THE UNLIMITED INDUSTRIAL
IOT CONNECTIVITY SOFTWARE

Reference Manual
vNode RestApi Client

v-1.17.0-180615

Index

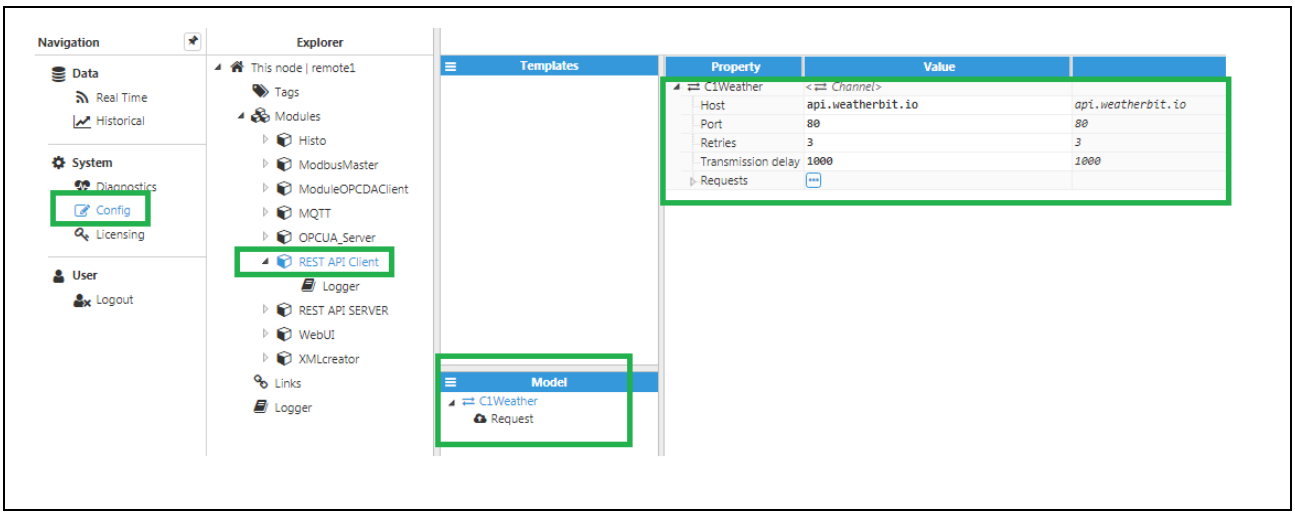
Introduction

RestApiClient is a vNode module capable of creating REST GET request to obtain data from a REST server, and save this data in one or various vNode tags.

Module Configuration

Module Calls

When selecting the RestApiClient in the configuration menu, the following options will be shown:



Step 1 : channel options

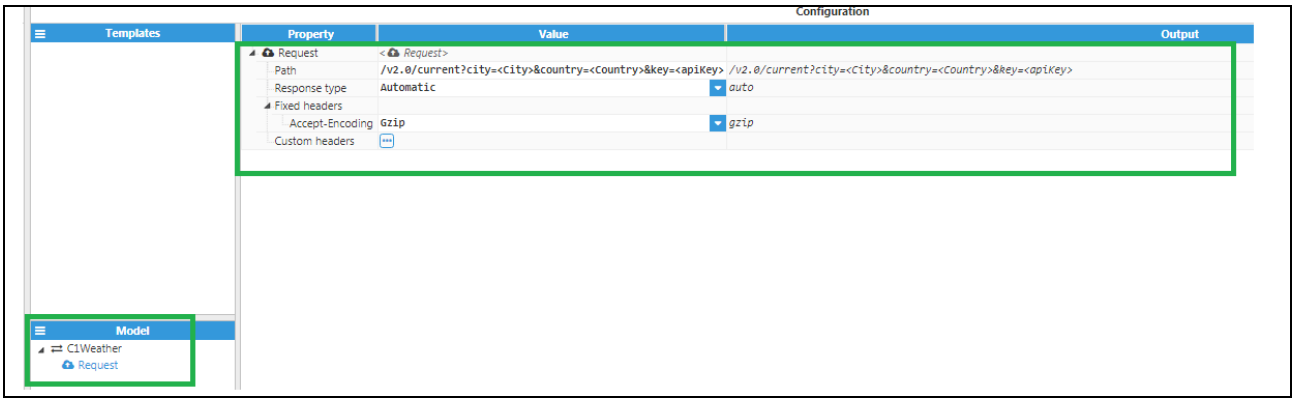
Channel Options

A channel represents a connection to a REST server. As such, the channel needs to specify a **Host** and a **Port** to connect to the server, as well as the number of **retries** for each request on this channel. Finally, the **transmission delay** is the delay between one request and the next.

