit>%</unit>
      <value>80</value>
      </tank>
  </fuel>
</store>
```

### 4.2.2 Task Pattern Examples

| XML PATTERNS | DATA |
|---|---|
| /store/sensor[1]/value/text() | 99.9 |
| /store/sensor[3]/value/text() | 56 |
| /store/sensor[4]/value/text() | Error=No elements found |
| /store/fuel/tank[2]/value/text() | 100 |
| /store/fuel/tank[name=’Tank 1’]/value/text() | 12 |
| //fuel/tank[@installYear=’2019’]/value/text() | 100 |
| //sensor[3]/value/text() | 56 |

### 4.2.3 Resources

**4.2.3.1** https://devhints.io/HTTP Data - An in depth cheatsheet about HTTP Data patterns using HTML

**4.2.3.2** https://www.freeformatter.com/HTTP Data-tester.html - HTTP Data tester and evaluator, contains examples

## 4.3 YAML

### 4.3.1 Data

```
---
store:
 fuel:
  -
   installYear: 2020
   name: "Tank 1"
   unit: percent
   value: 12
  -
   installYear: 2019
   name: "Tank 2"
   unit: percent
   value: 100
  -
   installYear: 2018
   name: "Tank 3"
   unit: percent
   value: 80

 sensor:
  -
   name: "Lobby light sensor"
   unit: lux
   value: 99.9
  -
   name: "Lobby Temperature sensor"
   unit: F
   value: 78.3
  -
   name: "Lobby Humidity sensor"
   unit: RH%
   value: 56
```

### 4.3.2 Task Patterns

YAML payloads are converted to JSON, and parsed accordingly

| JSON PATTERNS | DATA |
|---|---|
| $.store.sensor[0].value | 99.9 |
| $.store.sensor[2].value | 56 |
| $.store.sensor[3].value | Error=No elements found |
| $.store.fuel.tank[1].value | 100 |
| $.store.fuel.tank[@name='Tank 1'].value | 12 |
| $..tank[@installYear='2019'].value | 100 |
| $..sensor[2].value | 56 |

### 4.3.3 Resources – See 4.1.3 JSON Resources

## 4.4 CSV

### 4.4.1 Data

```
1, 12345678, 24.5, Temperature,
2, 12345678, 7.5, Humidity,
3, 12345678, 19.23, Outdoor Temperature,
4, 12345678, 06:20:00, Sunrise,
5, 12345678, 2020-Aug-19, Date,
```

When CSV data is processed, it's converted into a JSON payload

```
[
  ["1", "12345678", "24.5", "Temperature"],
  ["2", "12345678", "7.5", "Humidity"],
  ["3", "12345678", "19.23", "Outdoor Temperature"],
  ["4", "12345678", "06:20:00", "Sunrise"],
  ["5", "12345678", "2020-Aug-19", "Date"]
]
```

### 4.4.2 Task Patterns

CSV payloads are converted to JSON, and parsed accordingly

| JSON PATTERNS | DATA |
|---|---|
| $.2.2 | 19.23 |
| $.7.2 | Error=No elements found. |
| $.0.3 | Temperature |
| $.4.2 | 2020-Aug-19 |

## 4.5 RegEx

### 4.5.1 Data

```
Lobby light sensor: 99.9 lux
Lobby Temperature sensor: 78.3 F
Lobby Humidity sensor: 56 RH%
fuel tank 1: 12%
fuel tank 2: 100%
fuel tank 3: 80%
```

### 4.5.2 Task Patterns

| REGULAR EXPRESSION PATTERN | RESULT |
|---|---|
| [0-9]{2,} lux | 99.9 lux |

* The above Regular Expression will return `99.9 lux` which the FieldServer will then process as `99.9`

## 4.6 Plain Text

### 4.6.1 Data

