



Device integration

Last updated: 03/03/2026

This content applies to the latest CD version of Cumulocity.

Specifications contained herein are subject to change and these changes will be reported in subsequent versions.

Copyright © 2026 Cumulocity GmbH.

The name Cumulocity GmbH and all Cumulocity GmbH product names are either trademarks or registered trademarks of Cumulocity GmbH and/or its subsidiaries and/or its affiliates and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

This software may include portions of third-party products. Third-party terms are set out in a 3rd-party-licenses file linked to or included with each installation package.

Table of Contents

Table of Contents	3
INTRODUCTION	9
INTEGRATION VIA THIN-EDGE.IO	9
MICROCONTROLLER-BASED DEVICES	9
IOT GATEWAYS AND DATA INTEGRATION	9
LPWAN INTEGRATION	9
AGENT CONCEPTS	9
INTERFACING DEVICES	11
WHAT IS AN AGENT?	11
Protocol translation	11
Model transformation	12
Secure remote communication	12
WHAT AGENT ARCHITECTURE IS SUPPORTED?	12
AGENT LIFECYCLE	13
STARTING THE AGENT	13
SYNCHRONIZING INVENTORY DATA	13
RECEIVING DATA AND COMMANDS FROM APPLICATIONS	14
SENDING SENSOR READINGS, EVENTS, ALARMS AND AUDIT LOGS	14
UPDATING AGENT CONFIGURATION	14
INTEGRATING OF OTHER DATA SOURCES	14
SYSTEM INTEGRATION	14
HOW DOES THE PLATFORM SUPPORT DEVELOPING AGENTS?	14
FRAGMENT LIBRARY	16
INTRODUCTION	16
c8y_SupportedOperations fragments	16
GENERAL CONCEPTS	17
ANNOUNCING CAPABILITIES	17
COMMUNICATING CURRENT STATUS	18
OPERATION HANDLING	18
ERROR HANDLING DURING OPERATION PROCESSING	18
RECOVERING AFTER AGENT CRASH	18
IDEMPOTENT CASES	19
ALARMS	19
RAISING ALARMS	19
CLEARING ALARMS	20
CRITICAL ALARMS	20
CHILD DEVICES	20
ASSIGN CHILD DEVICE TO PARENT DEVICE	21
OPERATING A GATEWAY FOR CHILD DEVICES	21
CONFIGURATION	21
TEXT-BASED CONFIGURATION	21
LEGACY FILE-BASED CONFIGURATION	23
TYPED FILE-BASED CONFIGURATION	25
CONNECTIVITY	27
DEVICE AVAILABILITY	28
AVAILABILITY MONITORING	28
CONNECTION MONITORING	29
DEVICE INFORMATION	29
DEVICE MARKER	29
AGENT MARKER	30
DEVICE RESTART	30
HARDWARE INFORMATION	30
AGENT INFORMATION	31
DEVICE PROFILE	31
FIRMWARE	34

Installed firmware	34
INSTALLING A FIRMWARE IMAGE	35
INSTALLING A FIRMWARE PATCH	35
IDENTITY	36
REST	36
MQTT (SmartREST 2.0)	37
LOGS	37
SETTING SUPPORTED LOGS	37
UPLOADING LOG FILES	38
MANUAL STATUS UPDATE	40
MEASUREMENTS	40
NETWORK	40
Network status	41
SETTING NETWORK CONFIGURATION	42
PLATFORM CAPABILITIES	44
PLATFORM VERSION	44
RELAY	44
RELAYS	44
REMOTE ACCESS	46
REMOTE ACCESS CONNECT	46
SHELL	47
SEND A COMMAND TO A DEVICE	47
SOFTWARE	48
INSTALLED SOFTWARE	48
ADVANCED SOFTWARE MANAGEMENT	51
SERVICES	54
REST API EXAMPLES	54
SERVICE COMMANDS	55
TRACKING	56
TRACKING POSITION HISTORY	56
SETTING THE CURRENT POSITION	57
CORE MQTT	58
INTEGRATION LIFECYCLE	58
STARTUP PHASE	59
CYCLE PHASE	60
MQTT IMPLEMENTATION	60
CONNECTING VIA MQTT	60
SMARTREST PAYLOAD	61
DEVICE HIERARCHIES	62
MQTT FEATURES	62
MQTT RETURN CODES	64
DEBUGGING	65
MQTT BROKER CERTIFICATES	65
MQTT JWT SESSION TOKEN RETRIEVAL	65
MQTT EXAMPLES	66
HELLO MQTT	66
HELLO MQTT C	72
HELLO MQTT JAVA	75
HELLO MQTT JAVA WITH CERTIFICATES	78
HELLO MQTT BROWSER-BASED	80
HELLO MQTT NODE.JS	83
HELLO MQTT PYTHON	85
MQTT SERVICE	89
OVERVIEW	89
ARCHITECTURE	89
MQTT SERVICE COMPARED TO CORE MQTT	91
MQTT PROTOCOL IMPLEMENTATION	92
CONNECTING TO THE SERVICE	92

TOPICS	92
PAYLOAD	94
FEATURES	94
RETURN CODES	95
MQTT 5.0 FEATURES	95
MQTT TLS CERTIFICATES	95
CONNECTING MICROSERVICES AND APPLICATIONS	97
CONNECTING TO THE MESSAGING SERVICE	98
MESSAGE PAYLOADS AND PROPERTIES	100
CONSUMING MESSAGES FROM MQTT DEVICES	101
PUBLISHING MESSAGES TO MQTT DEVICES	102
MESSAGING SERVICE QUOTAS AND LIMITS	104
BEST PRACTICES FOR RELIABLE MESSAGE DELIVERY FROM DEVICES	105
HANDLING MESSAGING SERVICE ERRORS	105
EXAMPLE CLIENT	106
JAVA CLIENT	106
FREQUENTLY ASKED QUESTIONS	106
REST	108
INTEGRATION LIFECYCLE	108
STARTUP PHASE	109
CYCLE PHASE	117
REPLACING A PHYSICAL DEVICE	119
DEVICE AUTHENTICATION	120
JWT SESSION TOKEN RETRIEVAL	120
REST CLIENT EXAMPLES	121
HELLO REST	121
HELLO X509 REST	124
OPC UA	127
INTRODUCTION	127
GATEWAY CONFIGURATION AND REGISTRATION	128
THIN-EDGE.IO	128
MQTT FORWARDING MODE	129
CONFIGURATION PROFILE LOCATION ON THE FILESYSTEM	130
ADDITIONAL CUSTOMIZATIONS	131
LOGGING	135
DELETION OF GATEWAY	136
RUNNING THE GATEWAY	136
ADJUSTING GATEWAY MEMORY SETTINGS	137
PERFORMANCE TUNING FOR LARGE SETUPS	137
REGISTERING THE GATEWAY AS A CUMULOCITY DEVICE	138
GATEWAY DEVICE DETAILS	139
CONNECTING THE GATEWAY TO THE SERVER	140
SECURITY MODES	141
AUTHENTICATION	141
CHILD DEVICES	143
ADDRESS SPACE	144
MONITORING MEASUREMENTS	145
MONITORING ALARMS	147
MONITORING EVENTS	149
DEVICE PROTOCOLS	149
ADDING A NEW DEVICE PROTOCOL	150
ADDING A NEW VARIABLE	150
DEVICE PROTOCOL BEHAVIOR IN MQTT FORWARDING MODE	153
MONITORING EVENTS FOR DEVICE PROTOCOL APPLICATION	154
DATA REPORTING	156
APPLYING CONSTRAINTS	157
BAD STATUSCODE HANDLING	158
REST APIS	158

OPC UA SERVER RESOURCES	159
ADDRESS SPACE RESOURCES	163
DEVICE TYPE RESOURCES	167
OPERATIONS	176
SCANNING THE ADDRESS SPACE	176
READING THE VALUE OF A NODE/NODES	177
READING ALL ATTRIBUTES OF A NODE	178
READING AN ATTRIBUTE	178
READ COMPLEX	180
HISTORIC READ	181
HISTORIC DATA BINARY UPLOAD	182
READ FILE	183
WRITE VALUE	184
WRITE ATTRIBUTE	185
GET METHOD DESCRIPTION	186
CALL METHOD	187
TESTING A DEVICE TYPE AGAINST A NODE ON AN OPC UA SERVER	188
ANALYZING THE SET OF NODES TO WHICH A DEVICE TYPE CAN BE APPLIED (DRY RUN)	189
GET THE CURRENT APPLICATION STATE OF A DEVICE TYPE	191
EXPIRING OPERATIONS	192
OPC UA EVENTS	193
MODEL CHANGE EVENTS	193
TROUBLESHOOTING	193
PERMISSION DENIED ERROR WHEN RUNNING THE GATEWAY JAR FILE ON A LINUX OS	193
UNKNOWN HOST EXCEPTION WHEN RUNNING THE GATEWAY JAR	193
FAILED TO LOAD PROPERTY SOURCE FROM LOCATION WHEN RUNNING THE GATEWAY JAR	193
JAVA.NET.BINDEXCEPTION: ADDRESS ALREADY IN USE	193
CHANGING THE LOG LEVEL FOR TROUBLESHOOTING	194
JAVA MANAGEMENT EXTENSIONS (JMX)	194
OPC UA MAJOR VERSION UPGRADE NOTES	197
UPGRADING FROM 1021 TO 1022 GATEWAY VERSION	197

CLOUD FIELDBUS 198

INTRODUCTION	198
CONNECTING FIELDBUS DEVICES	198
CONNECTING MODBUS/RTU DEVICES	198
CONNECTING MODBUS/TCP DEVICES	199
CONNECTING CAN DEVICES	200
CONNECTING PROFIBUS DEVICES	201
MANAGING FIELDBUS DEVICES	202
COLLECTING MEASUREMENTS	202
MONITORING ALARMS	203
LOGGING EVENTS	203
MONITORING THE DEVICE STATUS	204
MONITORING THE DEVICE STATUS USING THE FIELDBUS DEVICE WIDGET	204
MONITORING THE DEVICE STATUS USING THE SCADA WIDGET	205
PREPARING SVG FILES FOR THE SCADA WIDGET	206
CONFIGURING FIELDBUS DEVICE PROTOCOLS	207
CONFIGURING MODBUS DEVICE PROTOCOLS	208
CONFIGURING CAN BUS DEVICE PROTOCOLS	210
CONFIGURING PROFIBUS DEVICE PROTOCOLS	210
CONFIGURING CANOPEN DEVICE PROTOCOLS	210
EXPORTING AND IMPORTING DEVICE PROTOCOLS	212

LWM2M 214

INTRODUCTION	214
REGISTERING LWM2M DEVICES	215
SINGLE DEVICE REGISTRATION	216
BULK DEVICE REGISTRATION	216
DUPLICATE LWM2M DEVICES	226

DEVICE DELETION	226
LWM2M CONNECTOR DEVICE	226
MIGRATION OF THE LWM2M DEVICES	227
INVALIDATE REGISTRATIONS	227
INVALIDATE REGISTRATIONS BY ENDPOINT	227
INVALIDATE REGISTRATIONS BY LWM2M REGISTRATION ID	228
LWM2M DEVICE PROTOCOLS	228
CREATING LWM2M DEVICE PROTOCOLS	228
ADDING ADDITIONAL FUNCTIONALITIES TO A RESOURCE	229
ALARMS ON DEVICE PROTOCOL MAPPING FAILURES	232
LWM2M DEVICE DETAILS	233
OBJECTS	233
LWM2M CONFIGURATION	235
LWM2M CLIENT AWAKE TIME	236
HANDLING LWM2M SHELL COMMANDS	236
SHELL COMMAND LIFECYCLE	239
ADDING VALIDATION RULES TO RESOURCES	240
COMPLEX RULESETS	241
DEVICE LIFECYCLE EVENTS	241
HANDLING LWM2M POST REGISTRATION ACTIONS	241
DEVICE OPERATIONS HANDLING	242
LWM2M DEVICE FIRMWARE UPDATE (FOTA)	242
FIRMWARE UPDATE STATE MACHINE	243
RESETTING THE STATE MACHINE	243
QUERYING THE DEVICE CONFIGURATION	244
FIRMWARE DELIVERY	244
TRIGGERING THE FIRMWARE UPDATE ON THE DEVICE	244
COMPLETING OF THE FIRMWARE UPDATE PROCESS	244
CANCELING THE FIRMWARE UPDATE PROCESS	245
LPWAN	246
INTRODUCTION	246
LORIIOT LORA	246
INTRODUCTION	246
DEVICE REGISTRATION VIA UPLINK MESSAGE	247
DEVICE REGISTRATION VIA THE CUMULOCITY PLATFORM	250
ASSIGNING THE LORIIOT ADMIN ROLE PERMISSION	254
CREATING DEVICE PROTOCOLS	255
ASSIGNING THE LORIIOT LORA DEVICE PROTOCOL	266
SENDING OPERATIONS	267
UPLINK MESSAGE PROCESSING	268
TROUBLESHOOTING	268
ACTILITY LORA	269
INTRODUCTION	269
CONFIGURING MULTIPLE THINGPARK ACCOUNT CONNECTIONS	269
CREATING DEVICE PROTOCOLS	272
REGISTERING ACTILITY LORA DEVICES	284
DEPROVISIONING LORA DEVICES	288
SENDING OPERATIONS	288
THINGPARK API AVAILABILITY MONITORING	290
UPLINK MESSAGE PROCESSING	290
TROUBLESHOOTING	290
SIGFOX	292
INTRODUCTION	292
MANAGING THE CONNECTIVITY SETTINGS	293
CREATING DEVICE PROTOCOLS	296
REGISTERING SIGFOX DEVICES	307
SENDING OPERATIONS	309
UPLINK MESSAGE PROCESSING	309
TROUBLESHOOTING	309

LPWAN CUSTOM PROTOCOLS	314
INTRODUCTION	314
IMPLEMENTING A CUSTOM CODEC MICROSERVICE	314
USING THE LPWAN CUSTOM CODEC LIBRARY	315
DEPLOYING THE SAMPLE CODEC MICROSERVICE	317

INTRODUCTION

Cumulocity offers a wide range of functionality for interfacing IoT devices and other IoT-related data sources with the Cumulocity platform.

The integration approach depends on the device capabilities and use cases.

INTEGRATION VIA THIN-EDGE.IO

We recommend integrating devices via [thin-edge.io](#), an open-source, cloud-agnostic edge framework optimized for lightweight IoT devices. thin-edge.io can be slimmed down to run with less than 1 MB footprint, making it suitable even for constrained devices while supporting both x86_64 and ARM-based processor architectures.

Advantages of using thin-edge.io:

- **Native SmartREST efficiency:** The agent automatically translates simple local JSON messages into Cumulocity's highly efficient SmartREST protocol, significantly reducing bandwidth.
- **Zero-code device management:** Get immediate access to all device management features - including software management, configuration updates, log retrieval, and remote access.
- **Automatic child device routing:** Acting as a gateway requires no extra logic; simply publishing data with a child ID causes thin-edge.io to automatically register the external devices (such as a sensor) in Cumulocity's inventory, and route the data to the correct representation of the child device in Cumulocity.
- **Modular extensibility:** The architecture is designed around plugins, allowing you to extend functionality without having to recompile the core agent.
- **Language-agnostic decoupling:** Because thin-edge.io uses a local MQTT bus for communication, your application logic can be written in any language and remains completely isolated from the connectivity logic.
- **Automated certificate lifecycle:** The built-in CLI tools handle the generation, signing, uploading, and rotation of X.509 security certificates.

See the tutorial [Getting started with thin-edge.io](#) for an easy-to-follow and hands-on example.

MICROCONTROLLER-BASED DEVICES

For highly constrained devices with microcontrollers that cannot run thin-edge.io, you can integrate directly via the Core [MQTT](#) and [REST](#) APIs along with [SmartREST](#) for efficient communication. These can be implemented using available MQTT client libraries such as [Eclipse Paho](#). For standard-compliant device management, the [LWM2M](#) protocol is also supported.

IOT GATEWAYS AND DATA INTEGRATION

Not all devices are directly connected to the internet. In such cases, IoT gateways act as intermediaries, collecting data from devices and forwarding it to Cumulocity. Several data integration options are available:

- **OPC UA:** Industrial automation protocol for connecting PLCs and industrial equipment.
- **MQTT Service:** Flexible MQTT endpoint allowing user-provided microservices to map between custom device payloads and the Cumulocity data model.
- **thin-edge.io protocol drivers:** Extend thin-edge.io with custom protocol support for proprietary or specialized device protocols.
- **Partner gateways:** Use pre-integrated partner gateways that bring support for the required protocol. Explore certified partner devices in the [Device Partner Portal](#).

LPWAN INTEGRATION

Low-Power Wide-Area-Network (LPWAN) technologies are critical for use cases requiring devices to:

- Run on a single battery for years at a very low cost.
- Only transmit small amounts of data intermittently.
- Be positioned in distributed or hard-to-reach locations.

Cumulocity provides dedicated integrations for various LPWAN technologies including LoRa and Sigfox. Refer to [LoRa Actility](#), [LoRa LORIoT](#), and [Sigfox](#) for details. For cellular devices, [LWM2M](#) can be used.

AGENT CONCEPTS

A device agent is a piece of software that runs locally on a device or gateway. Its primary purpose is to act as the intermediary between the device's physical hardware and the cloud platform. To learn more about the general concept of agents being used for interfacing IoT devices and data sources with Cumulocity, refer to [Interfacing devices](#).

INTERFACING DEVICES

To interface IoT data sources such as devices and external IT systems, Cumulocity provides agents. Agents are software components that enable a centralized perspective on all aspects and central operation of the IoT network.

This section explains concepts relevant for interfacing IoT devices and other IoT-related data sources with Cumulocity.

To interface these systems with Cumulocity, a driver software called *agent* is required. We first describe the main tasks of an agent and discuss the structural options for agents later. We will walk step by step through the tasks of an agent. Finally, we discuss the usage of agents for acquiring data from other data sources such as a tenant's IT system.

RELATED TOPICS

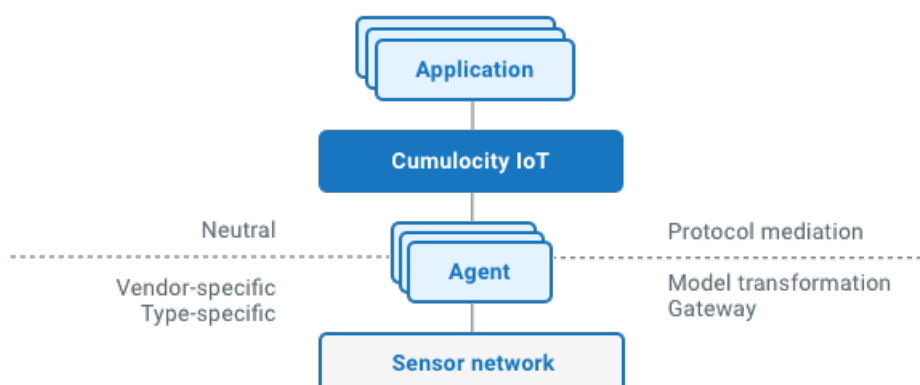
- [Getting started > Technical concepts > Cumulocity's domain model](#) for understanding the data structures exchanged between agents and the Cumulocity core.
- [Device management & connectivity > Device integration](#) for understanding in detail how to develop agent software using the REST or MQTT protocols.
- [REST implementation](#) in the Cumulocity OpenAPI Specification, for a detailed specification of the interfaces between agents and the Cumulocity core.

WHAT IS AN AGENT?

Internet of Things (IoT) devices come with a wide variety of protocols, parameters and network connectivity options. Protocols of devices range from low-level serial links to full-blown IT protocols such as web services. Today's IoT standards rarely define exactly how to access particular readings of particular sensors or manipulate particular controls. Devices can be connected through mobile networks and gateways.