In this example, the host is the REST server of "api.weatherbit.io", which is used to obtain weather data.

Request Options

Requests have the following options:



Step 2: Request options

The path is used along with the host and port to get a resource from the REST server. This path can have user defined variables enclosed in <> that will be resolved according to the value of these variables defined in each tag. In this example, we have two variables, <City> and <Country>.

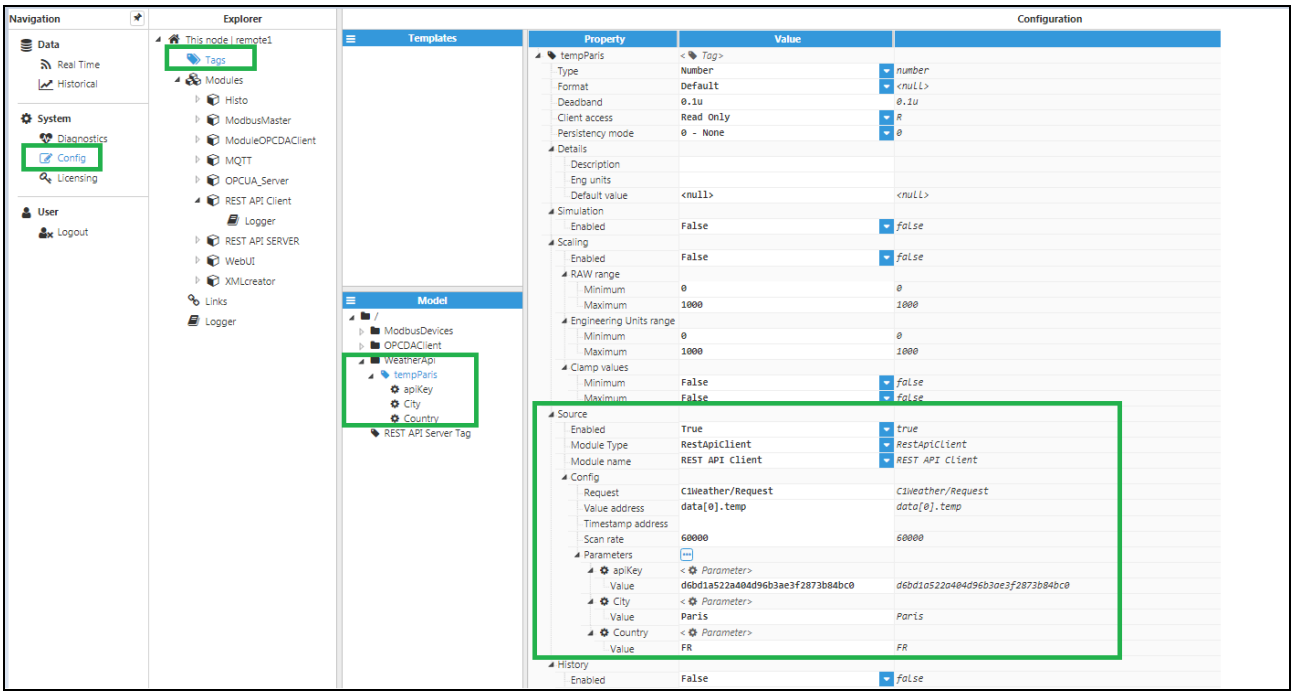
The response type is used to select which parser to use to parse the response. There are two parsers available, "**XML**" and "**JSON**", as well as an "**Automatic**" response type, which selects the parsers automatically based on the "Content-Type" header of the response. The supported MIME types are application/json and application/xml. If this header is not present, or the MIME type is not supported, it will use the **JSON** parser by default.

Next, the fixed headers are a set of headers that must be present in the request, in this case it's only one header, the "Accept-Encoding" header, used to tell the server whether the client can accept compressed payloads. The recommended value is "**Gzip**", in order to reduce the size of the response (reducing network consumption), however, it can also be set to "**None**" if desired, which will slightly reduce the CPU load on the server and client.

Finally, custom headers are headers that are not set by default by the client, but can be provided by the user. This is used if the server requires a certain header for it to process the request. In this example, there's a custom header called "**accept-charset**" that tells the server that the client can accept utf8 and iso-8859 with a 0.5 weight.