```
24.5
```

### 4.6.2 Task Patterns

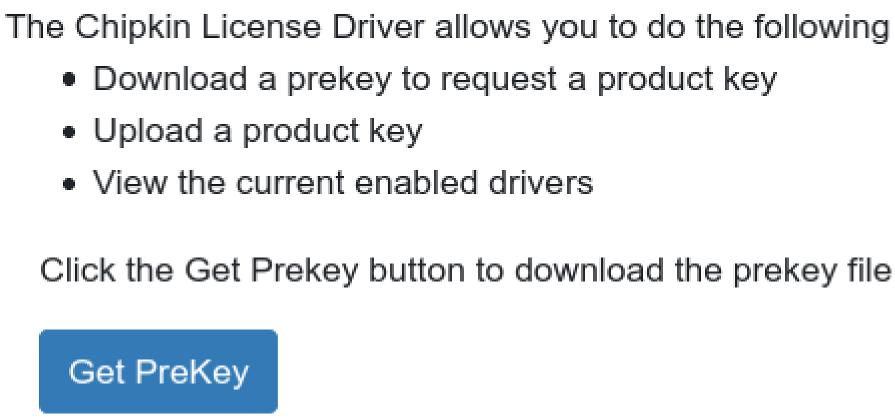
None, plain text payloads should just be their raw values, and do not require a Task Pattern

# 5 License

Some drivers such as HTTP Data require a license product key. To generate this license product key a pre-key from the hardware is required.

## 5.1 How to generate a pre-key

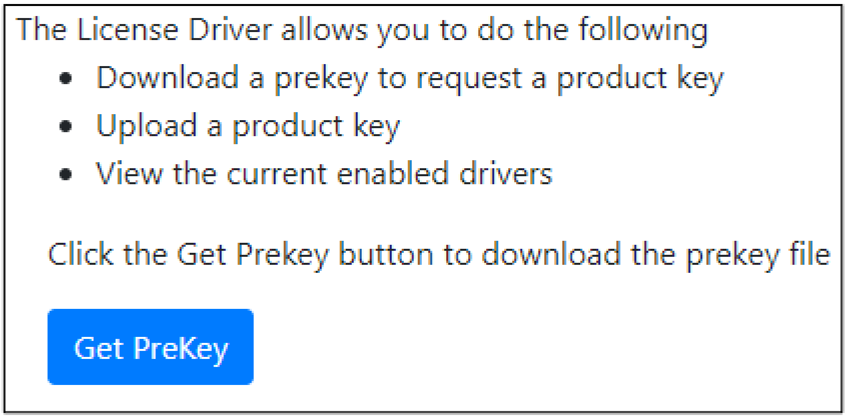
1. Goto the license page http://{IP_ADDRESS}/chipkin/ui/#/chipkinLicenseDriver
2. Click the Get Pre-Key button.



The Chipkin License Driver allows you to do the following

- Download a prekey to request a product key
- Upload a product key
- View the current enabled drivers

Click the Get Prekey button to download the prekey file



The License Driver allows you to do the following

- Download a prekey to request a product key
- Upload a product key
- View the current enabled drivers

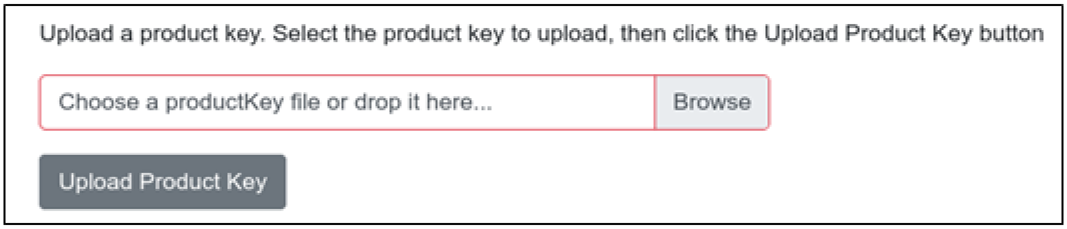
Click the Get Prekey button to download the prekey file

A pre-key.txt file will be created and downloaded to your system. Send this pre-key.txt and your Job number (FSE####) to Chipkin support.

## 5.2 How to activate a product key

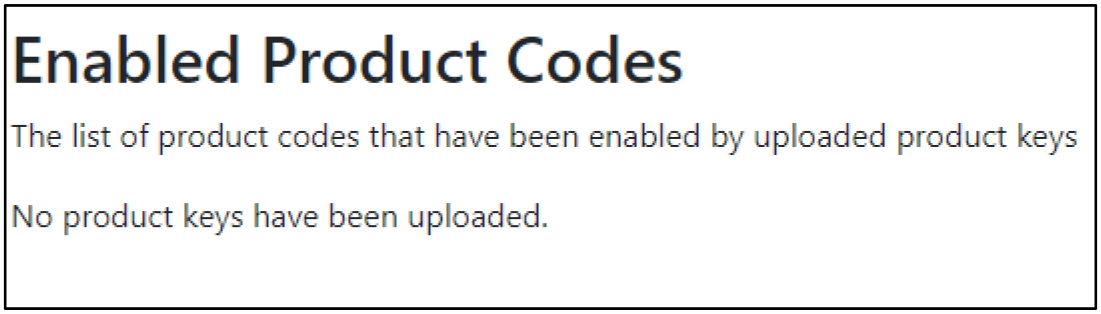
Chipkin support can generate a license product key from the hardware pre-key. The product key will be sent as a text file via email.

1. Goto the license page http://{IP_ADDRESS}/chipkin/ui/#/chipkinLicenseDriver
2. Click “Browse” button and select the productkey-XXXXX.txt file provided to you by Chipkin Support.
3. Click the “Upload Product Key” button and wait for the product key to finish uploading.



Upload a product key. Select the product key to upload, then click the Upload Product Key button

The list of enabled product codes can be viewed in the “Enabled Product Codes” list.



Enabled Product Codes

The list of product codes that have been enabled by uploaded product keys

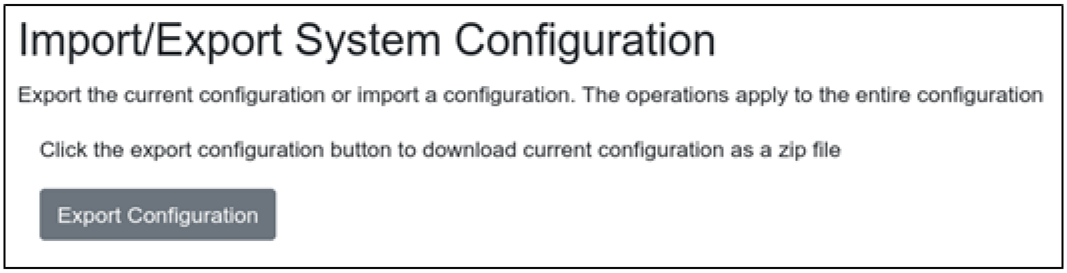
No product keys have been uploaded.