To shield machine-to-machine applications from this numbers of access options, Cumulocity uses *agents*. An agent is a function that complies with three duties for a specific vendor and type of devices:

- It translates the device-specific interface protocol into a single reference protocol.
- It translates the specific domain model of the device into a reference domain model.
- It enables secure remote communication in various network architectures.



Protocol translation

The configuration of parameters, readings, events and other information is either send to an agent ("push") or queried by an agent ("pull")

through a device-specific protocol on one side. The agent will convert these messages into the protocol that Cumulocity requires. It will also receive device control commands from Cumulocity ("switch off that relay") and translate these into a kind of protocol the device requires.

Cumulocity uses a simple and secure reference protocol based on REST (that is, HTTPS) and JSON, which can be used for a wide variety of programming environments down to small embedded systems. To support real-time scenarios, the protocol is designed around a "push" model, that is, data is sent as soon as it is available.

Model transformation

The configuration parameters, readings, events, they all have their device-specific name (and possibly units). An agent for a particular device will transform this device-specific model to the Cumulocity reference model. For example, an electricity meter provides the main reading as a parameter "Received Wh", so the agent will transform this reading into a reference "Total active energy" in kWh.

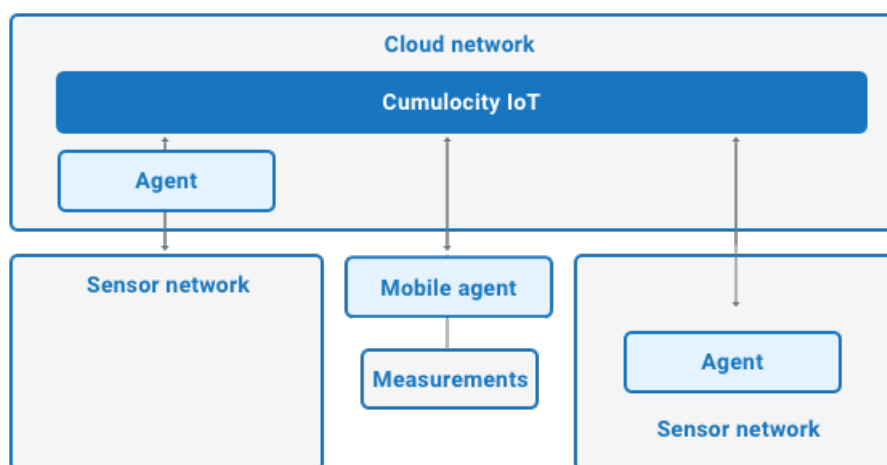
Secure remote communication

Devices can provide a protocol that is unsuitable for secure remote communication, in particular in public cloud environments. The protocol only supports local networking and does not pass through firewalls and proxies and it can contain sensitive data in clear text form. To avoid security issues like these, an agent can be co-located to the device and provide a secure, internet-enabled link to the remote device via Cumulocity.

To summarize the benefits of the agent concept: Agents enable IoT applications to securely interface with any type of remote IoT device and without imposing any mandatory system requirement on the device itself. They drastically simplify developing IoT applications by shielding the applications from the variety of IoT devices and protocols.

WHAT AGENT ARCHITECTURE IS SUPPORTED?

Agents can be deployed in various ways, as illustrated in the picture below. We distinguish two main variants: **server-side agents** and **device-side agents**.



Server-side agents are run in a cloud, hosted on Cumulocity as microservices or managed by yourself in your own cloud. Devices connect to server-side agents using their device-specific protocol. This option is mainly chosen when one or more of the following complies:

- The device is "closed", that means, it is not programmable and supports only one particular, pre-defined protocol to communicate with the outside world.
- The protocol on the device is secure and internet-enabled, that is, the device connects to the cloud and not vice-versa.

Device-side agents run on a device in the sensor network. These devices can be routers, mobile phones or modems. The agents perform in any kind of run-time environment the device supports, ranging from the very battery- and memory-consuming embedded microcontrollers to minicomputers running Embedded Linux. The agents will directly query connected sensors and manipulate connected controls. This usually results in a simpler architecture than server-side agents.

AGENT LIFECYCLE

STARTING THE AGENT

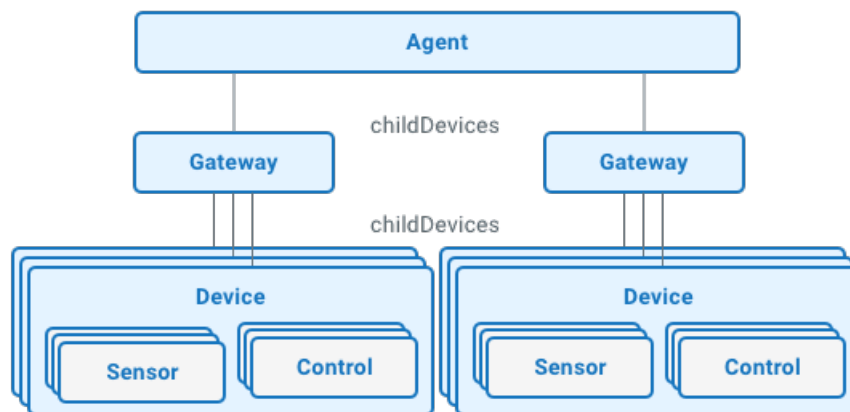
Server-side agents run continuously in the cloud, accepting connections from the device types that they support. Device-side agents run on the device and are started along with other device software when the device is powered on.

Both types of agents are pre-configured with a fixed platform endpoint URL. Using this platform endpoint URL, credentials for each connected device are acquired. These credentials enable the device to connect to a tenant in Cumulocity and to send data to the tenant as well as to accept operations from the tenant.

After starting, the agent will synchronize the inventory with the sensor subnetwork that the agent is responsible for.

SYNCHRONIZING INVENTORY DATA

To understand inventory synchronization, remember the communication hierarchy described in [Cumulocity's domain model](#). In the inventory, agents are located at the roots of the communication hierarchy. Below each agent, the topology of the subnetwork that the agent manages is reflected. This topology exists in the real network as well as in snapshot form in the inventory. It may change in the real network, and these changes must be reflected in the inventory.



Inventory synchronization is a two step procedure: The first step is to query the agent's entry from the inventory and to create it in the network. The second step is then to discover the subnetwork and synchronize it with the inventory based on the queried agent's entry.

The first step provides the option to pass configuration information to an agent as part of the agent entry into the network. This configuration information is determined by the type of agent and the connected devices. It contains, for example, polling intervals for measurements. It can also assign subnetwork tasks to the agent in case the agent cannot automatically discover its associated network.

For example, an agent installed on a mobile phone can discover a connected bluetooth heart monitor without further configuration. An agent installed on a local IP network can run a discovery procedure on a local network. Unlike a Multispeak agent requires the URL of a Multispeak server and credentials to be able to discover connected smart meters.

To keep inventory information up-to-date and maintain a centralized view on devices, two mechanisms are used:

- A regular inventory upload, which runs first when the agent is started and will be repeated periodically.
- A propagation of individual changes occurring while the agent runs.

The need for a regular inventory upload depends on the particular device protocol, which possibly supports change notifications. Assume, for example, that a device is operated locally through controls on the device or using a local device manager software. If the device protocol does not propagate these changes, they can only be discovered by a regular query. Another example would be the assumption that new devices can only be discovered by scanning a network address range in the sensor network regularly. This must be executed by an agent.

It is important to know that the device agent is assuming data ownership of configuration properties or device topology data and therefore modifies or overwrites this data accordingly.

RECEIVING DATA AND COMMANDS FROM APPLICATIONS

Now that the topology is established in the inventory, the devices are visible and operable from IoT applications. As described in the device control section of [Cumulocity's domain model](#), IoT applications can send operations to devices, which are queued in the core. The agent must query the core for operations intended for its devices.

If an operation was sent to an agent's device, the agent will translate the operation into the device-specific representation. For example, a Multispeak agent would translate an operation to set the state of a switch to a SOAP "initiateConnectDisconnect" request for an electricity meter. The translated operation is then sent to the device.

Finally, the agent acknowledges the execution of the operation and it would update the state of the switch in the inventory.

SENDING SENSOR READINGS, EVENTS, ALARMS AND AUDIT LOGS

Besides remote control of devices, the other main task of agents is to transmit data from sensors. This data can vary as outlined in the domain model sector:

- **Measurements** are produced by reading sensor values. In some cases, this data is read in static intervals and sent to the platform (for example temperature sensors or electrical meters). In other cases, the data is read on demand or at irregular intervals (for example health devices such as weight scales). Regardless what kind of protocol the device supports, the agent is responsible for converting it into a "push" protocol by uploading data to Cumulocity.
- **Events** that must be processed in realtime by IoT applications, for example, notifications from a motion detector or transactions from a vending machine.
- **Alarms** are events that require human intervention, for example, tamper events sent by an electrical meter.
- **Audit logs** are events that are recorded for risk management purposes, for example, login failures.

UPDATING AGENT CONFIGURATION

The agent configuration may need to be changed during run-time. For example, a new gateway to a sensor network is installed and the address and credentials for accessing that gateway must be sent to the agent. This is performed by sending a device control request targeted to the agent itself. After processing the configuration, the agent will publish changes within the device network.

INTEGRATING OF OTHER DATA SOURCES

SYSTEM INTEGRATION

Enterprises offering IoT-enabled services typically run other IT systems that supply important information on IoT assets and devices. Examples of those systems are:

- Asset management systems that provide additional information about the available devices and their location.
- Customer relationship management systems that provide information about the customer as device owner.
- Workforce management systems that provide information on the maintenance status of devices.

Technically, developing and running an agent for system integration is not different from an agent for device integration. However, the subset of data owned by the systems is different. Agents for device integration own the device hierarchy and device configuration information. Agents for system integration provide additional information for devices and own parts of the asset hierarchy. Together, they contribute to the device information stored in the inventory to provide a centralized view on everything related to the assets and devices that are relevant for the IoT service.

HOW DOES THE PLATFORM SUPPORT DEVELOPING AGENTS?

Cumulocity supports agent development on three different levels:

- It supports open source agents and drivers, for example [Cumulocity Device Management Reference Agent](#) or [thin-edge.io](#).
- Client libraries for major runtime environments such as C/C++, Java and Lua, again as open source in [github.com](#).
- Technology-neutral [REST APIs](#) for other runtime environments.

[🏠](#) > [Device integration](#) > [Interfacing devices](#)

FRAGMENT LIBRARY

INTRODUCTION

The device integrator library defines the data structures that are used in Cumulocity for device management activities like, for example, software management and configuration management. The data structures are expressed as fragments that can be used inside managed objects, operations and other resources. More information on the fragment concept can be found in [Cumulocity domain model](#).

In the following section you will find descriptions of the most important functionalities of a device, its managed objects and all its corresponding fragments to them. We will explain the relationship between the Cumulocity UI, the device object managed in our databases and what is being communicated to and from the device itself.

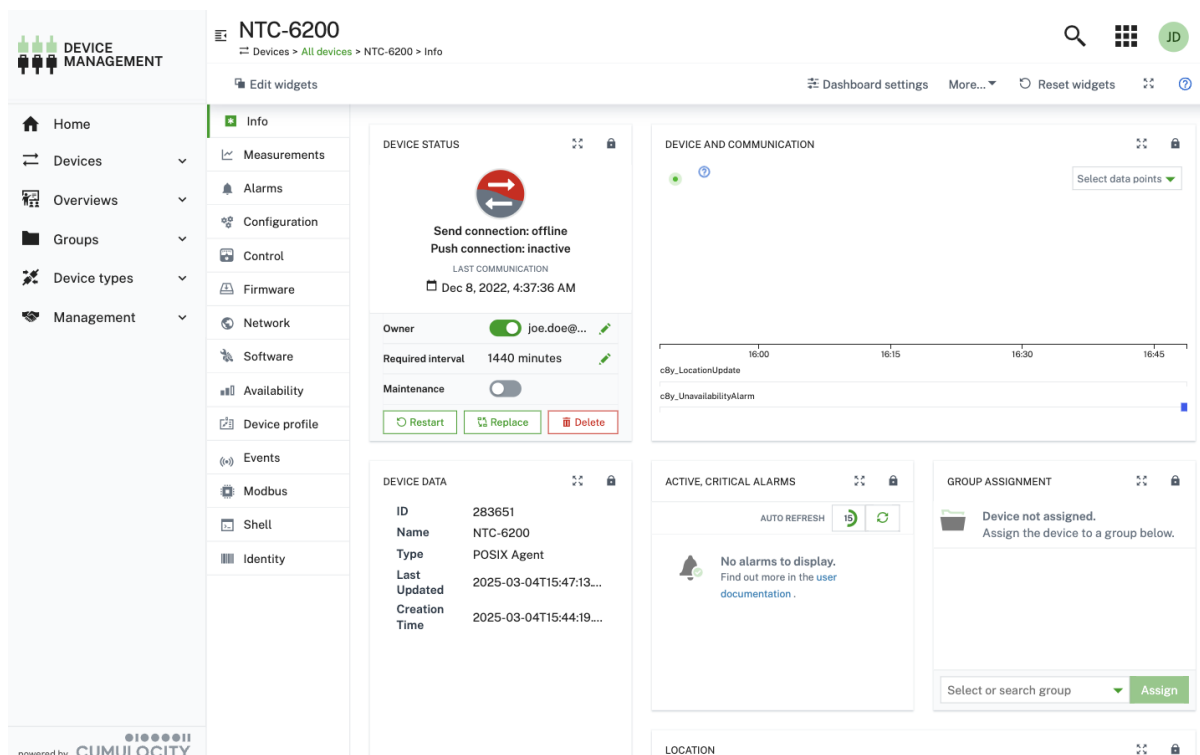
If you are interested in details on exposing the Cumulocity's functionalities through our REST API, see the [Cumulocity OpenAPI Specification](#) for further information.

Moreover, see [SmartREST](#) for more information on SmartREST and a complete list of all SmartREST templates mentioned throughout the following sections.

To start with device management, open the **All devices** tab in the **Devices** menu of the Device Management application. Click on a device in the list to open the device details of this particular device. You will see various tabs and particular information on each of them.

INFO

For a detailed explanation of each tab and its related configuration via the UI, see also [Device Management application > Viewing device details](#).



This list can be manipulated through the device fragments, that means, which tabs are shown depends on the capability the device supports. This is mainly operated by one fragment called `c8y_SupportedOperations`. Based on what is put in the array of this fragment, functionality such as tabs, buttons, and so on are enabled. For example if the `c8y_SupportedOperations` fragment contains `c8y_Firmware`, the firmware tab will be visible in the **Device details** page and the device can manage firmware objects.

[c8y_SupportedOperations fragments](#)

The following fragments can be added to the `c8y_SupportedOperations` fragment:

Fragment	Definition
<code>c8y_Availability</code>	Holds information about the device's status and its availability
<code>c8y_Command</code>	Allows the user to carry out interactive sessions with a device
<code>c8y_Configuration</code>	Contains the complete configuration state of the device including all control characters
<code>c8y_ConfigurationDump</code>	Permits managing binary configuration files of the device
<code>c8y_DeviceProfile</code>	Enables device profile functionality for a device
<code>c8y_DownloadConfigFile</code>	Permits the download of configuration files as binaries
<code>c8y_Firmware</code>	Contains information about a device's firmware
<code>c8y_Hardware</code>	Contains basic hardware information for a device, such as make and serial number
<code>c8y_LogfileRequest</code>	Requests a device to send a log file and view the log file in the log viewer
<code>c8y_MeasurementRequestOperation</code>	Displays a "Get measurements" action in the device context menu that sends an operation for triggering a manual status update of a device.
<code>c8y_Mobile</code>	Holds basic connectivity-related information, such as the equipment identifier of the modem (IMEI) in the device or the SIM card (for example ICCID)
<code>c8y_Network</code>	Sends data to the Network tab in the Device Management application and displays the network information
<code>c8y_Position</code>	Reports the geographical location of an asset in terms of latitude, longitude and altitude
<code>c8y_Profile</code>	Announces the target profile
<code>c8y_Restart</code>	Restarts a device
<code>c8y_Relay</code>	Enables switching device relay
<code>c8y_RelayArray</code>	Enables switching device relays in the array
<code>c8y_SendConfiguration</code>	Allows reloading a configuration through the user interface
<code>c8y_ServiceCommand</code>	Announces a service capability to receive commands
<code>c8y_SoftwareList</code>	Contains the entire list of software that is installed on the device
<code>c8y_SoftwareUpdate</code>	Contains a list of software to be installed or uninstalled
<code>c8y_UploadConfigFile</code>	Permits the upload of configuration files as binaries

GENERAL CONCEPTS

ANNOUNCING CAPABILITIES

Devices may announce their supported capabilities using the `c8y_SupportedOperations` fragment in their own managed object. The fragment itself is an array of strings. It may contain built-in operations with their meaning or custom operations for specific use cases, which are described in the following sections.

```
PUT /inventory/managedObjects/<deviceId>
```

```
{
  "c8y_SupportedOperations": [
    "c8y_Restart",
    "c8y_Network"
  ]
}
```

Field	Data type	Mandatory	Details
c8y_SupportedOperations	array	Yes	Array of strings of the supported operations

Some capabilities also introduce their own `c8y_Supported<...>` fragments with a similar concept. These fragments allow devices to announce the ability to handle certain types of log files or configuration.

SmartREST example

The 114 static template is available for devices to announce their supported operations:

```
114,c8y_Restart,c8y_Configuration,c8y_SoftwareList
```

COMMUNICATING CURRENT STATUS

Devices are responsible for communicating their current status to Cumulocity. The status is usually communicated in the device's own managed object. Cumulocity provide specific fragments for each capability. The device must update this data whenever it detects a change to its local state.

In practice this usually means that a device should publish its local state concerning all of its supported capabilities during startup, when requested to change its local state, and whenever any external change has been detected.

OPERATION HANDLING

Operations are always created with status PENDING. Devices are responsible for moving operations along into different statuses in their lifecycle. Before beginning to process an operation the device agent must update its status to EXECUTING. After processing is completed the device must set the operation status to SUCCESSFUL or FAILED depending on the outcome.

SmartREST 2.0

Cumulocity provides the static templates 501, 502, and 503 to manipulate the operation status. These templates take the operation type as input parameter and always update the oldest operation in the preceding status. We recommend you to let all operations be handled sequentially in the order they arrive at the device.

Devices can find their operation IDs by querying the Device Control API, by subscribing to the operation JSON topic, or by using a custom response template which includes the IDs, that is, template 504, 505, or 506, which enable setting the status of operations with a known ID.

ERROR HANDLING DURING OPERATION PROCESSING

If any error occurs during the processing of an operation the device must set the operation status to FAILED and provide a failure reason as descriptive as possible. This includes any unexpected or expected error conditions that prevent the operation to be fully completed and as expected. Even if only one step in an operation with multiple distinct steps fails, the entire operation must be considered as FAILED.

It is up to the device and its use case whether it should roll back any local state changes that happened before the error occurred. If any change of state remains after an operation failed the device must communicate this changed state with Cumulocity.

Handling of unknown operations

Future versions of Cumulocity may include new operation types. To ensure compatibility, devices should safely ignore any operations they do not recognize. When a device receives an unknown operation code, it should not respond or take any action. The operation will remain in a pending state, which is normal and does not indicate a problem.