Tag Configuration

By setting a tag's source to RespApiClient, the tag will have the following options :



The **Request** selects which request from which **Channel** to use to gather data from this tag. The format is "Channel/Request". The **Value address** and the **Timestamp address** are used to get the data and timestamp for this tag from the response received from the REST server. For example, in this case the server will respond with the following JSON:

```
{
        data.[0]temp :{
        }
}
```

As we can see, the value address points to the value in the JSON that we want to save.

More examples (including XML responses) can be found in **Appendix #1.**

The **Scan rate** is how often a request will be sent to get the value for this tag. A value of 30000, like in this example, means that a new request will be sent every 30 seconds. This value should be set as to avoid spamming the server with many requests. Nonetheless, all the tags that have the same URL and same scan rate will use the same request to get their data. For example, if the url is "www.example.com/data.json", and 10 tags refer to this request, only one request will be created.

Finally, each tag can have one or more parameters that will resolve the path of the request. These parameters need to have the same name as in the request. For example, in our request we had the parameters <Country> and <City>, and in the tag we give them the values "Paris". This resolves the path of the request to:

- **Original path**
- **Resolved path**
- **Full URL**

Appendix #1

Selecting a JSON property
For this example we will use the following JSON file (part of a JSON from api used above):


```
{
	"data": [{
				"rh": 38,
				"pod": "d",
				"lon": 2.3488,
				"pres": 1002.6,
				"timezone": "Europe/Paris",
				"ob_time": "2019-05-07 13:30",
				"country_code": "FR",
				"clouds": 100,
				"ts": 1557235800,
				"solar_rad": 168.7,
				"state_code": "11",
				"lat": 48.85341,
				"wind_spd": 3.1,
				"wind_cdir_full": "south",
				"wind_cdir": "S",
				"slp": 1014,
				"vis": 5,
				"h_angle": 11.3,
				"sunset": "19:13",
				"dni": 897.34,
				"dewpt": 1.7,
				"snow": 0,
				"uv": 8.82405,
				"precip": 0,
				"wind_dir": 170,
				"sunrise": "04:19",
				"ghi": 843.37,
				"dhi": 116.58,
				"aqi": 50,
				"city_name": "Paris",
				"weather": {
					"icon": "c04d",
					"code": "804",
					"description": "Overcast clouds"
					},
				"datetime": "2019-05-07:13",
				"temp": 16,
				"station": "LFPO",
				"elev_angle": 54.92,
				"app_temp": 16
			}
	],
	"count": 1
}
```

Selecting an XML property

This example will use the following XML sample:

```
<sample>
        <text> This is text data without attributes</text>
        <text_attr desc="This text has an attribute"> This is the text data</text_attr>
        <attribute_only name="No text" desc="No data, only attributes"/>
        <list attr="Attribute">
                <item type="String">Data 1</item>
                <item type="Number">1234</item>
                <item type="Boolean">true</item>
                <single>Single</single>
        </list>
        <array>
                <item><subitem>Data 1</subitem></item>
                <item><subitem>Data 2.1</subitem><subitem>Data 2.2</subitem></item>
        </array>
</sample>
```

Example 1 – Accesing text property in nodes without attributes

In this example, we will try to access the text data inside the <text> node. Since this node doesn't have attributes, we can simply access it directly. The address will be:

sample.text

Example 2 – Accesing attributes in nodes without text

In this example we access attribute data in nodes that don't have any text, like the <attribute_only> empty node. In order to access the attribute data of a node, we need to access that node, and then use the special character '$', followed by accessing the attribute using it's name. Using these rules, the final address will be:

sample.attribute_only.$.name → Access to the name attribute

sample.attribut_only.$.desc → Access to the desc attribute

Example 3 – Accesing attributes and text in the same node

When there are both attributes and text in a node (in our example is the node <text_attr>), we have to use special characters to select which one we want to access. If we want to access an attribute, we use the '$' character, just like in the previous example. If we want to access the text, we use the '_' character. Keeping this in mind, the final addresses are:

sample.text_attr.$.desc → Access to the desc attribute

sample.text_attr._ → Access to the text value

Example 4 – Accessing array nodes:

In this example we will access the text inside the second <item> node that resides inside the <list> node. Since the <list> node has several <item> nodes, we select the one we want by using it's index, starting with 0 for the first element. Also, since the <item> node has both text and attributes, we need to use the '_' character to access the text. The correct address will then be:

sample.list.item[1]._

Example 5 – Accessing nested arrays:

In this final example, we want to access the second <subitem> in the second <item> in the <array> node. Since both <item> and <subitem> are arrays, both need to use an index to select which one we need. Since it doesn't have attributes, we can access it directly without using '_'. Knowing this, the address we need to use is:

sample.array.item[1].subitem[1]