# 6 Importing and Exporting Configurations

It is possible to export the current configuration to back it up or simply to make some edits. Users can also import either the entire configuration via a zip file or a PE (Protocol Engine) configuration.

## 6.1 How to Export the Configuration

1. Goto the system configuration page http://{IP_ADDRESS}/chipkinConfiguration/ui/
2. Click the Export Configuration button.



Import/Export System Configuration

Export the current configuration or import a configuration. The operations apply to the entire configuration

Click the export configuration button to download current configuration as a zip file

## 6.2 How to Import the Configuration

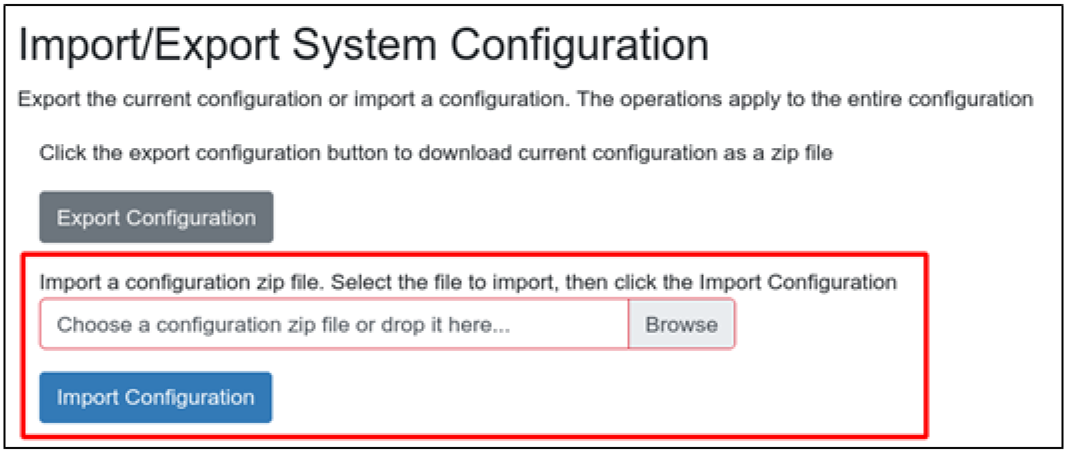
The file to import the configuration must be a zip file. The zip file should contain the following folders:

- ae - this folder contains any configuration files for the ae configuration
- documents - this folder contains any driver specific documents. For example, license product keys, etc.
- pe - this folder contains one config.csv file for the pe configuration.

To make sure the folder directory is correct, do an Export first, then extract the files, edit them, then zip them up again.

To import the configuration:

1. Goto the system configuration page http://{IP_ADDRESS}/chipkin/ui/#/chipkinConfiguration
2. Click the “Browse” button in the “Import/Export System Configuration” section and select the zip file containing the configuration to import.
3. Click the “Import Configuration” button and wait for the configuration to finish importing.
4. If successful, a success message will appear prompting a reboot of the FieldServer for the changes to take effect.



Import/Export System Configuration

Export the current configuration or import a configuration. The operations apply to the entire configuration

Click the export configuration button to download current configuration as a zip file

Import a configuration zip file. Select the file to import, then click the Import Configuration

## 6.3 Example AE Configuration