RECOVERING AFTER AGENT CRASH

After an unexpected restart a device must cleanly recover its status. This includes all status parameters communicated with the platform and all ongoing operations. Recovering the status can be done by updating all values in the cloud with the current values on the device. Recovering ongoing operations is more difficult. Devices are expected to keep track of all operations they moved to status EXECUTING. Typically devices keep information of longer-running operations in a persistent storage so that they can be resumed. In unexpected shutdown or crash scenarios this may not always be possible. In this case the device may cancel all ongoing operations to reset its own status.

```
GET /devicecontrol/operations?deviceId=<deviceId>&status=EXECUTING
```

```
{
  "operations": [
    {
      "creationTime": "2023-06-25T14:53:52.395Z",
      "deviceId": "123",
      "id": "101",
      "status": "EXECUTING",
      "c8y_Restart": {}
    },
    {
      "creationTime": "2023-06-25T14:57:29.089Z",
      "deviceId": "123",
      "id": "102",
      "status": "EXECUTING",
      "c8y_SendConfiguration": {}
    }
  ]
}
```

Then it must change the statuses one by one:

```
PUT /devicecontrol/operations/<operationId>
```

```
{
  "status": "FAILED"
}
```

SmartREST 2.0

Alternatively the static template 507 may be used. The template changes the status from EXECUTING to FAILED for all operations of the given type or for all types.

IDEMPOTENT CASES

In cases where a device receives an operation that requests a state that is already present, it is up to the device how the operation should be handled. This may for example be the case when a device is requested to install a software package that is already present in the requested version. Typically there are three different ways of handling such cases in device agents: skip, execute, or fail. In case of the mentioned software package that is already installed the following options could be selected:

1. Consider the package as already installed and skip its installation because the requested state is already present
2. Execute the operation as normal including re-installing the package
3. Fail the operation because the requested state may indicate that the command was created under false preconditions

The ideal option depends on the use case and the concrete operation. Regardless of which option is selected the device must ensure that its local state and the one communicated to Cumulocity remains consistent.

ALARMS

The **Alarm** tab is always shown for all devices. Its content is filled by alarm statuses reported by the device and other sources like analytics or smart rules. Devices raise alarms in Cumulocity as they occur. Once the alarm status was resolved the device must also update the status of its created alarm to CLEARED.

RAISING ALARMS

A device may raise an alarm at any time. Typically alarms are used to communicate problem statuses in the devices environment.

```
POST /alarm/alarms
```

```
{
  "source": {
    "id": "4801"
  },
  "type": "c8y_TemperatureAlarm",
  "text": "CPU temperature too high",
  "severity": "MAJOR",
  "time": "2021-10-07T12:00:00.000Z"
}
```

Field	Data type	Mandatory	Details
source	object	Yes	The ID of the device
type	string	Yes	Type of the alarm
text	string	Yes	Alarm text, describing the alarm status
severity	string	Yes	Alarm severity
time	string	Yes	Time of alarm occurrence

In addition to the required parameters above, the device may also include custom fragments with more details about the alarm status into the alarm.

SmartREST example

Cumulocity provides several static SmartREST templates for basic alarm management for device. They can be found with message IDs between 301 and 304 for different severities:

```
302,c8y_TemperatureAlarm,"CPU temperature too high"
```

CLEARING ALARMS

When a device detects that the local alarm status was resolved it must clear the alarm. This is done by updating the alarm status to CLEARED.

```
PUT /alarm/alarms/<alarmId>
```

```
{
  "status": "CLEARED"
}
```

Field	Data type	Mandatory	Details
status	string	Yes	The new alarm status

SmartREST example

The 306 static template is provided to clear an active alarm of a specified type:

```
306,c8y_TemperatureAlarm
```

CRITICAL ALARMS

When a device raises an alarm with the severity CRITICAL, the device is considered unavailable for the duration this alarm stays active. The aggregated availability overview in the **Service monitoring** tab will reflect this time as offline.

Devices should use the severity "CRITICAL alarm" only for alarm statuses that impact the device's ability to fulfill its use case.

CHILD DEVICES

The **Child Devices** tab shows a list of all child devices. It will be available only if the device has any child devices assigned to it.

We recommend to include the `c8y_IsDevice` fragment in all child devices. Accordingly, all aspects of devices are also applicable to child devices, for

example, querying and filtering of child devices. Moreover, child devices are also shown in the device lists (such as the **All devices** list, and in groups and during the creation of bulk operations).

ASSIGN CHILD DEVICE TO PARENT DEVICE

In order to link a device the parent device must post to its inventory API the following request containing the ID of the child device.

```
POST /inventory/managedObjects/<deviceId>/childDevices
Content-Type: application/vnd.com.nsn.cumulocity.managedobject+json
```

```
{
  "name": "New child object",
  "c8y_IsDevice": {}
}
```

Field	DataType	Mandatory	Details
managedObject.id	string	Yes	ID of the child device to link

SmartREST example

To add a child device to an existing device you must connect the connected device and call the child create template:

```
101,uniqueChildId,myChildDevice,myChildType
```

OPERATING A GATEWAY FOR CHILD DEVICES

Using the agent marker fragment `com_cumulocity_model_Agent` on the parent device but not on child devices effectively declares the device as a connected gateway for its children. The children are not directly connected to Cumulocity but send and receive data through the device and its integration.

In this case operations for the child devices are delivered to the connected parent device. The parent device then must determine the addressed child device based on the included device ID or other information. Then the command must be forwarded to the correct child.

The built-in static SmartREST response templates of Cumulocity always include a device identifier as first parameter to determine the targeted child device. Here is an example of the 510 static response template for the `c8y_Restart` operation with the device identifier highlighted.

```
510,DeviceSerial
```

Custom response templates also contain the targeted device's external ID as first parameter. We recommend you to implement a similar mechanism there as well.

CONFIGURATION

The **Configuration** tab allows three different formats for device configuration:

- Text-based configuration
- Legacy file-based configuration
- Typed file-based configuration

They all follow a similar concept where the device may upload its current configuration to the platform and users may install a new configuration on the device. This tab appears for devices when they announce support for any of the available formats.

TEXT-BASED CONFIGURATION

The most basic form of configuration is a simple text-based configuration. Here the configuration is stored and transferred directly as string. We recommend you to use this form for small human readable configuration files only, for example, for microcontroller-based devices.

The current configuration state of the device is communicated with the `c8y_Configuration` fragment in the device's own managed object. It contains the complete configuration including all control characters as a string. Special care must be taken that encoding is performed properly. Cumulocity supports UTF-8 characters, additionally escaping according to the [JSON specification](#) for JSON payloads, or the [SmartREST specification](#) for SmartREST payloads may be required.

We recommend you to upload the current configuration only on demand to save transfer data volume and device resources. There are specific operations designed to trigger a device to upload its current configuration to the platform documented below.

```
PUT /inventory/managedObjects/<deviceId>
```

```
{
  "c8y_Configuration": {
    "config": "c8y.url.http=https://management.cumulocity.com\nc8y.url.mqtt=mqtt.cumulocity.com\n"
  }
}
```

Name	Type	Mandatory	Description
config	string	No	Complete configuration text to be applied by the device

Upload current text configuration

For devices that include `c8y_SendConfiguration` in their `c8y_SupportedOperations` the **Configuration** tab offers a button to trigger a configuration upload from the device to Cumulocity. When the button is clicked a `c8y_SendConfiguration` is created.

```
{
  "c8y_SendConfiguration": {}
}
```

Name	Type	Mandatory	Description
c8y_SendConfiguration	object	Yes	Send configuration marker object designating the operation as a command to trigger the device to upload its configuration

The device is expected to perform the following actions:

1. Set operation status to EXECUTING
2. Upload the current configuration to its own managed object using the `c8y_Configuration` fragment
3. Set operation status to SUCCESSFUL

SmartREST example

There is no built-in static response template available for the `c8y_SendConfiguration` operation. Devices must create a custom template to implement this capability. Here is an example how such a template and its use could work.

Template creation:

```
11,100,,c8y_SendConfiguration,deviceId
```

Receiving the operation:

1. Receive the `c8y_SendConfiguration` operation using the custom template created above
`100,DeviceSerial,4801`
2. Set operation status to EXECUTING `501,c8y_SendConfiguration`
3. Upload the current configuration state using the 113 static template
`113,"c8y.url.http=https://management.cumulocity.com\nc8y.url.mqtt=mqtt.cumulocity.com\n"`
4. Set operation status to SUCCESSFUL `503,c8y_SendConfiguration`

Install text configuration

Devices that support installing configuration can communicate this by adding `c8y_Configuration` to their `c8y_SupportedOperations`. Then the **Configuration** tab will offer a button to send a user configured configuration to the device. This action consequently creates a `c8y_Configuration` operation with the same fragment signature as found in the device's managed object.

The device is expected to perform the following actions:

1. Set operation status to EXECUTING
2. Install and apply configuration as included in the nested config property with the `c8y_Configuration` fragment
3. Update the `c8y_Configuration` fragment in the device's managed object
4. Set operation status to SUCCESSFUL

SmartREST example

The 513 static response template is available to receive `c8y_Configuration` operations:

1. Receive `c8y_Configuration` operation
`513,DeviceSerial,"c8y.url.http=https://management.cumulocity.com\nc8y.url.mqtt=mqtt.cumulocity.com\n"`
2. Set operation status to EXECUTING
`501,c8y_Configuration`
3. Install and apply configuration as included
4. Update the `c8y_Configuration` fragment

```
113,"c8y.url.http=https://management.cumulocity.com\nc8y.url.mqtt=mqtt.cumulocity.com\n"
5. Set operation status to SUCCESSFUL
503,c8y_Configuration
```

LEGACY FILE-BASED CONFIGURATION

Devices that want to manage configuration as files can achieve a basic form using legacy file-based configuration. For new device integrations we recommend you to implement typed file-based configuration instead because it is more versatile.

This approach stores and transfers configuration as binary files.

INFO

This mechanism only works with the internal Cumulocity repository. Be aware that configurations with external URLs will not be supported if a device only supports legacy configuration. The ID which is stored in the device managed object (an example can be seen below) always refers to an internal binary saved in the inventory.

Upload current legacy configuration

Devices may signal their support for uploading their current configuration to Cumulocity by adding `c8y_UploadConfigFile` to their `c8y_SupportedOperations`. This enables a **Get snapshot from device** button in the **Configuration** tab. Clicking it generates a `c8y_UploadConfigFile` operation for the device.

```
{
  "c8y_UploadConfigFile": {}
}
```

Name	Type	Mandatory	Description
c8y_UploadConfigFile	object	Yes	Upload configuration file marker object designating the operation as a command to trigger the device to upload its configuration file

When uploading its configuration, the device must first upload the configuration file into Cumulocity inventory binaries, and then create a configuration repository entry as a managed object of type `c8y_ConfigurationDump` in the inventory. This object must then contain a link to the just uploaded file. We recommend you to create this entry with an easily recognizable name and description that allows users to find the desired configuration in the repository.

```
POST /inventory/managedObjects
```

```
{
  "name": "myDevice configuration",
  "description": "Uploaded by myDevice on 2021-09-15T12:00:00+0200",
  "url": "https://demos.cumulocity.com/inventory/binaries/154702",
  "type": "c8y_ConfigurationDump"
}
```

Name	Type	Mandatory	Description
name	string	Yes	Name of the configuration
description	string	No	Description of the configuration
url	string	Yes	URL where the configuration file was uploaded to
type	string	Yes	Type of the configuration repository entry object; must always be "c8y_ConfigurationDump"

The device is expected to perform the following actions:

1. Set operation status to EXECUTING
2. Upload the configuration file into Cumulocity inventory binaries
3. Create a configuration repository entry
4. Set the operation status to SUCCESSFUL

SmartREST example

The 520 static response template is available for this functionality:

1. Receive `c8y_UploadConfigFile` operation
`520,DeviceSerial`
2. Set operation status to EXECUTING
`501,c8y_UploadConfigFile`
3. Upload configuration to inventory binaries API using REST
4. Create configuration repository entry in inventory using REST
5. Set operation status to SUCCESSFUL
`503,c8y_UploadConfigFile`

Install legacy configuration

Devices that are capable of installing configuration remotely can announce this by adding `c8y_DownloadConfigFile` to their `c8y_SupportedOperations`. Then the **Configuration** tab offers a **Send configuration to device** button. When clicked, a `c8y_DownloadConfigFile` operation is created for the device.

```
{
  "c8y_DownloadConfigFile": {
    "url": "https://demos.cumulocity.com/inventory/binaries/9100",
    "c8y_ConfigurationDump": {
      "id": "9200"
    }
  }
}
```

Name	Type	Mandatory	Description
url	string	Yes	URL where the configuration file should be obtained from
c8y_ConfigurationDump	object	Yes	Configuration dump reference object containing the ID of the configuration repository entry object

After downloading the configuration from the specified URL and installing it, the device must reference its currently installed configuration in its own managed object. This is done by transferring the nested `c8y_ConfigurationDump` fragment entirely into the device's own managed object.

```
PUT /inventory/managedObjects/<deviceId>
```

```
{
  "c8y_ConfigurationDump": {
    "id": "9200"
  }
}
```

Name	Type	Mandatory	Description
id	string	Yes	ID of the referenced configuration object

The device is expected to perform the following actions:

1. Set operation status to EXECUTING
2. Download the configuration file from the specified URL
3. Install the configuration file
4. Update the currently installed configuration dump in the device's managed object
5. Set the operation status to SUCCESSFUL

SmartREST example

The 521 static response template is available for this functionality:

1. Receive `c8y_DownloadConfigFile` operation
`521,DeviceSerial,https://demos.cumulocity.com/inventory/binaries/9100`
2. Set operation status to EXECUTING
`501,c8y_DownloadConfigFile`
3. Download configuration from the URL specified
4. Install the configuration file
5. Set operation status to SUCCESSFUL and set the currently installed `c8y_ConfigurationDump` fragment implicitly
`503,c8y_DownloadConfigFile`

TYPED FILE-BASED CONFIGURATION

The most versatile way of managing device configuration is typed file-based configuration. Here a device can manage multiple configuration files at the same time. Typed file configuration is activated for a device by adding the `c8y_SupportedConfiguration` fragment to the device's own managed object.

```
PUT /inventory/managedObjects/<deviceId>
```

```
{
  "c8y_SupportedConfigurations": [
    "agent_conf",
    "ssh_conf"
  ]
}
```

Name	Type	Mandatory	Description
c8y_SupportedConfigurations	array	Yes	Array of strings of the supported configuration for this device

Cumulocity does not validate or further process configuration types. From a platform perspective they are simple strings. Associating these type strings to configuration files is responsibility of the device agent.

SmartREST example

The `c8y_SupportedConfiguration` fragment can be uploaded using the static template 119:

```
119,agent_conf,ssh_conf
```

Upload current configuration file

Similarly to legacy configuration, uploading typed configuration is announced by adding the `c8y_UploadConfigFile` to the `c8y_SupportedOperations`. In this case clicking the button creates a very similar `c8y_UploadConfigFile` operation with the targeted configuration type as additional parameter.

```
{
  "c8y_UploadConfigFile": {
    "type": "agent_conf"
  }
}
```

Name	Type	Mandatory	Description
type	string	Yes	Type of the configuration to upload

Then the device must create an event with the type equal to the configuration type. Cumulocity uses events here instead of the inventory like in legacy file-based config because events are automatically associated to the device and old events (and their binary attachments) included can be automatically cleaned up using retention rules.

```
POST /event/events
```

```
{
  "source": {
    "id": "4801"
  },
  "type": "agent_conf",
  "time": "2021-09-15T15:57:41.311Z",
  "text": "agent_conf upload requested"
}
```

Name	Type	Mandatory	Description
source	object	Yes	ID of the device object
type	string	Yes	Type of the configuration uploaded
time	string	Yes	ISO datetime when the configuration was uploaded

Name	Type	Mandatory	Description
text	string	Yes	Label text for the configuration

In order to attach the configuration file to the just uploaded event, the [Event binaries API](#) should be used. The file is attached using a *multipart/form-data* request.

```
POST /event/events/<eventId>/binaries

Host: https://<TENANT_DOMAIN>
Authorization: <AUTHORIZATION>
Accept: application/json
Content-Type: multipart/form-data;boundary="boundary"

--boundary
Content-Disposition: form-data; name="object"

{ "name": "agent.conf", "type": "text/plain" }
--boundary
Content-Disposition: form-data; name="file"; filename="agent.conf"
Content-Type: text/plain

c8y.url.http=https://management.cumulocity.com
c8y.url.mqtt=mqtt.cumulocity.com
--boundary--
```

The device is expected to perform the following actions:

1. Set operation status to EXECUTING
2. Create a configuration type event
3. Attach the configuration file to the event
4. Set the operation status to SUCCESSFUL

SmartREST example

Cumulocity provides the 526 static SmartREST template for typed `c8y_UploadConfigFile` operations:

1. Receive typed `c8y_UploadConfigFile` operation
`526,DeviceSerial,agent_conf`
2. Set operation status to EXECUTING
`501,c8y_UploadConfigFile`
3. Create a config type event using REST API
4. Attach the targeted config file to the event using REST API
5. Set operation status to SUCCESSFUL
`503,c8y_UploadConfigFile`

Install configuration file

Installing typed configuration also works very similarly to the legacy configuration. Adding the `c8y_DownloadConfigFile` to the device's `c8y_SupportedOperations` controls the availability of the **Send configuration to device** button. When it is clicked a `c8y_DownloadConfigFile` operation with the configuration type included is created.

```
{
  "c8y_DownloadConfigFile": {
    "type": "agent_conf",
    "url": "https://demos.cumulocity.com/inventory/binaries/156719"
  }
}
```

Name	Type	Mandatory	Description
type	string	Yes	Type of the configuration to apply
url	string	Yes	URL where the configuration file should be obtained from

When the device has downloaded and installed the configuration it must update the currently installed configuration of this specific type in its own managed object. This is done by adding the `c8y_Configuration_<config type>` fragment to the device's own managed object.

```
PUT /inventory/managedObjects/<deviceId>
```

```
{
  "c8y_Configuration_agent_conf": {
    "name": "agent_conf",
    "time": "2021-09-15T15:47:13.721Z",
    "type": "agent_conf",
    "url": "https://demos.cumulocity.com/inventory/binaries/156719"
  }
}
```

Name	Type	Mandatory	Description
c8y_Configuration_agent_conf	object	Yes	Fragment name with prefix "c8y_Configuration_" followed by the configuration type containing details of the currently installed configuration of that particular type
name	string	Yes	Optional name of the installed configuration file
time	string	Yes	ISO datetime indicating when the configuration was applied
type	string	Yes	Type of the configuration
url	string	Yes	URL where the configuration was obtained from

The device is expected to perform the following actions:

1. Set operation status to EXECUTING
2. Download configuration file from the included URL
3. Install the configuration file
4. Update the device's currently installed configuration with the typed configuration fragment in the device's own managed object
5. Set the operation status to SUCCESSFUL

SmartREST example

The 524 static SmartREST response template is available for typed `c8y_DownloadConfigFile` operations, and the 120 static template is prepared for uploading the current configuration:

1. Receive typed `c8y_UploadConfigFile` operation
524, DeviceSerial, https://demos.cumulocity.com/inventory/binaries/156719, agent_conf
2. Set operation status to EXECUTING
501, c8y_DownloadConfigFile
3. Download configuration from the included URL
4. Install the configuration as the targeted configuration type
5. Set the currently installed configuration
120, agent_conf, https://demos.cumulocity.com/inventory/binaries/156719, agent_conf.txt, 2021-09-15T15:47:13.721Z
6. Set operation status to SUCCESSFUL
503, c8y_DownloadConfigFile

CONNECTIVITY

The **Connectivity** tab integrates with a 3rd party SIM management platform to provide SIM management functionality within the Cumulocity Device Management application. The tab appears for a device when all of the following criteria are met:

1. Connectivity microservice is subscribed and configured
2. The device managed object contains the `c8y_Mobile` fragment with the MSISDN or ICCID property set
3. The SIM referenced by the device is managed by the SIM management provider configured for the tenant

```
PUT /inventory/managedObjects/<deviceId>
```

```
{
  "c8y_Mobile": {
    "msisdn": "380561234567",
    "iccid": "89100423481F445593U"
  }
}
```

Name	Type	Mandatory	Description
msisdn	string	No	MSISDN of the installed SIM
iccid	string	No	ICCID of the installed SIM

Depending on the configured connectivity provider either MSISDN or ICCID may be used to identify the SIM present in the device. We recommend you to always include both into the `c8y_Mobile` fragment. There are many more mobile connection related properties that may also be attached to the `c8y_Mobile` fragment, but only MSISDN or ICCID are relevant for connectivity management.