```
{
  "ae": {
      "httpDataDriver": {
      "connections": [
      {
            "type": "ethernet",
            "name": "Ethernet",
            "parameters": { "port": "n1" }
      }
      ],
      "nodes": [
      {
            "connection": "Ethernet",
            "method": "GET",
            "name": "Example Time API",
            "url":
"https://www.timeapi.io/api/Time/current/zone?timeZone=America/Vancouver",
            "scanInterval": 5
      }
      ],
      "tasks": [
      {
            "node": "Example Time API",
            "type": "json",
            "name": "Year",
            "dataBroker": { "pe": { "Name": "DA_AI", "Start": "0" } },
            "pattern": "$.year"
      },
      {
            "node": "Example Time API",
            "type": "json",
            "name": "Month",
            "dataBroker": { "pe": { "Name": "DA_AI", "Start": "1" } },
            "pattern": "$.month"
      },
      {
            "node": "Example Time API",
            "type": "json",
            "name": "Day",
            "dataBroker": { "pe": { "Name": "DA_AI", "Start": "2" } },
            "pattern": "$.day"
      },
      {
            "node": "Example Time API",
            "type": "json",
            "name": "Hour",
            "dataBroker": { "pe": { "Name": "DA_AI", "Start": "3" } },
            "pattern": "$.hour"
      },
      {
            "node": "Example Time API",
            "type": "json",
            "name": "Minute",
            "dataBroker": { "pe": { "Name": "DA_AI", "Start": "4" } },
            "pattern": "$.minute"
      },
      {
            "node": "Example Time API",
            "type": "json",
            "name": "Seconds",
            "dataBroker": { "pe": { "Name": "DA_AI", "Start": "5" } },
            "pattern": "$.seconds"
      },
      {
            "node": "Example Time API",
            "type": "json",
            "name": "Milliseconds",
            "dataBroker": { "pe": { "Name": "DA_AI", "Start": "6" } },
            "pattern": "$.milliSeconds"
      }
```

```
      ]
      }
  }
}
```

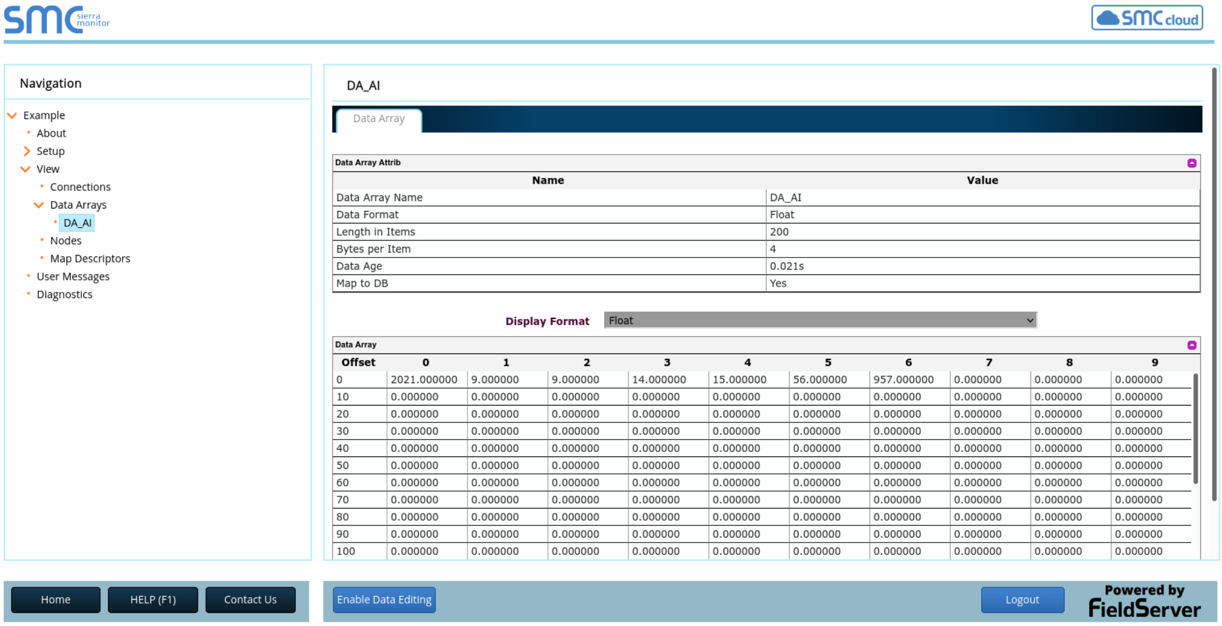
## 6.4 Example PE Configuration

```
Bridge
Title
Example


Data_Arrays
Data_Array_Name, Data_Format, Data_Array_Length
DA_AI, float, 200
```

# 7 Working Example

Following the instructions of section 6 (How to Import The Configuration), and using the configurations from section 6.1 and 6.2, we can set up a working example environment that queries the timeapi.io API and sets the first seven offsets to the date and time.



Data Array Attrib

| Name | Value |
|---|---|
| Data Array Name | DA_AI |
| Data Format | Float |
| Length in Items | 200 |
| Bytes per Item | 4 |
| Data Age | 0.021s |
| Map to DB | Yes |

Display Format: Float

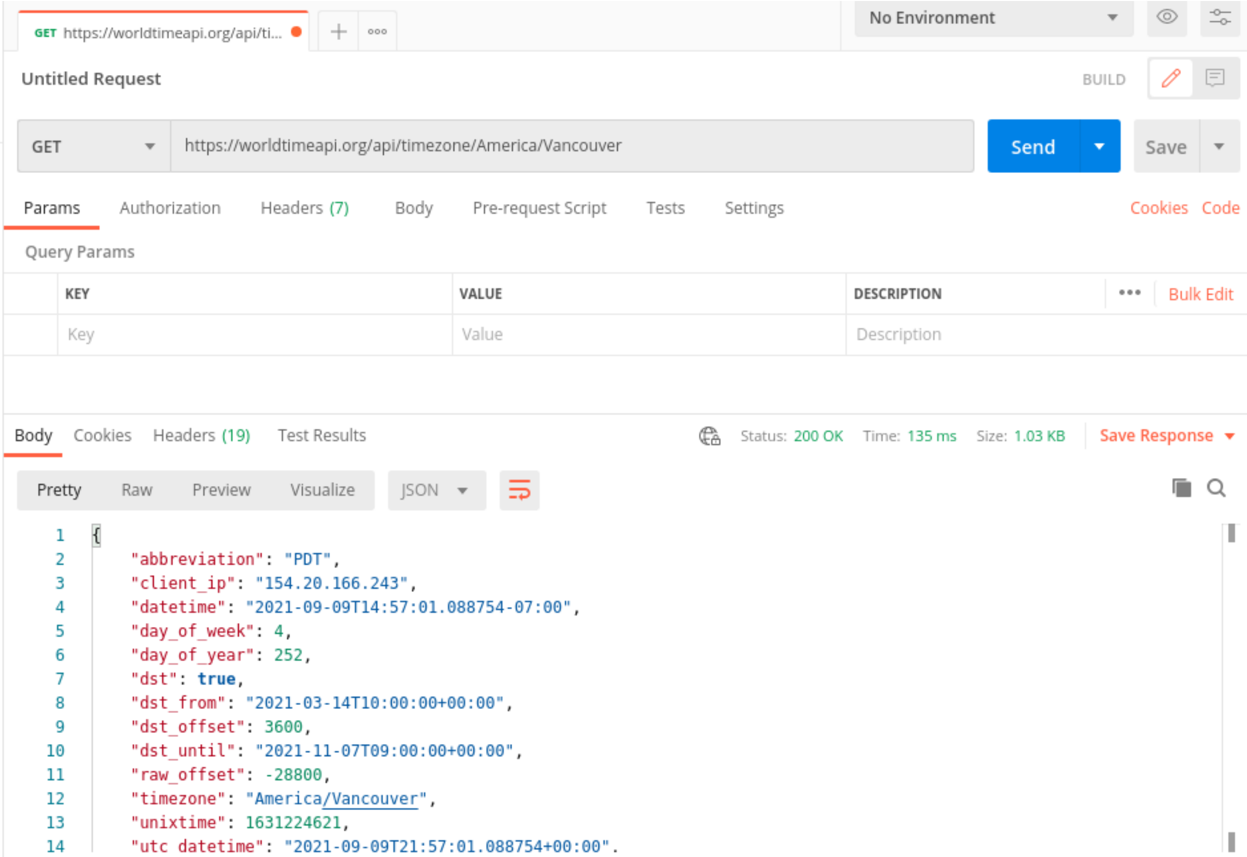
Data Array

| Offset | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2021.000000 | 9.000000 | 9.000000 | 14.000000 | 15.000000 | 56.000000 | 957.000000 | 0.000000 | 0.000000 | 0.000000 |
| 10 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 20 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 30 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 40 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 50 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 60 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 70 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 80 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 90 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| 100 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |

# 8 Advanced Topics

## 8.1 Debugging the HTTP Data Driver

- If the FieldServer is not receiving any HTTP(s) data, verify that the Server device is sending HTTP(s) data and that the FieldServer device is requesting this data from the appropriate URL. Checking the URL directly can be the easiest way to determine if it returns data. A user can also use a tool like Postman to do a more in depth test of the API.



GET https://worldtimeapi.org/api/timezone/America/Vancouver

Status: 200 OK Time: 135 ms Size: 1.03 KB