SmartREST example

The 111 static template is provided for devices to communicate their mobile information:

```
111,1234567890,89300000000000000459,54353
```

DEVICE AVAILABILITY

The **Device availability** tab shows the device's availability and connection status. To achieve this the device must communicate its required interval using the `c8y_RequiredAvailability` fragment in the device's own managed object. This action activates availability and connection monitoring for the device.

```
PUT /inventory/managedObjects/<deviceId>
```

```
{
  "c8y_RequiredAvailability": {
    "responseInterval": 15
  }
}
```

Name	Type	Mandatory	Description
responseInterval	integer	No	Expected response interval of the device in minutes. If it is <= 0 the device is considered in maintenance mode.

Usually devices should set their required interval only once during its first connection to the platform with a default value. Later changes can be left to platform users.

SmartREST example

The static template 117 is provided to set the required availability for SmartREST connected devices. This template can be sent in a fire-and-forget approach during device startup because it doesn't override already existing required availability configuration:

```
117,15
```

AVAILABILITY MONITORING

The response interval set in the `c8y_RequiredAvailability` fragment is used as interval in which the platform expects to receive data from the device. If no data is received in this interval the device will be marked as offline and an alarm of type `c8y_UnavailabilityAlarm` will be raised automatically. This alarm will also be cleared automatically when the device sends data again. No further action from the device is necessary.

The availability information computed by Cumulocity is stored in the fragments `c8y_Availability` and `c8y_Connection` of the device.

```
"c8y_Availability": { "lastMessage": "2022-05-21...", "status": "AVAILABLE" },
"c8y_Connection": { "status": "CONNECTED" }
```

Name	Type	Description
lastMessage	Date	The date and time when the device sent the last message to Cumulocity.
status	String	The current status, one of AVAILABLE, UNAVAILABLE, MAINTENANCE.

The following requests are considered a device's heartbeat and will mark the device as available and update the last message timestamp of a device, as long

as the `X-Cumulocity-Application-Key` header is not set:

- Creation of an event, measurement or alarm (for given device as source)
- Updates to the device itself (with a given ID), in the form of empty PUT requests or requests with an ID only, that is `{}` or `{"id": ... }`

INFO

Keep in mind that after updating the last message it may take some minutes until the new status has been saved in a database.

CONNECTION MONITORING

Cumulocity also provides connection monitoring for devices. When the device establishes a connection where it is able to receive operations the platform considers this device as connected. This applies to HTTP longpolling connection or a MQTT session equally.

A monitored device has one of the following statuses for `c8y_Connection`:

Status (string)	Description
CONNECTED	A device push connection is established.
DISCONNECTED	<code>responseInterval</code> is larger than 0 and the device is neither AVAILABLE nor CONNECTED.
MAINTENANCE	<code>responseInterval</code> is smaller or equal to 0; the device is under maintenance.

INFO

If a device is not connected via device push, but a message was sent within the required response interval, `c8y_Availability` can still have the status AVAILABLE, even if `c8y_Connection` does not have the status CONNECTED.

DEVICE INFORMATION

The **Device information** tab is a predefined dashboard with several widgets that combine default device information. The status widget, for example, will get its information from the `c8y_Availability` fragment, which holds information about the device's status and when it was last available. For details see [Availability](#).

DEVICE MARKER

A device is marked in the inventory with a `c8y_IsDevice` fragment in its own managed object. Only devices with this fragment appear in the all **All devices** list in the Device Management application.

```
PUT /inventory/managedObjects/<deviceId>
```

```
{
  "c8y_IsDevice": {}
}
```

The devices with this fragment can be queried and filtered for. Along with the **All devices** list, they will also be shown in groups as well as during the creation of bulk operations.

All devices including child devices should contain this fragment.

INFO

Devices created through SmartREST 2.0 will automatically contain this fragment.

AGENT MARKER

In order to receive any operation a device must declare the agent marker fragment in its own managed object. This will enable the platform to send operations to this device to for all child devices in its child hierarchy that don't carry this fragment themselves.

```
PUT /inventory/managedObjects/<deviceId>
```

```
{
  "com_cumulocity_model_Agent": {}
}
```

INFO

Devices created through SmartREST 2.0 will automatically contain this fragment.

DEVICE RESTART

Devices capable of restarting remotely can announce this capability by adding the `c8y_Restart` operation to the device's own `c8y_SupportedOperations` fragment. Then the **Device details** page will enable a **Restart** button within its context menu.

Restart operation

Upon clicking the **Restart** button in the Device Management application an operation as follows is sent:

```
{
  "c8y_Restart": {}
}
```

Field	Data type	Mandatory	Details
c8y_Restart	object	Yes	Restart marker fragment, that designates this operation as a restart operation

The device is expected to perform the following actions:

1. Set the operation status to EXECUTING
2. Perform the requested restart
3. Set the operation status to SUCCESSFUL

SmartREST example

Cumulocity provides the 510 static response template:

1. Device receives command via 510 static response template
`510,DeviceSerial`
2. Device sets operation status to EXECUTING
`501,c8y_Restart`
3. Device confirms successful execution by setting operation status to SUCCESSFUL
`503,c8y_Restart`

HARDWARE INFORMATION

Devices may announce their underlying hardware information to Cumulocity using the `c8y_Hardware` fragment in the device's own managed object.

```
PUT /inventory/managedObjects/<deviceId>
```

```
{
  "c8y_Hardware": {
    "serialNumber": "1234567890",
    "model": "myModel",
    "revision": "1.2.3"
  }
}
```

Field	Data type	Mandatory	Details
serialNumber	string	No	The hardware serial number of the device
model	string	No	A text identifier of the hardware model
revision	string	No	A text identifier of the hardware revision

SmartREST example

Upload hardware information using the 110 static template. Usually this can be done once during agent application startup:

```
110,1234567890,myModel,1.2.3
```

AGENT INFORMATION

All devices should provide information about the agent they are running, that is the software that integrates them with Cumulocity.

```
PUT /inventory/managedObjects/<deviceId>
```

```
{
  "c8y_Agent": {
    "name": "thin-edge.io",
    "version": "0.6",
    "url": "https://thin-edge.io",
    "maintainer": "Cumulocity"
  }
}
```

Field	Data type	Mandatory	Details
name	string	Yes	Name of the agent
version	string	Yes	Version of the agent
url	string	No	The agent URL
maintainer	string	Yes	Maintainer of the agent

SmartREST example

Upload agent details using the 122 static template:

```
122,thin-edge.io,0.6,https://thin-edge.io,Cumulocity
```

Do this once at agent initialization.

DEVICE PROFILE

The **Device profile** tab shows the different parameters of the added device profiles. From a device agent perspective, device profiles are a combination of firmware update, software update, and typed file-based device configuration. Large parts of the agent code to support these capabilities can be reused.

Device profile functionality is enabled when the device announces the `c8y_DeviceProfile` operation in its `c8y_SupportedOperations`. The **Device profile** tab allows users to apply a profile to a device. This creates a `c8y_DeviceProfile` operation according to the configured profile. If present the firmware, software, and configuration should be handled exactly like their individual operations (`c8y_Firmware`, `c8y_SoftwareUpdate`, and typed `c8y_DownloadConfigFile`). We recommend you to execute a `c8y_Profile` operation by installing firmware first, software second and configuration third to minimize the potential of later actions overriding earlier ones.

```

{
  "profileName": "my profile",
  "profileId": "158751",
  "c8y_DeviceProfile": {
    "software": [
      {
        "name": "curl",
        "action": "install",
        "softwareType": "rpm",
        "version": "2.3.4",
        "url": "http://my.url.com"
      },
      {
        "name": "cumulocity_agent",
        "action": "install",
        "version": "1.2.3",
        "url": "https://cumulocity.com/agent"
      }
    ],
    "configuration": [
      {
        "name": "ssh_conf",
        "type": "ssh_conf",
        "url": "http://cumulocity.com/conf"
      },
      {
        "name": "agent_conf",
        "type": "agent_conf",
        "url": "https://demos.cumulocity.com/inventory/binaries/156719"
      }
    ],
    "firmware": {
      "name": "device_fw",
      "version": "1.0.1",
      "url": "https://cumulocity.com/fw"
    }
  }
}

```

Name	Type	Mandatory	Description
profileName	string	Yes	Name of the device profile being applied
profileId	string	Yes	ID reference to the device profile object
c8y_DeviceProfile	object	Yes	Device profile object containing all information necessary for the installation
c8y_DeviceProfile.software	array	No	Array of software objects to install or remove
c8y_DeviceProfile.software.name	string	Yes	Name of the software package
c8y_DeviceProfile.software.action	string	Yes	Action to perform on the package describing if the software should be installed or removed
c8y_DeviceProfile.software.softwareType	string	No	Type of the software package. Can be used to describe the nature, role, or intended use of the software.
c8y_DeviceProfile.software.version	string	Yes	Version of the software
c8y_DeviceProfile.software.url	string	Yes	URL where the software binary should be obtained from
c8y_DeviceProfile.configuration	array	No	Array of configuration objects to apply
c8y_DeviceProfile.configuration.name	string	Yes	Name of the configuration
c8y_DeviceProfile.configuration.type	string	Yes	Type of the configuration
c8y_DeviceProfile.configuration.url	string	Yes	URL where the configuration file should be obtained from

Name	Type	Mandatory	Description
c8y_DeviceProfile.firmware	object	No	Firmware object containing target firmware details
c8y_DeviceProfile.firmware.name	string	Yes	Name of the firmware to install
c8y_DeviceProfile.firmware.version	string	Yes	Version of the firmware
c8y_DeviceProfile.firmware.url	string	Yes	URL where the firmware binary should be obtained from

When a device receives a `c8y_Profile` operation it should announce the target profile in its own managed object first.

```
PUT /inventory/managedObjects/<deviceId>
```

```
{
  "c8y_Profile": {
    "profileName": "my profile",
    "profileId": "158751",
    "profileExecuted": false
  }
}
```

Name	Type	Mandatory	Description
profileName	string	Yes	Name of the device profile
profileId	string	Yes	The ID reference of the device profile object
profileExecuted	boolean	Yes	Indicator showing if the profile has been applied fully; must be false in this context

After completing each of the three subsections the device must announce its current state in its own managed object the same way as described in the individual operations using the fragments `c8y_Firmware`, `c8y_SoftwareList`, and `c8y_Configuration_<type>` respectively. Then the device should update its installed profile state in its managed object by updating the `profileExecuted` property to true.

```
PUT /inventory/managedObjects/<deviceId>
```

```
{
  "c8y_Profile": {
    "profileName": "my profile",
    "profileId": "158751",
    "profileExecuted": true
  }
}
```

Name	Type	Mandatory	Description
profileName	string	Yes	Name of the device profile
profileId	string	Yes	The ID reference of the device profile object
profileExecuted	boolean	Yes	Indicator showing if the profile has been applied fully; must be true in this context

The device is expected to perform the following actions:

1. Set operation status to EXECUTING
2. Set the `c8y_Profile` fragment in the device's own managed object with `profileExecuted = false`
3. Install firmware if included and complete the installation by updating the `c8y_Firmware` fragment in its own managed object
4. Install software if included and complete the installation by updating the device installed software information, for example, by [setting software list](#) or by [setting advanced software list](#)
5. Install configuration if included and complete the installation by updating the `c8y_Configuration_<type>` fragment for each configuration in its own managed object
6. Set the `c8y_Profile` fragment in the device's own managed object with `profileExecuted = true`
7. Set the operation status to SUCCESSFUL

SmartREST example

In addition to the static templates for firmware, software, and configuration provided by Cumulocity there are specific templates available for handling device profiles. The 527 static response template is designed to receive the operation. The 121 static template can be used to set the current state of device profile:

1. Receive `c8y_DeviceProfile` operation

```
527,DeviceSerial,$FW,device_fw,1.0.1,https://cumulocity.com/fw,false,,$SW,curl,2.3.4,http://my.url.com,install,cumulocity_agent,1.2.3,https://cumulocity.com/agent,install,$CONF,http://cumulocity.com/conf,ssh_conf,https://demos.cumulocity.com/inventory/binaries/156719,agent_conf
```

Template 527 is triggered whenever a `c8y_DeviceProfile` operation is created and does not include the software item types. In cases where there is at least one software element in the software list with a defined software type, template 531 will also be triggered alongside with template 527. Template 531 carries the software type information. Both templates will be sent to devices simultaneously.

```
531,DeviceSerial,$FW,device_fw,1.0.1,https://cumulocity.com/fw,false,,$SW,curl,2.3.4,rpm,http://my.url.com,install,cumulocity_agent,1.2.3,https://cumulocity.com/agent,install,$CONF,http://cumulocity.com/conf,ssh_conf,https://demos.cumulocity.com/inventory/binaries/156719,agent_conf
```

2. Set operation status to EXECUTING
`501,c8y_DeviceProfile`
3. Set target profile
`121,false,`
4. Download, install and confirm firmware installation state
`115,device_fw,1.0.1,https://cumulocity.com/fw`
5. Download, install and confirm software installation state
`116,curl,2.3.4,http://my.url.com,cumulocity_agent,1.2.3,https://cumulocity.com/agent`
6. Download, install and confirm configuration installation state for each configuration
`120,ssh_conf,http://cumulocity.com/conf,config,`
`120,agent_conf,https://demos.cumulocity.com/inventory/binaries/156719,agent.cfg,`
7. Set the target profile as executed
`121,true,`
8. Set operation status to SUCCESSFUL
`503,c8y_DeviceProfile`

FIRMWARE

The **Firmware** tab displays currently installed firmware of a device and allows users to install a different version. A device can have only one firmware installed at a time. It depends on the device, what a firmware can be in Cumulocity. Typical use cases are: operating system, microcontroller firmware or BIOS.

Firmware can be installed with a full installation or with a patch. Which variant is sent to the device depends on how the firmware was created in the firmware repository.

Installed firmware

A device must announce its current state to the platform first. Then the installed firmware should be entered into the `c8y_Firmware` fragment into the device's own managed object. A device must upload its current state to Cumulocity during startup and any time a local change is detected. This includes cases where an update was triggered remotely.

```
PUT /inventory/managedObjects/<deviceId>
```

```
{
  "c8y_Firmware": {
    "name": "ubuntu core",
    "version": "20.04.2",
    "url": "http://test.com"
  }
}
```

Field	DataType	Mandatory	Details
name	string	Yes	Name of the firmware package
version	string	Yes	A version identifier of the firmware
url	string	No	A URL pointing to the location where the firmware file was obtained from

Similar to software the URL field is optional and may be omitted by devices.

SmartREST example

The 115 static template is available for devices to communicate their currently installed firmware state:

```
115,ubuntu core,20.04.3,http://test.com
```

INSTALLING A FIRMWARE IMAGE

When a user selects a complete firmware image for installation, an operation with a similar `c8y_Firmware` fragment as found in the device managed object is created. This operation should be considered as the desired state that should be achieved by the device.

```
{
  "c8y_Firmware": {
    "name": "ubuntu core",
    "version": "20.04.3",
    "url": "http://test.com"
  }
}
```

Field	DataType	Mandatory	Details
name	string	Yes	Name of the firmware package
version	string	Yes	A version identifier of the firmware
url	string	Yes	A URL pointing to the location where the firmware file should be downloaded from

The device is expected to perform the following actions:

1. Set operation status to EXECUTING
2. Install firmware image
3. Update the installed firmware state in the device's own managed object
4. Set operation status to SUCCESSFUL

Updating a devices often changes fundamental system components. This operation should always be considered as a critical operation. The device must ensure that all of its connection parameters to Cumulocity are preserved through the upgrade and that connectivity can be resumed afterwards. We recommend you to make sure that the device uses an A/B firmware switching mechanism with a possibility to roll back if necessary.

SmartREST example

Cumulocity provides the 515 static response template to deal with installing firmware images:

1. Receive `c8y_Firmware` (image) operation
`515,DeviceSerial,ubuntu core,20.04.3,http://test.com`
2. Set operation status to EXECUTING
`501,c8y_Firmware`
3. Install firmware image
4. Update device's installed firmware state in inventory
`115,ubuntu core,20.04.3,http://test.com`
5. Set operation status to SUCCESSFUL
`503,c8y_Firmware`

INSTALLING A FIRMWARE PATCH

In case a user selects a firmware patch to be installed on a device, a `c8y_Firmware` operation is created. In this case two additional parameters are included to help with installation of a firmware patch. The device agent is responsible for a firmware patch process instead of a regular installation.

```
{
  "c8y_Firmware": {
    "name": "ubuntu core",
    "version": "20.04.4",
    "url": "http://test.com",
    "dependency": "20.04.3",
    "isPatch": true
  }
}
```

Field	DataType	Mandatory	Details
name	string	Yes	Name of the firmware
version	string	Yes	A version identifier of the firmware
url	string	Yes	A URL pointing to the location of the firmware file
dependency	string	Yes	Version of the firmware the patch depends on
isPatch	boolean	Yes	Indicator showing that this firmware package is a patch

The device is expected to perform the following actions:

1. Set operation status to EXECUTING
2. Verify if the currently installed firmware version is equal to the dependency version
3. Install firmware patch
4. Update the installed firmware state in the device's own managed object
5. Set operation status to SUCCESSFUL

SmartREST example

Cumulocity provides the 525 static response template to deal with installing firmware patches. It works very similarly to the 515 template, it just adds the dependency parameter as fifth parameter. The fact that a patch instead of a complete image should be installed is implicit because this template is only triggered for patches.