```
{
    "abbreviation": "PDT",
    "client_ip": "154.20.166.243",
    "datetime": "2021-09-09T14:57:01.088754-07:00",
    "day_of_week": 4,
    "day_of_year": 252,
    "dst": true,
    "dst_from": "2021-03-14T10:00:00+00:00",
    "dst_offset": 3600,
    "dst_until": "2021-11-07T09:00:00+00:00",
    "raw_offset": -28800,
    "timezone": "America/Vancouver",
    "unixtime": 1631224621,
    "utc_datetime": "2021-09-09T21:57:01.088754+00:00",
```

- Check the Stats page to see if the FieldServer is receiving data.
- If the FieldServer is not extracting values from the data, check that the Pattern is valid and the Type selected matches the pattern used, and is not mismatched.
  EG: JSON data, and JSON as the Type, but the Pattern is for XML

## 8.2 How to take a Diagnostic Log

Please see https://store.chipkin.com/articles/how-to-take-a-diagnostic-log-on-a-quickserverfieldserver for the most up-to-date information

## 8.3 Large Payloads

The HTTP Data Driver works serially, and can handle many Nodes and Tasks at one time.

The QuickServer hardware can hang when processing large payloads of data from a single Node, and can cause the device to fall over into recovery mode.

We recommend splitting up Nodes in such a way to limit the size of individual payloads. Limiting what the payload contains by only retrieving the needed points, or configuring the QuickServer configuration to poll individual devices rather than all at once are two such ways to accomplish this.

As an example, let’s say you’re retrieving temperature data from thermostats within a large building with 1000 thermostats. Each thermostat by default responds to each request with 10 settings points, 10 temperature points, and additional metadata. Let’s say 40 lines of data per thermometer, and  you only need 1 or 2 points of the 40 returned.

By limiting each thermostat to only send the required temperature points, we can reduce the size of the payload immediately by +90%.

Additionally, if the set up allows for it, querying each thermostat individually will inherently reduce the size of each payload. Since the driver works in a serial manner, this allows the QuickServer hardware time to respond to other requests between each Node process.

## 8.4 JSON GET database

The data in the database/ data arrays can be retrieved via HTTP GET.

Open API/Swagger documentation /chipkin/ui/#/swagger

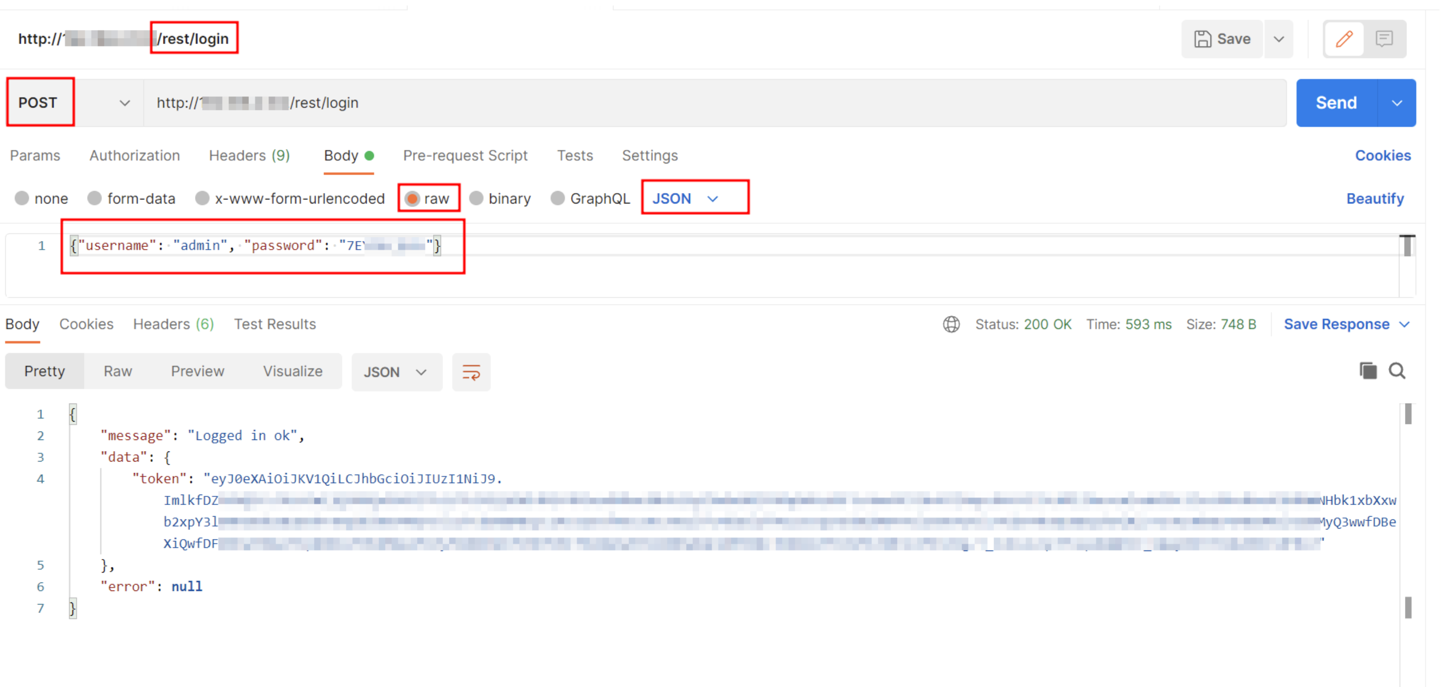
### 8.4.1 Request a login token

To make a HTTP request for data the HTTP client must get a token (happn_token_https) first.

**HTTP POST /rest/login**