1. Receive `c8y_Firmware` (patch) operation
`525,DeviceSerial,ubuntu core,20.04.3,http://test.com,20.04.3`
2. Set operation status to EXECUTING
`501,c8y_Firmware`
3. Verify if currently installed firmware version and dependency version match
4. Install firmware image
5. Update device's installed firmware state in inventory
`115,ubuntu core,20.04.3,http://test.com`
6. Set operation status to SUCCESSFUL
`503,c8y_Firmware`

INFO

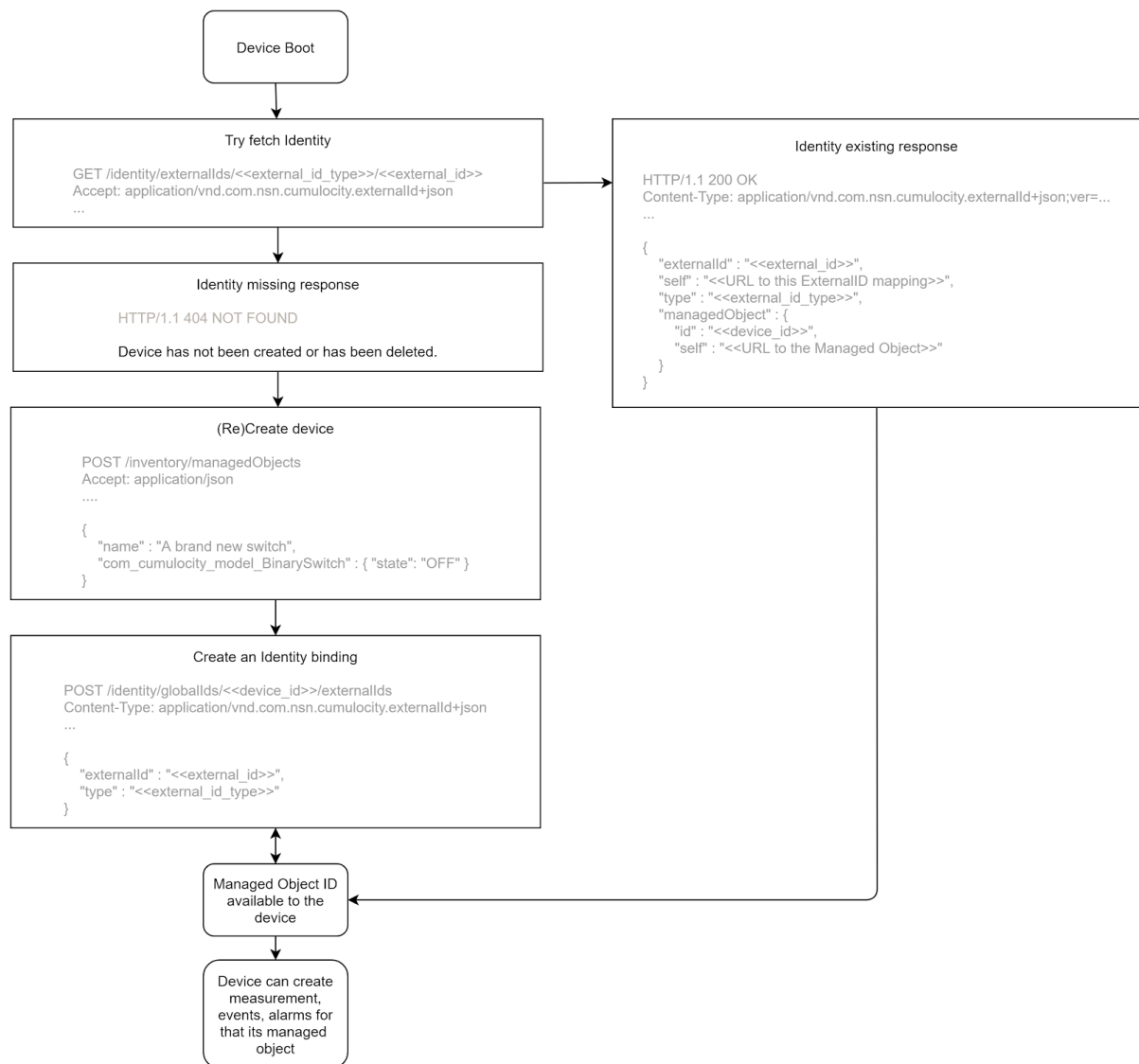
If the URL is set to an empty string, no firmware file is uploaded or linked from the platform. In this case, the device is fully responsible for resolving the correct binary. If the device cannot resolve the binary, it must mark the operation as **FAILED**.

This option is useful if the device can retrieve the correct firmware file dynamically based on its environment, architecture, or internal logic.

IDENTITY

The **Identity** tab shows all identities associated with the device. If no identities are available the tab is not shown. Identities map from a unique device identifier (for example, IMEI or SN) to the device's managed object in Cumulocity. This allows the device to find its managed object.

REST



MQTT (SmartREST 2.0)

In the context of MQTT the client ID is used as the device's external ID. Upon connecting the MQTT client ID is used to automatically associate the connection to a device in the inventory. No additional identity bindings must be created by the device. They are created internally when the 100 and 101 templates are used. All messages to create data are automatically associated with the device context.

Creating a device by a device connected to Cumulocity:

```
100, createdDeviceName, deviceType
```

Creating a child device for a device connected to Cumulocity:

```
101, uniqueChildId, myChildDevice, myChildType
```

LOGS

The **Logs** tab is used to extract logs from the device. The **Logs** tab is available if the fragment `c8y_LogfileRequest` is present in the `c8y_SupportedOperations` of the device. The device should contain a fragment called `c8y_SupportedLogs`, which holds an array of the types of logs that it supports. The types of logs will later be referenced when logs are requested.

SETTING SUPPORTED LOGS

Supported log types are announced by devices using the `c8y_SupportedLogs` fragment in the device's own managed object.

```
PUT /inventory/managedObjects/<deviceId>
```

```
{
  "c8y_SupportedLogs": [
    "syslog",
    "dmesg"
  ]
}
```

Field	DataType	Mandatory	Details
c8y_SupportedLogs	array	Yes	Array of strings of the supported log types

SmartREST example

The 118 static template is available to announce the supported logs of a device:

118, syslog, dmesg

UPLOADING LOG FILES

When users request log files from devices via the **Logs** tab a `c8y_LogfileRequest` operation is created.

```
{
  "c8y_LogfileRequest": {
    "searchText": "kernel",
    "logFile": "syslog",
    "dateTo": "2021-09-22T11:40:27+0200",
    "dateFrom": "2021-09-21T11:40:27+0200",
    "maximumLines": 1000
  }
}
```

Field	DataType	Mandatory	Details
dateFrom	string	Yes	Start date for log lines
dateTo	string	Yes	End date for log lines
logFile	string	Yes	Type of log for the specific device (c8y_SupportedLogs)
searchText	string	Yes	A text filter to apply to individual log lines
maximumLines	string	Yes	Maximum amount of lines to transfer

When the device has gathered the logs it uploads them to Cumulocity as a file. We recommend you to create an event and upload the log file as a binary attachment of the event. To avoid conflicts with other events bearing binary attachments (for example, for [typed file-based configuration](#)) we recommend you to use `c8y_Logfile` as event type. The following is an example of such an event:

```
POST /event/events
```

```
{
  "source": {
    "id": "4801"
  },
  "type": "c8y_Logfile",
  "time": "2021-09-15T15:57:41.311Z",
  "text": "syslog log file"
}
```

Field	DataType	Mandatory	Details
source	string	Yes	ID of the device

Field	Data Type	Mandatory	Details
type	string	Yes	Type of the event holding the log file should always be <code>c8y_Logfile</code>
time	string	Yes	Time when the event occurred
text	string	Yes	Event text

If desired the device may also include the `c8y_LogfileRequest` fragment from the operation or the operation ID into the event. The file is then attached to the event using its event ID and event binaries API.

```
POST /event/events/<eventId>/binaries

Host: https://<TENANT_DOMAIN>
Authorization: <AUTHORIZATION>
Accept: application/json
Content-Type: multipart/form-data;boundary="boundary"

--boundary
Content-Disposition: form-data; name="object"

{ "name": "syslog.txt", "type": "text/plain" }
--boundary
Content-Disposition: form-data; name="file"; filename="syslog.txt"
Content-Type: text/plain

Oct 25 13:28:53 wtp kernel: [ 719.554855] sd 6:0:0:0: [sdb] Write Protect is off
Oct 25 13:28:53 wtp kernel: [ 719.554864] sd 6:0:0:0: [sdb] Mode Sense: 03 00 00 00
Oct 25 13:28:53 wtp kernel: [ 719.555033] sd 6:0:0:0: [sdb] No Caching mode page found
--boundary--
```

After successful completion of the upload, the device must include a URL to the uploaded file into the `c8y_LogfileRequest` fragment of the operation. The link must be presented as property "file". This action can be combined with setting the operation status to SUCCESSFUL.

```
PUT /devicecontrol/operations/<operationId>
```

```
{
  "status": "SUCCESSFUL",
  "c8y_LogfileRequest": {
    "searchText": "kernel",
    "logFile": "syslog",
    "dateTo": "2021-09-22T11:40:27+0200",
    "dateFrom": "2021-09-21T11:40:27+0200",
    "maximumLines": 1000,
    "file": "https://demos.cumulocity.com/event/events/157700/binaries"
  }
}
```

Field	Data Type	Mandatory	Details
status	string	Yes	Operation status
dateFrom	string	Yes	Start date for log lines
dateTo	string	Yes	End date for log lines
logFile	string	Yes	Type of log for the specific device (<code>c8y_SupportedLogs</code>)
searchText	string	Yes	A text filter to apply to individual log lines
maximumLines	integer	Yes	Maximum amount of lines to transfer
file	string	Yes	URL where the log file was uploaded to

The device is expected to perform the following actions:

1. Set operation status to EXECUTING

2. Load, filter, and crop the log file as specified in the operation
3. Create a log file event
4. Upload the log file as binary attachment to said event
5. Set the operation status to SUCCESSFUL and include a URL to the uploaded log file

SmartREST example

Cumulocity provides the 522 static response template for receiving `c8y_LogfileRequest` operations. When the log file is uploaded the device may use the implicit parameter functionality of the 503 static template to set the operation status and provide the file link at the same time:

1. Receive `c8y_LogfileRequest` operation
`522,DeviceSerial,syslog,2021-09-21T11:40:27+0200,2021-09-22T11:40:27+0200,ERROR,1000`
2. Set operation status to EXECUTING
`501,c8y_LogfileRequest`
3. Create log file event using REST API
4. Upload log file as event binary to newly created event using REST API
5. Set operation status to SUCCESSFUL and supply uploaded file URL
`503,c8y_LogfileRequest,"https://demos.cumulocity.com/event/events/157700/binaries"`

MANUAL STATUS UPDATE

For cases where devices need a manual trigger for uploading a status update to the platform, the **Get measurements** button in the action bar of a device is available. It is shown when the device's `c8y_SupportedOperations` contains `c8y_MeasurementRequestOperation`.

This action creates a `c8y_MeasurementRequestOperation` operation.

```
{
  "c8y_MeasurementRequestOperation": {}
}
```

There is no additional configuration to specify the set of requested data points available. We recommend triggering an upload of the complete device state.

On receiving the operation the device is expected to perform the following actions:

1. Set the operation status to EXECUTING.
2. Upload the current status of all data points to the platform.
3. Set the operation status to SUCCESSFUL.

MEASUREMENTS

The **Measurements** tab creates a single graph per measurement fragment sent by the device. This means all included series will be shown together in one graph. Therefore, any device integrations must be made considering this grouping. Its visibility is controlled by the device's supported measurements. Cumulocity automatically and dynamically populates the device's supported measurements based on previously sent measurements. This means the measurements tab effectively appears after the device has sent its first measurement.

SmartREST example

There are several static templates available to create measurements in the 2xx range of message IDs. We provide the 200 static template to create a measurement with a dynamic fragment and series:

```
200,c8y_Temperature,T,25
```

While using this template is possible for many use cases, we recommend you to create a custom template for all use cases where dynamically defining fragment and series are not required.

NETWORK

The **Network** tab displays network information. It is shown if the `c8y_Network` fragment is present in the device managed object. There are three subsections: WAN, LAN and DHCP. Each of these can be activated by the nested fragments `c8y_WAN`, `c8y_LAN`, and `c8y_DHCP` respectively.

Network status

Devices may announce their current local network status and configuration to the platform using the `c8y_Network` fragment in the device's own managed object.

```
PUT /inventory/managedObjects/<deviceId>
```

```
{
  "c8y_Network": {
    "c8y_LAN": {
      "netmask": "255.255.255.0",
      "ip": "192.168.128.1",
      "name": "br0",
      "enabled": 1,
      "mac": "00:60:64:dd:a5:c3"
    },
    "c8y_WAN": {
      "password": "user-password",
      "simStatus": "SIM OK",
      "authType": "chap",
      "apn": "example.apn.com",
      "username": "test"
    },
    "c8y_DHCP": {
      "dns2": "1.1.1.1",
      "dns1": "8.8.8.8",
      "domainName": "my.domain",
      "addressRange": {
        "start": "192.168.128.100",
        "end": "192.168.128.199"
      },
      "enabled": 1
    }
  }
}
```

Name	Type	Mandatory	Description
c8y_LAN	object	No	Optional nested object containing local network information
c8y_LAN.netmask	string	No	Subnet mask configured for the network interface
c8y_LAN.ip	string	No	IP address configured for the network interface
c8y_LAN.name	string	No	Identifier for the network interface
c8y_LAN.enabled	integer	No	Indicator showing if the interface is enabled or not
c8y_LAN.mac	string	No	MAC address of the network interface
c8y_WAN	object	No	Optional nested object describing mobile internet connectivity interface status
c8y_WAN.password	string	No	SIM connectivity password
c8y_WAN.simStatus	string	No	SIM connection status
c8y_WAN.authType	string	No	Auth type used by the SIM connectivity
c8y_WAN.apn	string	No	APN used for internet access
c8y_WAN.username	string	No	SIM connectivity username
c8y_DHCP	object	No	Optional nested object containing information for DHCP server status
c8y_DHCP.dns1	string	No	First configured DNS server
c8y_DHCP.dns2	string	No	Second configured DNS server

Name	Type	Mandatory	Description
c8y_DHCP.domainName	string	No	Domain name
c8y_DHCP.addressRange.start	string	No	Start of address range assigned to DHCP clients
c8y_DHCP.addressRange.end	string	No	End of address range assigned to DHCP clients
c8y_DHCP.enabled	integer	No	Indicator showing if the DHCP server is enabled or not

SETTING NETWORK CONFIGURATION

If the device contains the `c8y_Network` operation in its `c8y_SupportedOperations` users may also update a device's network configuration in the **Network** tab. The changed configuration is sent as `c8y_Network` operation with a very similar fragment as also present in the device managed object. The `c8y_Network` fragment within this operation may contain one or more of its nested fragments.

```
{
  "c8y_Network": {
    "c8y_LAN": {
      "netmask": "255.255.255.0",
      "ip": "192.168.128.1",
      "enabled": 1
    },
    "c8y_WAN": {
      "password": "user-password",
      "authType": "chap",
      "apn": "example.apn.com",
      "username": "ee"
    },
    "c8y_DHCP": {
      "dns2": "1.1.1.1",
      "dns1": "8.8.8.8",
      "domainName": "my.domain",
      "addressRange": {
        "start": "192.168.128.100",
        "end": "192.168.128.199"
      },
      "enabled": 1
    }
  }
}
```

Name	Type	Mandatory	Description
c8y_LAN	object	No	Optional nested object containing local network information
c8y_LAN.netmask	string	Yes	Subnet mask configured for the network interface
c8y_LAN.ip	string	Yes	IP address configured for the network interface
c8y_LAN.enabled	integer	Yes	Indicator showing if the interface is enabled or not
c8y_WAN	object	No	Optional nested object describing mobile internet connectivity interface status
c8y_WAN.password	string	Yes	SIM connectivity password
c8y_WAN.authType	string	Yes	Auth type used by the SIM connectivity
c8y_WAN.apn	string	Yes	APN used for internet access
c8y_WAN.username	string	Yes	SIM connectivity username
c8y_DHCP	object	No	Optional nested object containing information for DHCP server status
c8y_DHCP.dns1	string	No	First configured DNS server

Name	Type	Mandatory	Description
c8y_DHCP.dns2	string	No	Second configured DNS server
c8y_DHCP.domainName	string	No	Domain name
c8y_DHCP.addressRange.start	string	No	Start of address range assigned to DHCP clients
c8y_DHCP.addressRange.end	string	No	End of address range assigned to DHCP clients
c8y_DHCP.enabled	integer	No	Indicator showing if the DHCP server is enabled or not

The device is expected to perform the following actions:

1. Set operation status to EXECUTING
2. Apply WAN, LAN, and DHCP configuration
3. Set new network configuration status the device managed object
4. Set operation status to SUCCESSFUL

Changes to the network configuration potentially impact the device's ability to connect to the Cumulocity platform. We recommend you to implement a connection test with the new settings. If the test fails the device should rollback the settings to their previous state and set the operation status to FAILED.

SmartREST example

There are no static templates available for populating the network fragment or receiving network operations. A custom template can be used instead:

```
10,100,PUT,INVENTORY,false,c8y_Network.c8y_LAN.name,STRING,,c8y_Network.c8y_LAN.ip,STRING,,c8y_Network.c8y_LAN.netmask,STRING,,c8y_Network.c8y_LAN.mac,STRING,,c8y_Network.c8y_LAN.enabled,STRING,,c8y_Network.c8y_WAN.simStatus,STRING,,c8y_Network.c8y_WAN.apn,STRING,,c8y_Network.c8y_WAN.username,STRING,,c8y_Network.c8y_WAN.password,STRING,,c8y_Network.c8y_WAN.authType,STRING,,c8y_Network.c8y_DHCP.addressRange.start,STRING,,c8y_Network.c8y_DHCP.addressRange.end,STRING,,c8y_Network.c8y_DHCP.domainName,STRING,,c8y_Network.c8y_DHCP.dns1,STRING,,c8y_Network.c8y_DHCP.dns2,STRING,,c8y_Network.c8y_DHCP.enabled,STRING,
```

```
11,200,,c8y_Network.c8y_WAN,c8y_Network.c8y_WAN.apn,c8y_Network.c8y_WAN.username,c8y_Network.c8y_WAN.password,c8y_Network.c8y_WAN.authType
```

```
11,201,,c8y_Network.c8y_LAN,c8y_Network.c8y_LAN.ip,c8y_Network.c8y_LAN.netmask,c8y_Network.c8y_LAN.enabled
```

```
11,202,,c8y_Network.c8y_DHCP,c8y_Network.c8y_DHCP.addressRange.start,c8y_Network.c8y_DHCP.addressRange.end,c8y_Network.c8y_DHCP.domainName,c8y_Network.c8y_DHCP.dns1,c8y_Network.c8y_DHCP.dns2,c8y_Network.c8y_DHCP.enabled
```

The example custom template provides the 100 template for the device to upload its current network configuration completely. At device agent startup and each time a change is detected this template should be used to synchronize the information in the cloud with the local truth.

The example template also provides three response templates to receive the three different nested fragments as operations. Each response template may be handled separately.

WAN

1. Receive WAN configuration operation
200,example.apn.com,user,secret,chap
2. Set operation status to EXECUTING
501,c8y_Network
3. Apply WAN configuration
4. Update device's network configuration in inventory
100,br0,192.168.128.1,255.255.255.0,00:60:64:dd:a5:c3,1,SIM
OK,example.apn.com,user,secret,chap,192.168.128.100,192.168.128.199,my.domain,8.8.8.8,1.1.1.1,1
5. Set operation status to SUCCESSFUL
503,c8y_Network

LAN

1. Receive LAN configuration operation
201,192.168.128.2,255.255.255.0,1
2. Set operation status to EXECUTING
501,c8y_Network
3. Apply LAN configuration
4. Update device's network configuration in inventory
100,br0,192.168.128.2,255.255.255.0,00:60:64:dd:a5:c3,1,SIM
OK,example.apn.com,user,secret,chap,192.168.128.100,192.168.128.199,my.domain,8.8.8.8,1.1.1.1,1
5. Set operation status to SUCCESSFUL
503,c8y_Network

DHCP

- 1. Receive DHCP configuration operation
`202,192.168.128.150,192.168.128.199,my.other.domain,1.1.1.1,8.8.8.8,1`
- 2. Set operation status to EXECUTING
`501,c8y_Network`
- 3. Apply DHCP configuration
- 4. Update device's network configuration in inventory
`100,br0,192.168.128.1,255.255.255.0,00:60:64:dd:a5:c3,1,SIM`
`OK,example.apn.com,user,secret,chap,192.168.128.150,192.168.128.199,my.other.domain,1.1.1.1,8.8.8.8,1`
- 5. Set operation status to SUCCESSFUL
`503,c8y_Network`

PLATFORM CAPABILITIES

Devices may require information about platform capabilities. To enable new device-side functionality, a new API or optional components may be required. For this purpose, Cumulocity provides dedicated interfaces.

PLATFORM VERSION

To enable functionality that requires a minimum version of the Cumulocity platform it is best practice to first query this version before attempting to use a newly added API.

```
GET /tenant/system/options/system/version

{
  "category": "system",
  "value": "2025.0.0",
  "key": "version"
}
```

Field	Data type	Details
category	string	Category of the system option (always "system")
value	string	Platform version
key	string	Key of the system option (always "version")

SmartREST example

The 600 static request template and its corresponding 601 static response template are available for SmartREST enabled devices:

- Sending request `600`
- Receiving response `601,2025.0.0`

RELAY

RELAYS

A relay is a kind of binary state switch which can be in the states OPEN or CLOSED. Relays can be used for many purposes, for example to connect or disconnect the consumer power supply through a smart energy meter. In a managed object, a relay control model includes the state of the control. When the control state changes, the inventory model should be updated to include the new state.

Single Relay

To manage and monitor a single relay Cumulocity offers the [Relay control widget](#).

Relay state

Devices may announce their relay state by updating the `c8y_Relay` fragment in their managed object.


```
PUT /inventory/managedObjects/<deviceId>
```

```
"c8y_Relay": {
  "relayState": "OPEN"
}
```

Name	Type	Mandatory	Description
c8y_Relay.relayState	string	Yes	The relay state: "OPEN" or "CLOSED".

Setting relay state

Devices that support changing the relay position remotely may add the `c8y_Relay` operation to `c8y_SupportedOperations`. Then users can request an update to the relay position in the [Relay control widget](#). This creates an operation of fragment type `c8y_Relay` for the device.

The operation representation is the same as the inventory representation:

```
"c8y_Relay": {
  "relayState": "OPEN"
}
```

Operation	States	Description
state	OPEN, CLOSED	OPEN commands the relay into the open position, CLOSED commands it to the closed position.

On receiving the operation the device is expected to perform the following actions:

1. Set the operation status to EXECUTING.
2. Set the relay state.
3. Set the new relay state in its managed object.
4. Set the operation status to SUCCESSFUL.

SmartREST example

Cumulocity provides the 518 static response template for setting the relay state.

1. The device receives the command via the 518 static response template
`518,OPEN`
2. The device sets the operation status to EXECUTING
`501,c8y_Relay`
3. The device sets its relay state.
4. The device confirms successful execution by setting the operation status to SUCCESSFUL
`503,c8y_Relay`

Multiple relays

To manage and monitor multiple relays Cumulocity offers the [Relay array control widget](#).

Multiple relays state

Devices may announce their multiple relays state by updating the `c8y_RelayArray` fragment in their managed object.

```
PUT /inventory/managedObjects/<deviceId>
```

```
"c8y_RelayArray": [
  "OPEN",
  "CLOSED",
  "CLOSED",
  "OPEN"
]
```

Name	Type	Mandatory	Description
c8y_RelayArray	array	Yes	Array of strings of relays states.

Setting multiple relays states

Devices that support changing their relays positions remotely may add the `c8y_RelayArray` operation to `c8y_SupportedOperations`. Then users can request an update to the relay position in the [Relay array control widget](#). This creates an operation of fragment type `c8y_RelayArray` for the device.

The operation representation is the same as the inventory representation:

```
"c8y_RelayArray" : [
  "OPEN",
  "CLOSED",
  "CLOSED",
  "OPEN"
]
```

On receiving the operation the device is expected to perform the following actions:

1. Set the operation status to EXECUTING.
2. Set the relays states to the respective values.
3. Set the new relays states in its managed object.
4. Set the operation status to SUCCESSFUL.

SmartREST example

Cumulocity provides the 519 static response template:

1. The device receives the command via the 519 static response template
`519,OPEN,CLOSED,CLOSED,OPEN`
2. The device sets the operation status to EXECUTING
`501,c8y_RelayArray`
3. The device confirms successful execution by setting the operation status to SUCCESSFUL and sets the relay array state with the implicit parameters in the 503 static template
`503,c8y_RelayArray,OPEN,CLOSED,CLOSED,OPEN`

REMOTE ACCESS

The **Remote access** tab is used to configure and access devices for remote control through remote control protocols.

The **Remote access** tab is available if the following criteria are met:

- The Cloud Remote Access microservice is subscribed to the needed tenant
- The user has the correct permissions granted (Remote access admin rights)
- `c8y_RemoteAccessConnect` is added to the device's `c8y_SupportedOperations`

For more information, see [Using Cloud Remote Access](#).

REMOTE ACCESS CONNECT

When a user selects a remote access endpoint and clicks the **Connect** button a `c8y_RemoteAccessConnect` operation is created.

```
{
  "c8y_RemoteAccessConnect": {
    "hostname": "10.0.0.67",
    "port": 5900,
    "connectionKey": "eb5e9d13-1caa-486b-bdda-130ca0d87df8"
  }
}
```

Field	DataType	Mandatory	Details
connectionKey	string	Yes	Shared secret to authenticate the connection request from device side
hostname	string	Yes	Endpoint on local network to connect to
port	integer	Yes	Port to be used on the local network endpoint

With this operation the device must open a WebSocket connection to the microservice endpoint using the connection key for authentication (`wss://<c8y`

`host>/service/remotaccess/device/<connectionKey>`) and a local socket to the specified hostname and port. Then it must establish bidirectional forwarding of data between the WebSocket and the local socket.

Cumulocity currently supports VNC, SSH, and Telnet. The device agent should be implemented independently of the remote access protocol used. The operation intentionally does not transfer the protocol. The agent must be capable of forward arbitrary data between both its established sockets.

The device is expected to perform the following actions:

1. Set operation status to EXECUTING
2. Establish WebSocket connection to remote access microservice
3. Establish local socket connection to the specified host and port
4. Establish bidirectional forwarding between WebSocket and local socket
5. Set operation status to SUCCESSFUL

The operation is set to SUCCESSFUL after the connection is established. Disconnecting happens silently. Even if the connection was not terminated gracefully by any of the involved components, the operation status must stay in SUCCESSFUL. Whenever one of the connections is terminated (WebSocket or TCP) the device agent should consider the session as ended and should also terminate both connections.

SmartREST example

The 530 static response template is available for receiving `c8y_RemoteAccessConnect` operations:

1. Receive `c8y_RemoteAccessConnect` operation
`530,DeviceSerial,10.0.0.67,22,eb5e9d13-1caa-486b-bdda-130ca0d87df8`
2. Device sets operation status to EXECUTING
`501,c8y_RemoteAccessConnect`
3. Establish WebSocket connection using HTTP
4. Establish local socket connection
5. Device sets operation status to SUCCESSFUL
`503,c8y_RemoteAccessConnect`

SHELL

The **Shell** tab allows to send arbitrary device-specific commands to the device. It is shown if the `c8y_Command` operation is present in the device's `c8y_SupportedOperations`.

SEND A COMMAND TO A DEVICE

You may enter an arbitrary string into the command text. The format and its interpretation is up to the device integration. Click the **Execute** button to create a `c8y_Command` operation.

```
{
  "c8y_Command": {
    "text": "get sw.version; get hw.version"
  }
}
```

Field	DataType	Mandatory	Details
<code>c8y_Command.text</code>	string	Yes	The command text to be executed by the device

After completing the execution, the device must provide a return string for the command in addition to setting the operation status to SUCCESSFUL. The result is provided as string property nested within the `c8y_Command` fragment in the operation.

```
PUT /devicecontrol/operations/<operationId>
```

```
{
  "status": "SUCCESSFUL",
  "c8y_Command": {
    "text": "get sw.version; get hw.version",
    "result": "1.2.3; 9.8.7"
  }
}
```

Field	DataType	Mandatory	Details
status	string	Yes	Operation status indicating if the operation was completed as intended
c8y_Command	object	Yes	c8y_Command object received via the pending operation at the beginning
c8y_Command.text	string	Yes	The command that was executed by the device
c8y_Command.result	string	Yes	Execution result after running the command

The device is expected to perform the following actions:

1. Set operation status to EXECUTING
2. Execute the command and obtain the return value
3. Set the operation status to SUCCESSFUL and include the return value in it

SmartREST example

For SmartREST connected devices the 511 static response template is available for receiving `c8y_Command` operations. Additionally, the implicit parameter functionality of the 503 static template can be used to supply the return value:

1. Receive `c8y_Command` operation
`511,DeviceSerial,"get sw.version; get hw.version"`
2. Set operation status to EXECUTING
`501,c8y_Command`
3. Set operation status to SUCCESSFUL and supply return value
`503,c8y_Command,"1.2.3; 9.8.7"`

SOFTWARE

The **Software** tab allows you to install and uninstall a set of software files for a device. The files can be located using an URL or they can be hosted in the Cumulocity Software Repository. Device agents are fully responsible for their local installation, management, and uninstall procedures and any kind of error handling during the operation.

The **Device details** page shows a **Software** tab for devices that announce `c8y_SoftwareList` and/or `c8y_SoftwareUpdate` in their `c8y_SupportedOperations` fragment in their device managed objects. It also shows a **Services** tab for devices that have at least one software service running. The service can have measurements, alarms and events assigned.

INSTALLED SOFTWARE

The installed software packages are listed in the `c8y_SoftwareList` fragment which may be placed in the device managed object or in the single child addition of type `c8y_InstalledSoftwareList`. The first approach to managing software is referred to as "legacy" and the second as "advanced".

A software package list entry must contain the following properties:

Field	Mandatory	Details
name	Yes	The name of the software
version	Yes	A version identifier of the software
url	No	A URL pointing to the location where the software file was obtained from
softwareType	No	An arbitrary string for organizing software artifacts

The name and the version are used to identify the package. Already mentioned `c8y_SupportedSoftwareTypes` fragment restricts possible software types that can be installed on the device.

Legacy Software Management

A device may update its software list by updating its managed object `c8y_SoftwareList` fragment.

```
PUT /inventory/managedObjects/<deviceId>
```

```
{
  "c8y_SoftwareList": [
    {
      "name": "software_a",
      "version": "3.0.0",
      "url": "http://example.com/software_a",
      "softwareType": "type A"
    },
    {
      "name": "software_b",
      "version": "2.0.0",
      "url": "http://example.com/software_b",
      "softwareType": "type B"
    }
  ]
}
```

Devices should upload the complete list of installed software during startup. Additionally the list should be updated any time a local change is triggered or detected. This includes cases where a change was requested through Cumulocity UI.

SmartREST example

Cumulocity provides the static SmartREST template 116 for devices to upload their installed software. It takes a dynamic length list of triples per software package as parameters. Each triple is interpreted as the name, version, and URL property of an individual package:

```
116,software_a,3.0.0,http://example.com/software_a,software_b,2.0.0,http://example.com/software_b
```

Changing installed software

Within the **Software** tab users can select software to install, to update, and to uninstall for a device. After confirming, the desired software configuration is sent to the device as an operation. The operation format depends on the device's `c8y_SupportedOperations` fragment.

Software list

If the device only supports the `c8y_SoftwareList` operation and the `c8y_SupportedOperations` fragment does not contain `c8y_SoftwareUpdate`, a `c8y_SoftwareList` operation is sent to the device. This operation contains a very similar `c8y_SoftwareList` fragment to the one that is already present in the device's own managed object. The `c8y_SoftwareList` operation always contains the entire list of software that should be installed on the device. Exactly the packages in the list should be installed. Any installed packages not contained in the list should be removed.

```
{
  "c8y_SoftwareList": [
    {
      "name": "software_a",
      "version": "4.0.0",
      "url": "http://example.com/software_a"
    },
    {
      "name": "software_b",
      "version": "3.0.0",
      "url": "http://example.com/software_b"
    }
  ]
}
```

Field	DataType	Mandatory	Details
name	string	Yes	Name of the software
version	string	Yes	A version identifier of the software
url	string	Yes	A URL pointing to the location where the software file should be downloaded from

The device is expected to perform the following actions:

1. Set operation status to EXECUTING
2. Uninstall software that should be uninstalled
3. Install software that should be installed
4. Update the software list in the device's own managed object
5. Set operation status to SUCCESSFUL

If the desired state cannot be achieved for any reason the operation should be concluded with status FAILED.

SmartREST example

The 516 static response template is available for dealing with software list operations. It works very similarly to the 116 template used for updating the device's own managed object:

1. Receive `c8y_SoftwareList` operation
`516,DeviceSerial,software_a,4.0.0,http://example.com/software_a,software_b,3.0.0,http://example.com/software_b`
2. Set operation status to EXECUTING
`501,c8y_SoftwareList`
3. Uninstall and install software
4. Update device's software list in inventory
`116,software_a,4.0.0,http://example.com/software_a,software_b,3.0.0,http://example.com/software_b`
5. Set operation status to SUCCESSFUL
`503,c8y_SoftwareList`

Software update

If a device supports the `c8y_SoftwareUpdate` operation in its `c8y_SupportedOperations` fragment the **Software** tab will create `c8y_SoftwareUpdate` operations for the device. Conceptually the software update is very similar to the software list. A desired state of software is sent to the device in form of a list of packages. The difference is that the `c8y_SoftwareUpdate` should be considered as a partial list. Each list element contains an additional instruction whether the package should be installed or uninstalled. Any package not listed in the software update list should not be touched.

```
{
  "c8y_SoftwareUpdate": [
    {
      "name": "software_a",
      "version": "4.0.0",
      "url": "http://example.com/software_a",
      "action": "install"
    },
    {
      "name": "software_b",
      "version": "3.0.0",
      "url": "http://example.com/software_b",
      "action": "delete"
    }
  ]
}
```

Field	DataType	Mandatory	Details
name	string	Yes	Name of the software
version	string	Yes	A version identifier of the software
url	string	Yes	A URL pointing to the location where the software file should be downloaded from
action	string	Yes	Action to be executed from the device on the software (possible values: "install" or "delete")

The device is expected to perform the following actions:

1. Set operation status to EXECUTING
2. Iterate through the list of packages contained in the operation and perform the respective action for each one
3. Update the software list in the device's own managed object
4. Set operation status to SUCCESSFUL

SmartREST example

The 528 static response template is available for dealing with software update operations:

1. Receive `c8y_SoftwareUpdate` operation
`528,DeviceSerial,software_a,4.0.0,http://example.com/software_a,install,software_b,3.0.0,http://example.com/software_b,delete`
2. Set operation status to EXECUTING
`501,c8y_SoftwareUpdate`
3. Uninstall and install software
4. Update device's software list in inventory
`116,software_a,3.0.0,http://example.com/software_a`
5. Set operation status to SUCCESSFUL
`503,c8y_SoftwareUpdate`



INFO

If the URL is set to an empty string, no file is uploaded or linked from the platform. In this case, the device is fully responsible for resolving the correct binary. Typically, this process works through package managers that retrieve the correct binary from the name, version, and other device-specific details. If the device cannot resolve the binary, it must mark the operation as FAILED.

ADVANCED SOFTWARE MANAGEMENT

Using the “advanced” approach, the `c8y_SoftwareList` fragment is no longer present in the device managed object. The data is separated from the device managed object which keeps the size of the device managed object low even for very large lists of installed software. All installed software for a device can be read and managed through the Advanced Software Management default microservice.

Devices support Advanced Software Management when they include the `c8y_SoftwareUpdate` operation in their `c8y_SupportedOperations` fragment and list their supported software types in the `c8y_SupportedSoftwareTypes` fragment.

```
{
  "c8y_SupportedOperations": [
    "c8y_SoftwareUpdate"
  ],
  "c8y_SupportedSoftwareTypes": [
    "type_a",
    "type_b"
  ]
}
```

Querying, adding and removing software packages can be done with the microservice REST endpoints or using SmartREST static templates.

Querying the software packages:

```
GET /service/advanced-software-mgmt/software?deviceId=<deviceId>
```

```
{
  "softwareList": [
    {
      "name": "software_a",
      "version": "3.0.0",
      "url": "http://example.com/software_a",
      "softwareType": "type A"
    },
    {
      "name": "software_b",
      "version": "2.0.0",
      "url": "http://example.com/software_b",
      "softwareType": "type B"
    }
  ],
  "statistics": {
    "currentPage": 1,
    "pageSize": 5
  }
}
```

Query paramater	Mandatory	Details
deviceId	Yes	ID of the device
name	No	Filter parameter for the software name
version	No	Filter parameter for the software version
type	No	Filter parameter for the software type
pageSize	No	The number of items on the page of the paginated result, between 1 and 2000
currentPage	No	The current page of the paginated result

Query paramater	Mandatory	Details
withTotalPages	No	When set to <code>true</code> , the returned result will contain the total number of the pages in the statistics object

Setting software packages

Advanced Software Management allows devices to set their installed software, similarly to legacy software management. In this case any software communicated to the platform before is overwritten entirely with then new packages.

```
POST /service/advanced-software-mgmt/software?deviceId=<deviceId>
```

```
[
  {
    "name": "software_a",
    "version": "3.0.0",
    "url": "http://example.com/software_a",
    "softwareType": "type A"
  },
  {
    "name": "software_b",
    "version": "2.0.0",
    "url": "http://example.com/software_b",
    "softwareType": "type B"
  }
]
```

SmartREST example

Devices may also use the SmartREST static template 140 instead. It takes a list of software packages of dynamic length, where each package is represented by its name, version, software type and URL:

```
140,software_a,3.0.0,"type A",http://example.com/software_a,software_b,2.0.0,"type B",http://example.com/software_b
```

Adding software packages

With Advanced Software Management devices may also append packages to their installed software without having to announce the entire list.

```
PUT /service/advanced-software-mgmt/software?deviceId=<deviceId>
```

```
[
  {
    "name": "software_a",
    "version": "3.0.0",
    "url": "http://example.com/software_a",
    "softwareType": "type A"
  },
  {
    "name": "software_b",
    "version": "2.0.0",
    "url": "http://example.com/software_b",
    "softwareType": "type B"
  }
]
```

SmartREST example

Devices also use the SmartREST static template 141 instead. Similarly to 140, it takes a list of software packages of dynamic

```
141,software_a,3.0.0,"type A",http://example.com/software_a,software_b,2.0.0,"type B",http://example.com/software_b
```

Removing software

In order to complete partial updates of installed software Advanced Software Management offers an interface to remove individual packages from a device's installed software.

```
DELETE /service/advanced-software-mgmt/software?deviceId=<deviceId>
```



```
[
  {
    "name": "software_a",
    "version": "3.0.0"
  },
  {
    "name": "software_b",
    "version": "2.0.0"
  }
]
```

SmartREST example

Devices may also use the SmartREST static template 142 instead. It takes a list of software packages of dynamic length, where each package is represented by its name and version, as URL and software type are not used to identify a package:

142,software_a,3.0.0,software_b,2.0.0

Changing installed software

Similarly, in the Advanced Software Management approach updating software packages requires sending to the device one of the operations: `c8y_SoftwareUpdate` or `c8y_SoftwareList`, depending on which are specified in `c8y_SupportedOperations` fragment. The only difference is that now software type property is required for software packages.

Software update

The `c8y_SoftwareUpdate` operation contains also partial list of software packages, each with an instruction whether it should be installed or uninstalled. This is very similar to legacy software management, however an additional parameter indicating the software type of each package is also included.

```
{
  "c8y_SoftwareUpdate": [
    {
      "name": "software_a",
      "version": "4.0.0",
      "url": "http://example.com/software_a",
      "softwareType": "type A",
      "action": "install"
    },
    {
      "name": "software_b",
      "version": "3.0.0",
      "url": "http://example.com/software_b",
      "softwareType": "type B",
      "action": "delete"
    }
  ]
}
```

Field	DataType	Mandatory	Details
name	string	Yes	Name of the software
version	string	Yes	A version identifier of the software
url	string	Yes	A URL pointing to the location where the software file should be downloaded from
softwareType	string	Yes	An arbitrary string for organizing software artifacts
action	string	Yes	Action to be executed from the device on the software (possible values: "install" or "delete")

The device is expected to perform the following actions:

- 1. Set operation status to EXECUTING
- 2. Iterate through the list of packages contained in the operation and perform the respective action for each one
- 3. Update the software list in the device's own managed object
- 4. Set operation status to SUCCESSFUL

SmartREST example

The 529 static response template is available for dealing with software update operations for devices that support Advanced Software Management:

1. Receive `c8y_SoftwareUpdate` operation
`529,DeviceSerial,software_a,4.0.0,"type A",http://example.com/software_a,install,software_b,3.0.0,"type B",http://example.com/software_b,delete`
2. Set operation status to EXECUTING
`501,c8y_SoftwareUpdate`
3. Uninstall and install software
4. Remove from the inventory uninstalled software packages
`142,software_b,3.0.0`
5. Add to the inventory installed software packages
`141,software_a,4.0.0,"type A",http://example.com/software_a`
6. Set operation status to SUCCESSFUL
`503,c8y_SoftwareUpdate`

SERVICES

The Cumulocity UI allows you to monitor software services running on a device. The services are represented in Cumulocity domain model as the device managed object child additions with `c8y_Service` type.