```
Header: Content-Type: application/json
{"username": "admin", "password": "PASSWORD"}
```

Will return a token that you can use for subsequent requests.



http://[redacted]/rest/login

POST http://[redacted]/rest/login

Body: ○ none ○ form-data ○ x-www-form-urlencoded ● raw ○ binary ○ GraphQL JSON

```
{"username": "admin", "password": "7E[redacted]"}
```

Body — Status: 200 OK Time: 593 ms Size: 748 B — JSON

```
{
    "message": "Logged in ok",
    "data": {
        "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
            ImlkfDZ[redacted]NHbk1xbXxw
            b2xpY3l[redacted]MyQ3wwfDBe
            XiQwfDF[redacted]"
    },
    "error": null
}
```

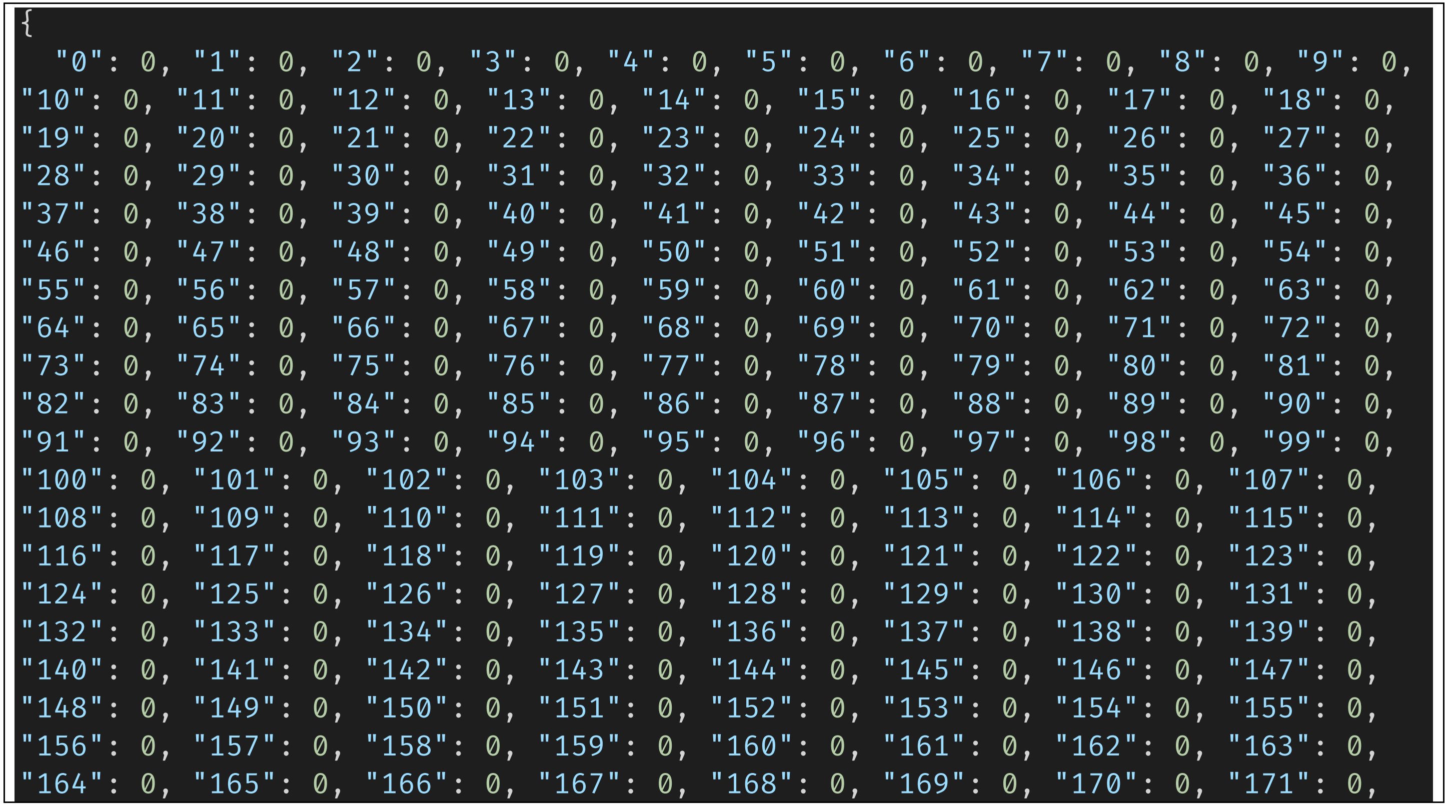
### 8.4.2 Get whole Data Array

Get a PE data array as JSON. This is the preferred way of getting data from the QuickServer

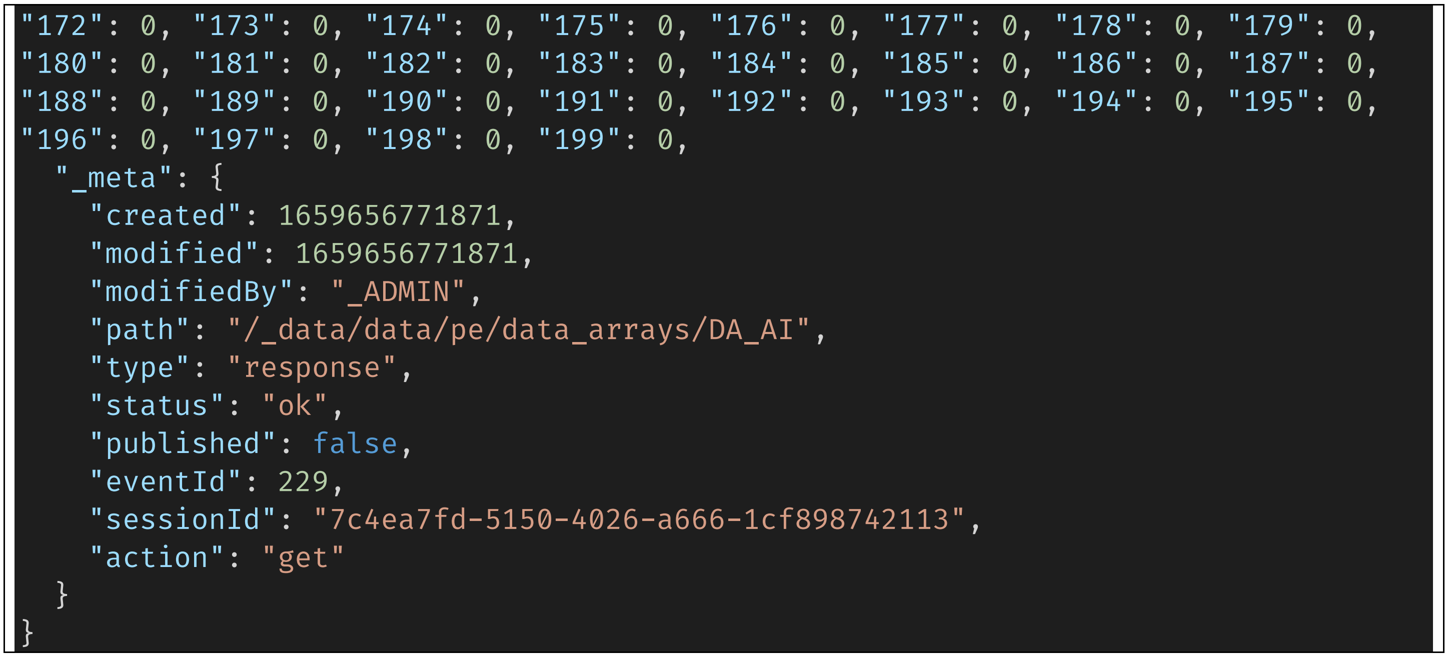
HTTP GET http://{IP_ADDRESS}/cape/getData?path=/pe/data_arrays/{DATA_ARRAY}

Example request: http://192.168.2.102/cape/getData?path=/pe/data_arrays/DA_AI

Example payload