The **Device details** page shows a **Services** tab for devices that have at least one software service. A service can have measurements, alarms and events assigned.

Query, update, add and remove services using Cumulocity REST API for manipulating managed objects.

REST API EXAMPLES

Announcing a service to the platform

Using the Inventory REST API:

```
POST /inventory/managedObjects/<deviceId>/childAdditions
```

```
Content-Type: "application/vnd.com.nsn.cumulocity.managedObject+json"
```

```
{
  "name": "DatabaseService",
  "type": "c8y_Service",
  "serviceType": "systemd",
  "status": "up"
}
```

Field	Mandatory	Details
name	Yes	Name of the service
type	Yes	Type of the managed object, must always be 'c8y_Service'
serviceType	Yes	An arbitrary string for organizing services
status	Yes	'up', 'down', 'unknown' or any custom service status

Using [SmartREST static template 102](#) sent to topic `s/us/<serviceId>` :

The second parameter, the unique ID, does not reference the internal numeric ID but a string-based external ID which is defined by the device instead of the platform. We recommend you to prefix the unique ID with a device-specific prefix to avoid clashes with other devices running the same service:

```
102,myDatabaseDevice,systemd,DatabaseService,up
```

Updating the status of a service

Using Inventory REST API:

```
PUT /inventory/managedObjects/<serviceId>
```

```
Content-Type: "application/vnd.com.nsn.cumulocity.managedObject+json"
```

```
{
  "status": "down"
}
```

Field	Mandatory	Details
status	Yes	'up', 'down', 'unknown' or any arbitrary string specifying the service status

Or using [SmartREST static template 104](#) sent to topic `s/us/<serviceId>` :

`104, down`

Sending service data

Measurement REST API:

```
POST /measurement/measurements
Content-Type: "application/vnd.com.nsn.cumulocity.measurement+json"
```

```
{
  "source": {
    "id": "<serviceManagedObjectId>"
  },
  "time": "2020-03-19T12:03:27.845Z",
  "type": "c8y_Memory",
  "c8y_Memory": {
    "allocated": {
      "unit": "MB",
      "value": 100
    }
  }
}
```

Or using [SmartREST static template 200](#) sent to topic `s/us/<serviceId>` :

`200, c8y_Memory, allocated, 100, MB`

Similarly to measurements, alarms and events associated with the service can also be sent.

SERVICE COMMANDS

A service can announce its capability to receive commands by adding the `c8y_ServiceCommand` operation in its `c8y_SupportedOperations` . This allows the service to execute predefined or custom commands sent from the platform.

Available service commands

The list of commands a service supports is defined in the `c8y_SupportedServiceCommands` fragment. If this fragment is not present, the system assumes a default set of commands: `START` , `STOP` , and `RESTART` .

```
{
  "c8y_SupportedServiceCommands": [
    "START",
    "STOP",
    "SAVE_SNAPSHOT"
  ]
}
```

Name	Type	Mandatory	Description
<code>c8y_SupportedServiceCommands</code>	Array	No	List of available commands for the service. If not provided, defaults to <code>START</code> , <code>STOP</code> , <code>RESTART</code> .

Executing service commands

When a command is sent, the following operation structure is created:

```
{
  "deviceId": "<serviceManagedObjectId>",
  "c8y_ServiceCommand": {
    "command": "START",
    "serviceName": "Nginx Web Server",
    "serviceType": "systemd"
  }
}
```

The device is expected to perform the following actions:

1. Set operation status to EXECUTING
2. Find the service based in the deviceId, the serviceName, and the serviceType included in the operation
3. Execute the service command with the found service
4. Set operation status to SUCCESSFUL

TRACKING

The **Tracking** tab allows the device's path to be visualized. To achieve this the device must periodically send tracking events and update its position.

TRACKING POSITION HISTORY

A device may upload its current or past positions as location updated events with the `c8y_Position` fragment. This is useful for tracking devices on a route or generally tracing the location history of devices.

POST /event/events

```
{
  "c8y_Position": {
    "alt": 67,
    "lng": 6.95173,
    "lat": 51.151977
  },
  "time": "2013-06-22T17:03:14.000+02:00",
  "source": {
    "id": "10300"
  },
  "type": "c8y_LocationUpdate",
  "text": "LocUpdate"
}
```

Field	DataType	Mandatory	Details
c8y_Position	object	Yes	Holds geographical location properties
c8y_Position.alt	number	No	Optional altitude of the device position in meters
c8y_Position.lng	number	Yes	Longitude of the device position in degrees
c8y_Position.lat	number	Yes	Latitude of the device position in degrees
time	string	Yes	The time the position was measured
source	object	Yes	ID of the source device
type	string	Yes	Must always be c8y_LocationUpdate to designate this event as location update event
text	string	Yes	Description of the event. This parameter is required because it is a required for events in general. There are no further semantics applied to the text in the context of tracking devices.

SmartREST example

Cumulocity provides the 401 static template to send location update events using SmartREST:

401,51.151977,6.95173,67,2013-06-22T17:03:14.000+02:00

SETTING THE CURRENT POSITION

The current device position is represented using the `c8y_Position` fragment in the device's own managed object.

```
PUT /inventory/managedObjects/<deviceId>

{
  "c8y_Position": {
    "alt": 67,
    "lng": 6.95173,
    "lat": 51.151977
  }
}
```

Field	DataType	Mandatory	Details
alt	number	No	Optional altitude of the device position in meters
lng	number	Yes	Longitude of the device position in degrees
lat	number	Yes	Latitude of the device position in degrees

SmartREST example

Devices may update their current position using the 112 static template:

112,51.151977,6.95173,67,

In Cumulocity tracking over time is done using events and the current position is represented as fragment in the device's own managed object (see above for details). In practice devices usually determine their current position in an interval and upload the coordinates as they are measured. In this case the current position and the position at that time are the same and can be uploaded at the same time using 402 static template.

402,51.151977,6.95173,67,,

CORE MQTT

The Core MQTT implementation of Cumulocity provides the following benefits:

- Multi-tenancy support: A single endpoint serves multiple tenants.
- Device identity management: Devices authenticate using device-specific credentials.
- Device registration: Non-personalized devices can be deployed by pairing them with Cumulocity tenants.
- Device management: Rich, pre-defined device management payload formats to enable out-of-the-box management of millions of devices.
- Standard IoT payload formats: Pre-defined payload formats to support IoT sensor readings, alarm management, remote control and device hierarchies.
- Custom payload formats: Additional payload formats can be added.
- Minimum traffic overhead.
- Processing modes: Control whether data is persisted in Cumulocity database, transiently passed to real-time processing, processed using quiescent mode which ensures that real-time notifications are disabled or is processed using CEP mode that ensures data is transiently sent to real-time processing engine only with real-time notifications disabled.
- Full bi-directional communication.
- MQTT over WebSockets support.
- TLS support.
- Full horizontal scalability.

The Core MQTT capability of the Cumulocity platform allows MQTT devices to send messages directly into Cumulocity, provided that the device implements the pre-defined topic schema and payload formats of Core MQTT. To integrate MQTT devices that do not support the specific Cumulocity protocol, a tenant must implement a mapping between the device protocol and the Cumulocity API. This can be done using a microservice integrated with the Cumulocity [MQTT Service](#), or with an external [agent](#).

Also see our [SmartREST documentation](#).

This documentation does not describe the basics of MQTT communication. If you are unfamiliar with MQTT, we recommend you to consult one of the numerous introductions on the internet. Some references can be found on the [MQTT website](#).

INFO

For all Core MQTT connections to the platform, the maximum accepted payload size is 16184 bytes (16KiB), which includes both message header and body. The header size varies, but its minimum is 2 bytes.

INTEGRATION LIFECYCLE

The basic lifecycle for integrating devices into Cumulocity is discussed in [Interfacing devices](#).

In this section, we will show how this lifecycle can be managed using the MQTT implementation.

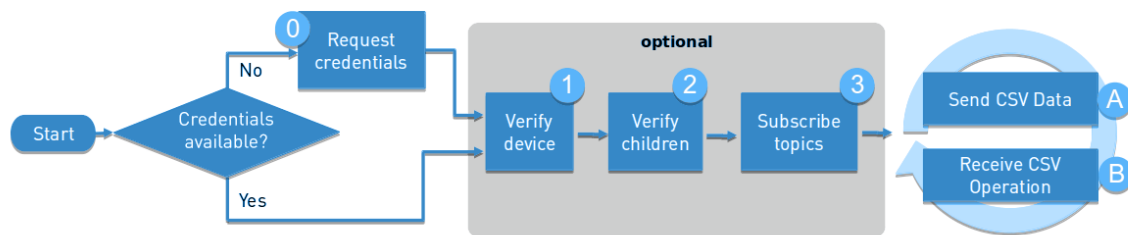
The lifecycle consists of two phases, a startup phase and a cycle phase.

The [startup phase](#) can be as short as just checking the credentials:

- [Step 0](#): Request device credentials, if they have not been requested yet.
- [Step 1](#): Ensure that the device exists.
- [Step 2](#): Ensure that the device children exist.
- [Step 3](#): Subscribe to the topics.

The [cycle phase](#) consists of two kinds of actions:

- [Step A](#): Send CSV data
- [Step B](#): Receive CSV operations



✓ REQUIREMENTS

- Access via bootstrap user credentials. See [Device credentials](#) for more information on how to obtain and use bootstrap credentials.

STARTUP PHASE

Step 0: Request device credentials

In Cumulocity, every MQTT connection must be authenticated. You can use the device credentials topics in the MQTT implementation to generate new credentials for a device.

Once the device retrieved the credentials, it needs to store them locally for further connections.

To establish a connection you must configure the following parameters:

- Host: <your_cumulocity_url>
- User: <tenantID>/<username> (user alias is not supported)
- Password: <your_cumulocity_password>

For more information, refer to the [Hello MQTT](#) section.

The process works as follows:

- Cumulocity assumes that each device has some form of unique ID. For instance, a good device identifier can be the MAC address of the network adapter, the IMEI of a mobile device or a hardware serial number.
- When you take a new device into use, you enter this unique ID into **Device registration** in the Device Management application in Cumulocity, and start the device.
- The device will use this ID as part of the [MQTT ClientId](#) and static user credentials that can be enquired from [product support](#).
- The device subscribes to the topic `s/dcr`.
- The device starts publishing continuous empty messages on the topic `s/ucr` to notify the server that it is ready to retrieve credentials.
- Next, you must accept the connection from the device in the **Device Registration** page.
- When the device sends the next empty message it should receive credentials in the format `70,<tenantID>,<username>,<password>`.

After receiving the credentials, the device can close the MQTT connection and create a new one with the received credentials.

Step 1: Verify device

As MQTT supports an automatic device creation if the client sends data and there is no device present, this step is only required if you want to create the device manually.

The device creation can be achieved by employing the [static template 100](#). This template can be blindly used on every boot of the device as it will only create the device if it is not already present.

The device will be linked automatically to the ID the client uses with its MQTT ClientId.

```
100,Device Name,Device Type
```

INFO

The topic used for Cumulocity's pre-provided static templates is `s/us`.

Step 2: Verify children

Like the root device, also its children are covered by the automatic device creation.

To handle this step manually you can send the [static template 101](#) for creating a child device. The template will only create the child if it does not already exist.

```
101,Unique Child ID,Child Name,Child Type
```

Step 3: Subscribe topics

If the device supports operations, it should subscribe to all required topics (static templates and SmartREST 2.0).

CYCLE PHASE

Step A: Send CSV data

While the device holds an active MQTT connection, it can publish either on the topics for static templates or on the topics for a SmartREST template to send data to the server.

Based on the MQTT ClientId, the physical device is directly connected to the device object in Cumulocity. Therefore, the data you send is automatically connected to the device.

To send data to a child device, publish the data to the topics described in [Device hierarchies](#).

Step B: Receive CSV operations

By subscribing to a topic the device automatically tells Cumulocity that it wants to receive operations. Any operation created will be automatically parsed using either the static templates or the templates the device defines.

MQTT IMPLEMENTATION

This section lists the implementation details for the MQTT protocol. The Cumulocity implementation supports MQTT Version 3.1.1.

CONNECTING VIA MQTT

Cumulocity supports MQTT both via TCP and WebSockets. As URL you can use the domain of the instance in the format `mqtt.<instance_domain>` (for example `mqtt.cumulocity.com`) or your tenant domain (for example `mytenant.cumulocity.com/mqtt`).

Available ports:

	TCP	WebSockets
SSL	8883	443
no SSL	1883	80

INFO

Port 80 is deactivated in cloud systems.

Port 8883 supports two types of SSL: two-way SSL using certificates for client authorization and one-way SSL using username and password for client authorization. The two-way SSL support is enabled by default. To disable it please contact [product support](#).

INFO

To use WebSockets you must connect to the path `/mqtt` and follow the [MQTT standard](#) for WebSocket communication.

SMARTREST PAYLOAD

The Cumulocity MQTT implementation uses SmartREST as a payload. SmartREST is a CSV-like message protocol that uses templates on the server side to create data in Cumulocity. It incorporates the highly expressive strength of the REST API but replaces JSON with comma-separated values (CSV) to avoid the complexity of JSON parsing for embedded devices. Additionally, the simple and compact syntax of CSV renders it highly efficient for IoT communication via mobile networks. It can save up to 80% of mobile traffic compared to other HTTP APIs.

INFO

For all MQTT connections to the platform, the maximum accepted payload size is 16184 bytes, which includes both message header and body. The header size varies, but its minimum is 2 bytes.

SmartREST basics

A SmartREST message is a single row in which each parameter is separated by comma. The first parameter is an ID that defines the message. You can send multiple messages in a single publish by using a line break between messages.

SmartREST escaping

The CSV (comma-separated values) format is used for communication with the SmartREST endpoint. The following rules must be followed to ensure a frictionless communication.

- Every row must be terminated by the `\n` character sequence.
- Values are always separated by a comma (`,`).
- If a value contains double-quotes (`"`), commas (`,`), leading or trailing whitespaces, line-breaks (`\n`), carriage returns (`\r`) or tab stops, it must be surrounded by quotes (`"`). Contained double-quotes (`"`) must be escaped by prepending a backslash (`\`).

The same escaping rules apply to messages that will be sent from the server to the client.

Publish example:

```
100,"This value, needs escaping",This value does not need escaping
```

Subscribe example:

```
511,myDeviceSerial,"execute this\nand this\nand \"this\""
```

INFO

`\n` does not create a new line in the output (for example console, UI); to achieve this, a new line character (ASCII 0A) must be used.

DEVICE HIERARCHIES

MQTT sessions are linked to a single device, but this device can have a freely configurable device hierarchy below it.

All children require a unique ID defined when creating the device. We recommend you to use a combination of the unique ID of the root device and a unique ID within the hierarchy.

To create data for a child instead of the root device, the unique ID of the child is added as another section in the topic (for example `s/us/myChildDeviceIdentifier`).

The client will automatically receive operations for every child in the hierarchy by subscribing to the respective topic. It is not required to subscribe for each child.

Every operation received will contain the template ID followed by the ID of the device/child for which the operation was created (followed by other operation parameters).

MQTT FEATURES

MQTT authentication

The communication with Cumulocity employing MQTT supports authentication in two ways:

- Username and password. The MQTT username must include the tenant ID and username in the format `<tenantID/username>`.
- Device certificates. For secure communication, devices must contain the entire chain of certificates leading to the trusted root certificate, or if only the device certificate is provided, then the immediate issuer certificate must be uploaded to the platform's truststore. Also, they must contain the server certificate in their truststore.

Troubleshooting

A device sends correct username and password, but incorrect certificate at the same time

If the platform is configured to support two-way SSL, your devices have a configured keystore with invalid certificates, and you want to use basic authorization, we recommend you to turn off sending certificates during connection. Certificates may be invalid because they expired or the root certificate is not uploaded to the platform. Turn off certificate sending in the device's software. If that is not possible, to make the connection work, check the following:

- The platform's trust store cannot be empty. At least one trusted certificate must be uploaded to the platform.
- The device's MQTT client must be configured to not send certificates if it does not find its root certificate in the accepted issuers list returned by the server during handshake. In most cases this happens automatically. It is known that it's not working with the MQTT client and Java 11. However, it works with Java 8.
- In order to support this situation, the platform must be configured accordingly. In case you experience issues please contact [product support](#).
- If all of the cases above are met and the device connection is still rejected due to certificates validation, then probably some other tenant uploaded a certificate with the same 'Common Name' as one of those sent by your device. In this case the device will always try to authorize itself with certificates.

MQTT ClientId

The MQTT ClientId is a field to uniquely identify each connected client. The Cumulocity implementation also uses the ClientId to link the

client directly to a device. Therefore, the following format should be used for the ClientId:

```
connectionType:deviceIdIdentifier:defaultTemplateIdentifier
```

Field	Mandatory	Description
connectionType	NO	Indication of connection type default: d (device)
deviceIdIdentifier	YES	A unique identifier for your device, for example, IMEI, Serial number
defaultTemplateIdentifier	NO	Check MQTT static templates for more information about template identifiers

For the simplest version of a client, the MQTT clientId can just be the `deviceIdIdentifier`. It will automatically be interpreted as device connection.

! IMPORTANT

The colon character has a special meaning in Cumulocity. Hence, it must not be used in the `deviceIdIdentifier`.

Examples of ClientIds:

```
mySerialNumber
d:mySerialNumber
d:mySerialNumber:myDefaultTemplate
```

The uniqueness of the MQTT ClientId is determined only by the `deviceIdIdentifier`. Therefore, from the above examples only one client can be connected at the same time.

During an SSL connection with certificates, the `deviceIdIdentifier` must match the 'Common Name' of the used certificate (first certificate in the chain, which is provided by the device).

MQTT Quality of Service (QoS)

The Cumulocity implementation supports all 3 levels of MQTT QoS:

- QoS 0: At most once
 - The client just sends the message once (fire and forget).
 - No reaction from the server.
- QoS 1: At least once
 - The client repeats the message until it receives a server acknowledgement.
- QoS 2: Exactly once
 - The client sends a message.
 - The server acknowledges (holds the message).
 - The client sends a release command.
 - The server processes the messages and acknowledges again.

For subscriptions to the operation or error topics, we will deliver all messages in the QoS which the client defined when subscribing to the topic.

MQTT clean session

Cumulocity requires clean session to be set to "1" (true). Currently we cannot guarantee that disabling clean session will work reliably, hence we recommend you to always enable clean session.

MQTT retained flag

In the current Cumulocity implementation, subscriptions to topics where devices publish data are not allowed. Publishing data with the retained flag on this topic is allowed but has no practical difference to sending it without the flag. Messages published by Cumulocity like operations and errors do not contain the retained flag.

MQTT last will

In MQTT, the "last will" is a message that is specified at connection time and that is executed when the client loses the connection. For example, using `400,c8y_ConnectionEvent,"Device connection was lost."` as last will message and `s/us` as last will topic, raises an event whenever the device loses the connection.

INFO

The execution of the "last will" updates the device availability.

MQTT RETURN CODES

When there is an MQTT error, the platform responds with a `CONNACK` message with a non-zero return code. This message is the first clue that there is a problem. Such a return code can be treated similarly to REST API HTTP codes, such as 401. They can be returned because of an unexpected error, lack of permissions, and so on.