```
{
  "0": 0, "1": 0, "2": 0, "3": 0, "4": 0, "5": 0, "6": 0, "7": 0, "8": 0, "9": 0,
"10": 0, "11": 0, "12": 0, "13": 0, "14": 0, "15": 0, "16": 0, "17": 0, "18": 0,
"19": 0, "20": 0, "21": 0, "22": 0, "23": 0, "24": 0, "25": 0, "26": 0, "27": 0,
"28": 0, "29": 0, "30": 0, "31": 0, "32": 0, "33": 0, "34": 0, "35": 0, "36": 0,
"37": 0, "38": 0, "39": 0, "40": 0, "41": 0, "42": 0, "43": 0, "44": 0, "45": 0,
"46": 0, "47": 0, "48": 0, "49": 0, "50": 0, "51": 0, "52": 0, "53": 0, "54": 0,
"55": 0, "56": 0, "57": 0, "58": 0, "59": 0, "60": 0, "61": 0, "62": 0, "63": 0,
"64": 0, "65": 0, "66": 0, "67": 0, "68": 0, "69": 0, "70": 0, "71": 0, "72": 0,
"73": 0, "74": 0, "75": 0, "76": 0, "77": 0, "78": 0, "79": 0, "80": 0, "81": 0,
"82": 0, "83": 0, "84": 0, "85": 0, "86": 0, "87": 0, "88": 0, "89": 0, "90": 0,
"91": 0, "92": 0, "93": 0, "94": 0, "95": 0, "96": 0, "97": 0, "98": 0, "99": 0,
"100": 0, "101": 0, "102": 0, "103": 0, "104": 0, "105": 0, "106": 0, "107": 0,
"108": 0, "109": 0, "110": 0, "111": 0, "112": 0, "113": 0, "114": 0, "115": 0,
"116": 0, "117": 0, "118": 0, "119": 0, "120": 0, "121": 0, "122": 0, "123": 0,
"124": 0, "125": 0, "126": 0, "127": 0, "128": 0, "129": 0, "130": 0, "131": 0,
"132": 0, "133": 0, "134": 0, "135": 0, "136": 0, "137": 0, "138": 0, "139": 0,
"140": 0, "141": 0, "142": 0, "143": 0, "144": 0, "145": 0, "146": 0, "147": 0,
"148": 0, "149": 0, "150": 0, "151": 0, "152": 0, "153": 0, "154": 0, "155": 0,
"156": 0, "157": 0, "158": 0, "159": 0, "160": 0, "161": 0, "162": 0, "163": 0,
"164": 0, "165": 0, "166": 0, "167": 0, "168": 0, "169": 0, "170": 0, "171": 0,
```



```
"172": 0, "173": 0, "174": 0, "175": 0, "176": 0, "177": 0, "178": 0, "179": 0,
"180": 0, "181": 0, "182": 0, "183": 0, "184": 0, "185": 0, "186": 0, "187": 0,
"188": 0, "189": 0, "190": 0, "191": 0, "192": 0, "193": 0, "194": 0, "195": 0,
"196": 0, "197": 0, "198": 0, "199": 0,
  "_meta": {
    "created": 1659656771871,
    "modified": 1659656771871,
    "modifiedBy": "_ADMIN",
    "path": "/_data/data/pe/data_arrays/DA_AI",
    "type": "response",
    "status": "ok",
    "published": false,
    "eventId": 229,
    "sessionId": "7c4ea7fd-5150-4026-a666-1cf898742113",
    "action": "get"
  }
}
```

### 8.4.3 Get a single data point from a Data Array

Get a single data point from a data array. This can be useful if you don’t need the whole database.

HTTP GET /cape/api/db/pe?Name=DA_AI&Start=0
header: Cookie: happn_token=${token}

```
{
  "value": 42.580002,
  "_meta": {
    "created": 1630349814887,
    "modified": 1658858901781,
    "path": "/_data/data/pe/data_arrays/DA_AI/0"
  }
}
```

# 9 Marketing

## 9.1 Case Study

A series of case studies for HTTP Data Driver can be found here

[ToDo] – Add Case Study or link to Case Studies…

## 9.2 Keywords

XML, HTTP, HTTPS, REST, API, CDN, XML, XML-HTTP, JSON, YAML, CSV, RegEx, TLS, XPath, JSONPath

## 9.3 Glossary of Terms

1. **HTTP(s)** – Hypertext Transfer Protocol (secure)
2. **API** – Architecture Programming Interface
3. **REST** – Representational State Transfer
4. **CDN** – Content Delivery Network
5. **TLS** – Transport Layer security
6. **CSV** – Comma Separated Value
7. **XML** – Extensible Markup Language
8. **JSON** – JavaScript Object Notation
9. **YAML** – Yet Another Markup Language
10. **RegEx** – Regular Expression
11. **PE** - Protocol Engine
12. **AE** - Application Engine

# 10 Revision History

This table summarizes the update history for this document. Please contact Chipkin for an updated version of this document if required.

| DATE | RESP | DOC. REV. | COMMENT |
|---|---|---|---|
| 21 May 2021 | JJK | 1 | Created initial document |
| 07 Sep 2021 | JJK | 2 | Updated document with additional sections |
| 08 Sep 2021 | JJK | 3 | Fixed typos |
| 06 Oct 2021 | YC | 4 | Updated to latest template |
| 26 Jul 2022 | SWS | 5 | Added section about HTTP Get for database |
| 04 Aug 2022 | SWS | 6 | Updated HTTP Get database section with getData method |