`CONNACK` is not only a response to a `CONNECT` message, but also a way to signal errors that occurred in the platform. Therefore, it is possible to receive this message a second time during a normal connection, and without a direct action. It is also a way to signal a closing connection, as most MQTT clients treat `CONNACK` with a code other than `0` like the connection needs to be closed. See the details below.

The table below shows the list of errors returned by Cumulocity:

Code	Canonical message	Troubleshooting
0	Connection accepted	No issue, connection is working.
1	Connection refused, unacceptable protocol version	Unsupported version of the MQTT protocol. Currently, Cumulocity only allows 3.1 and 3.1.1.
2	Connection refused, identifier rejected	ClientId is not accepted by the platform.
3	Connection refused, Server unavailable	General platform side error, used on internal errors and unknown authorization problems. Can be received on network issues. The error should be temporary and independent of device state, therefore the usual solution to this is to try again later.
4	Connection refused, bad username or password	Incorrect credentials (wrong username and/or password, but not on empty password). This error is never returned when authenticating with certificates.

Code	Canonical message	Troubleshooting
5	Connection refused, not authorized	<p>Mostly a device side related problem, used when the device doesn't have permissions or is doing something forbidden. For example, if the client sends malformed messages or tries to execute an operation without authenticating first, such as publishing a message.</p> <p>Thrown on any issue with certificate authentication (for example, wrong common name, failed auto registration).</p> <p>Also thrown on general issues with receiving device data or some other authorization problem related to the device state on the platform. For example, device managed object problems, or the sudden removal of permissions. In this situation it may be required to take action on the platform to investigate and apply a fix.</p> <p>When clientId is too long the user can receive this error when using 3.1 version of MQTT. This can happen if clientId has 24 characters or more.</p> <p>Lastly, it can also be thrown on unexpected exceptions like performance issues, especially during connection. Therefore it is a good approach to repeat the connection a few times to overcome temporary performance issues.</p>

Refer to [MQTT Version 3.1.1 > 3.2 CONNACK - Acknowledge connection request](#) for details on the official MQTT connection return codes.

DEBUGGING

To support developers during development, it is possible to subscribe to the topic `/s/e`. On this topic the device can retrieve debug and error messages that occur during a publish from the device.

INFO

This topic is purely designed to support the development of clients. It is not recommended to always subscribe to this channel as the messages are verbose and can significantly increase the data usage. Also, you should not use this topic to trigger actions of the device based on what you receive on the topic. It is not a response channel.

MQTT BROKER CERTIFICATES

MQTT broker uses the certificates which are assigned to the main environment domain. MQTT broker always sends these certificates during TLS handshake to devices. Moreover, Enterprise tenants are not able to customize MQTT broker certificates via the SSL Management feature.

MQTT JWT SESSION TOKEN RETRIEVAL

The code of the Cumulocity MQTT example client implemented in Java, which connects to the platform using x.509 certificates, is available here: <https://github.com/Cumulocity-IoT/cumulocity-examples/tree/develop/mqtt-client>. This example client uses the implementation of Eclipse Paho, which is described in detail on their website: <https://www.eclipse.org/paho/index.php?page=documentation.php>.

Here is an example that shows how to add the needed dependency in Maven to use the Eclipse Paho client:

```
<dependency>
  <groupId>org.eclipse.paho</groupId>
  <artifactId>org.eclipse.paho.client.mqtty3</artifactId>
  <version>${paho.version}</version>
</dependency>
```

Then the instance of the MQTT client can be created with a single line:

```
MqttClient mqttClient = new MqttClient(BROKER_URL, "d:" + CLIENT_ID, new MemoryPersistence());
```

The `BROKER_URL` must contain the protocol, URL and port to which the client will connect, like this: `ssl://<cumulocity url>:8883`. The `CLIENT_ID` value must match the value of the common name of the device certificate that will be used. The certificate's common name should not contain `:` characters, see [MQTT ClientId](#) for more information. The `"d:"` prefix is used in Cumulocity for device connections and it should not be removed or changed. Now the only thing that must be configured to establish the SSL connection is to fill paths in the code fragment:

```
sslProperties.put(SSLSocketFactoryFactory.KEYSTORE, getClass().getClassLoader().getResource(KEYSTORE_NAME).getPath());
sslProperties.put(SSLSocketFactoryFactory.KEYSTOREPWD, KEYSTORE_PASSWORD);
sslProperties.put(SSLSocketFactoryFactory.KEYSTORETYPE, KEYSTORE_FORMAT);
sslProperties.put(SSLSocketFactoryFactory.TRUSTSTORE, getClass().getClassLoader().getResource(TRUSTSTORE_NAME).getPath());
sslProperties.put(SSLSocketFactoryFactory.TRUSTSTOREPWD, TRUSTSTORE_PASSWORD);
sslProperties.put(SSLSocketFactoryFactory.TRUSTSTORETYPE, TRUSTSTORE_FORMAT);
```

- The certificate's common name should not contain `:` characters, see [MQTT ClientId](#) for more information.
- `KEYSTORE_NAME` - The path to your keystore which contains the private key and the chain of certificates, which the device uses to authenticate itself.
- `KEYSTORE_PASSWORD` - The password created for keystore to use its private key.
- `KEYSTORE_FORMAT` - Either `"JKS"` or `"PKCS12"` depending on the file format. The path is provided by `KEYSTORE_NAME`.
- `TRUSTSTORE_NAME` - The path to your truststore which contains the certificate of the server.
- `TRUSTSTORE_PASSWORD` - The password to access the truststore.
- `TRUSTSTORE_FORMAT` - Either `"JKS"` or `"PKCS12"` depending on the file format. The path is provided by `TRUSTSTORE`.

After filling in this data, the example client will use the provided data to connect to the specified platform using certificates. The example also shows how to create the callback for the connection. First thing is to create the class which implements the interface `MqttCallbackExtended`. Then such a class can be created and an instance of it can be provided to the MQTT client:

```
mqttClient.setCallback(this);
```

In general, the MQTT Eclipse Paho client uses the Java Secure Socket Extension, which is part of the Java Development Kit, to provide secure connections via SSL. JSSE provides the Java implementation of the SSL and TLS protocol, which can be configured by developers using its classes. The documentation of the Java Secure Socket Extension shows how the SSL connection is established and provides some examples of customizing the implementation. The full document is available on the [official Oracle website](#).

MQTT EXAMPLES

HELLO MQTT

In this tutorial, you will learn how to use MQTT with Cumulocity using pre-defined messages (called "static templates").

Prerequisites

In order to follow this tutorial, check the following prerequisites:

- You have a valid tenant, a user and a password in order to access Cumulocity.
- You have installed [MQTTBox](#) or a similar MQTT tool.

INFO

- The screenshots in the tutorial use MQTTBox. Other tools may look different.
- If you are using a trial tenant, the default user will not work with this tutorial. Create an additional user instead. The tenant ID and URL data will also differ from trial tenant information.

Configuring the MQTT connection

To configure the MQTT connection, you must pass the following connection parameters (see the screenshot below).

- MQTT Client Name – Give your client a name to identify it, for example, Cumulocity MQTT.
- MQTT Client Id – You can use the “Generate a random ID” button (most tools will offer such a button) or provide one yourself. This ID will be linked to your device in Cumulocity. To reconnect to the same device, use the same ID.
- Protocol – Select the protocol to be used, for example, mqtt/tcp.
- Host – Provide in the URL your tenant domain, for example, *mytenant.cumulocity.com/mqtt*.
- Username – In this case, the username is formed as <tenantID>/<service-user>. You can use the same credentials you use to log into the Cumulocity platform (user alias is not supported). As seen in the example below, for the tenant ID “t76543210” and service user “manga” the username is “t76543210/manga”.
- Password: The password of the service user.

Cumulocity supports MQTT both via TCP and WebSockets. As URL you can use your tenant domain (for example *mytenant.cumulocity.com/mqtt*) or the domain of the instance in the format *mqtt.<instance_domain>* (for example *mqtt.cumulocity.com*).

MQTT CLIENT SETTINGS [Client Settings Help](#)

MQTT Client Name <input type="text" value="Cumulocity MQTT"/>	MQTT Client Id <input type="text" value="my_mqtt_cs_client_mqttbox"/> Generate	Append timestamp to MQTT client id? <input type="checkbox"/> No	Broker is MQTT v3.1.1 compliant? <input checked="" type="checkbox"/> Yes
Protocol <input type="text" value="mqtt / tcp"/>	Host <input type="text" value="mytenant.cumulocity.com/mqtt"/>	Clean Session? <input checked="" type="checkbox"/> Yes	Auto connect on app launch? <input checked="" type="checkbox"/> Yes
Username <input type="text" value="t76543210/user"/>	Password <input type="password" value=""/>	Reschedule Pings? <input checked="" type="checkbox"/> Yes	Queue outgoing QoS zero messages? <input checked="" type="checkbox"/> Yes
Reconnect Period (milliseconds) <input type="text" value="1000"/>	Connect Timeout (milliseconds) <input type="text" value="30000"/>	KeepAlive (seconds) <input type="text" value="10"/>	
Will - Topic <input type="text" value="Will - Topic"/>	Will - QoS <input type="text" value="0 - Almost Once"/>	Will - Retain <input type="checkbox"/> No	Will - Payload <input type="text"/>

[Save](#) [Delete](#)

INFO

You may review [Tenants > Tenant ID and tenant domain](#) in the Cumulocity OpenAPI Specification to get a better understanding between tenant ID and tenant domain.

Other configurations like “clean session” are not important for this example. You can change them to your needs. After clicking **Save**, you will see a screen similar to the following screenshot.

Menu

←

Connected

Add publisher

Add subscriber

⚙

Topic to publish

Topic to publish

QoS

0 - Almost Once

Retain ☐

Payload Type

Strings / JSON / XML / Characters

e.g: {"hello":"world"}

Payload

Publish

Topic to subscribe

Topic to subscribe

QoS

0 - Almost Once

Subscribe

If there is a blue button on the top bar with a label **Not Connected**, verify your configuration (especially username and password). If the button is green, you successfully established an MQTT connection to Cumulocity.

Sending data

All MQTT publish messages in this tutorial will be sent to the topic `s/us`. This is the topic used for Cumulocity's pre-provided static templates.

Menu

←

Connected

Add publisher

Add subscriber

⚙️

Topic to publish

s/us

QoS

0 - Almost Once

Retain ☐

Payload Type

Strings / JSON / XML / Characters

e.g: {'hello':'world'}

Payload

Publish

Create the device

The first message sent will create our device. Although the static templates support automatic device creation, in this example we will create the device manually. The template 100 will create a new device. It can be used with two optional parameters (deviceName, deviceType).

```
100,My first MQTT device,c8y_MQTTdevice
```

Afterwards, you will find this device in the Device Management application as a new device. If you switch to the **Identity** tab of the device you will notice that there was an identity created automatically to link the device to the MQTT ClientId.

Besides the name and the type, the device does not have more information, so master data must be added.

You can use multiple static templates per publishing separated by a line break (one template per row). This feature is used to set the hardware and the required interval for the device in a single published message.

The hardware can be set with the template 110. It can take 3 parameters (serialNumber, model, revision). Optional parameters in static templates can be left empty if you don't want to set them. For the hardware all parameters are optional.

The required interval can be set with the template 117 and just takes a single parameter (the interval in minutes).

```
110,,MQTT test model,1.2.3
117,10
```

After a reload of the **Info** page of your device in the Device Management application, you should see the information we just added.

Create measurements

Now the device has some master data and we can start sending some measurements. There are a couple of measurements that can be created directly by using a static template:

- 210: Signal strength measurement
- 211: Temperature measurement
- 212: Battery measurement

The temperature and battery measurement just take the value and time as parameters. For the signal strength, you can pass two values (RSSI and BER).

Passing timestamps in the Cumulocity MQTT implementation is always optional. If you don't pass them along, the server will automatically create a timestamp with the current server time.

We will make use of this feature in this example. Also, if you do not set the last parameters, you do not need to enter the remaining commas.

```
210,-87
211,24
212,95
```

Besides the measurements above, we can also use the template `200` to create a more custom measurement. It will take the measurement fragment, series, value, unit and time as its parameters.

```
200,myCustomTemperatureMeasurement,fahrenheit,75.2,F
```

After a reload in the Device Management application, you should see 4 graphs with the newly added measurements in the **Measurements** tab of your device.

Create alarms

Now we will create some alarms for this device. There are templates to create alarms for the 4 alarm severities:

- 301: CRITICAL
- 302: MAJOR
- 303: MINOR
- 304: WARNING

Each of them note a type (which is mandatory), a text and a time (both optional).

```
301,gpio_critical,There is a GPIO alarm
304,simple_warning
```

The alarm list of your device should now contain one critical alarm and one warning.

Note that we did not set any text for the warning, so it was created with a default alarm text.

Now we will clear the critical alarm again. To achieve this, we use the template `306` which refers to the type of the alarm that should be cleared.

```
306,gpio_critical
```

The critical alarm should be cleared afterwards.

Note that you did not have to handle any alarm IDs with the MQTT implementation. Cumulocity will take over this part so that the device communication can be as easy as possible.

Create events

Next, we will create some location events for the device. If you wish, you may use the [LatLong website](#) to get the latitude and longitude of your city.

The template `401` lets you create location events and takes latitude, longitude, altitude, accuracy and the time as parameters, but for now we will just use the first two.

```
401,51.227741,6.773456
```

In the Device Management application, you can see one event in the event list but the location has not been updated. This is because on REST these are different requests. Instead of the template `401`, you can use the template `402` in MQTT. It works exactly the same as `401` but additionally it also updates the position of the device itself.

```
402,51.227741,6.773456
```

Now you should see both the **Location** and the **Tracking** tab in the device with the **Location** tab having the same latitude and longitude as the last location event.

Receiving data

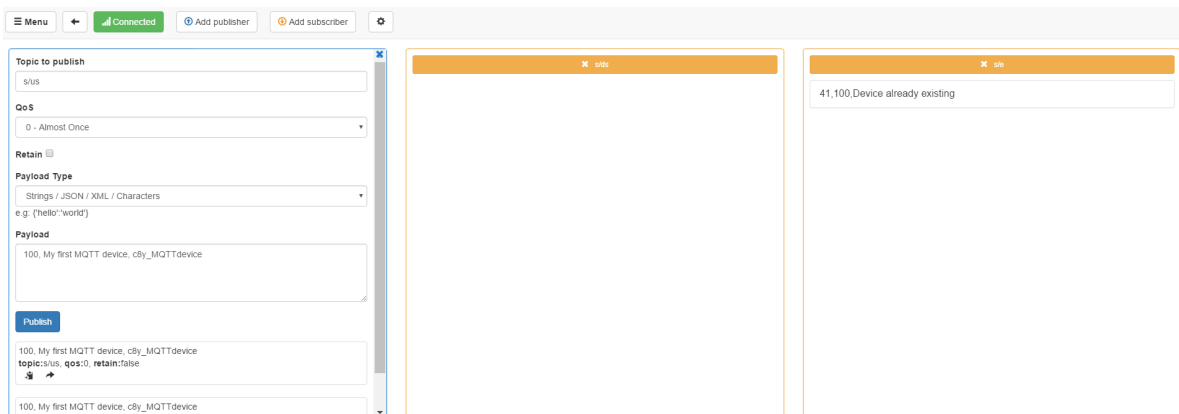
So far we have only used MQTT to send data from the client to the server. Now we will send data from the server to the client.

To achieve this, we must first subscribe to the responsible topic. We will do two subscriptions:

- `s/ds` : This will subscribe to the static operation templates for the device
- `s/e` : This will subscribe to an error topic that can be used for debugging

You can enter both topics after another in the **Subscribe** field and click **Subscribe**. The QoS selection does not matter for this example.

Afterwards, your MQTTBox should look like this:



Receive operations

At the current state, the UI does not show any tabs for operations. Up to this point, it was unknown what exactly the device supports, but the list of supported operations can be modified with the template `114`. A list of supported operations can be added here.

We will add support for the configuration and shell.

```
114,c8y_Command,c8y_Configuration
```

After reloading the UI, the two new tabs will appear (**Configuration** and **Shell**).

We can now create a shell command from the UI and click **Execute**.

In the MQTTBox, you should now have received a new message for the `s/ds` subscription.

The screenshot shows the MQTT client interface. On the left, the 'Publish' configuration panel is visible with the following settings:

- Topic to publish:** s/us
- QoS:** 0 - Almost Once
- Retain:** ☐
- Payload Type:** Strings / JSON / XML / Characters
- Payload:** 114,c8y_Command,c8y_Configuration

Below the configuration panel, a 'Publish' button is shown. The message history window on the right displays a message with the following content:

```
511,757e2355-75a8-4889-a232-44665443cfb11519038253508,Example_Command
```

The `511` is indicating what kind of operation we received (in this case `c8y_Command`). This will be followed by the **deviceIdentifier** to locate the device with the dedicated operation. This is required if you have a hierarchy with multiple children. In such case, you must know for which of the children the operation was dedicated. Finally, you have the operation specific parameters, which in the case of `c8y_Command` is only the command text.

After receiving the operation, we can start executing it to initiate the client's handling the operation. Similar to changing the status of an alarm, you can add the type of operation to the template.

```
501,c8y_Command
```

After completing the handling, the operation can be set to successful with the template `503`.

Besides the operation type, this operation can also take additional parameters based on what kind of operation it was. We can return a result for the `c8y_Command`.

```
503,c8y_Command,Everything went fine
```

Learning from errors

The topic `s/e` can help you debugging in case something went wrong. For instance, if we try to send

```
999,I made this up
```

we can see a message on the topic because the template `999` is unknown.

```
40,999,No static template for this message id
```

HELLO MQTT C

In this tutorial, you will learn how to use MQTT client in C with Cumulocity using pre-defined messages (called "static templates").

Prerequisites

In order to follow this tutorial, check the following prerequisites:

- You have a valid tenant, a user, and a password in order to access Cumulocity.
- Verify that you have a gcc compiler installed:

```
$ gcc --version
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

- Download, compile and install the MQTT C Paho Client. You will find more details about Paho on the [Paho website](#).

Developing the “Hello, MQTT world!” client

To develop a very simple “Hello, world!” MQTT client for Cumulocity, you must

- create the application,
- build and run the application.

Create the application

Create a source file (for example *hello_mqtt.c*) with the following content:

```

#include "stdlib.h"
#include "string.h"
#include "unistd.h"
#include "MQTTClient.h"

#define ADDRESS    "<<serverUrl>>"
#define CLIENTID   "<<clientId>>"

void publish(MQTTClient client, char* topic, char* payload) {
    MQTTClient_message pubmsg = MQTTClient_message_initializer;
    pubmsg.payload = payload;
    pubmsg.payloadlen = strlen(pubmsg.payload);
    pubmsg.qos = 2;
    pubmsg.retained = 0;
    MQTTClient_deliveryToken token;
    MQTTClient_publishMessage(client, topic, &pubmsg, &token);
    MQTTClient_waitForCompletion(client, token, 1000L);
    printf("Message '%s' with delivery token %d delivered\n", payload, token);
}

int on_message(void *context, char *topicName, int topicLen, MQTTClient_message *message) {
    char* payload = message->payload;
    printf("Received operation %s\n", payload);
    MQTTClient_freeMessage(&message);
    MQTTClient_free(topicName);
    return 1;
}

int main(int argc, char* argv[]) {
    MQTTClient client;
    MQTTClient_create(&client, ADDRESS, CLIENTID, MQTTCLIENT_PERSISTENCE_NONE, NULL);
    MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
    conn_opts.username = "<<tenant_ID>><<username>>";
    conn_opts.password = "<<password>>";

    MQTTClient_setCallbacks(client, NULL, NULL, on_message, NULL);

    int rc;
    if ((rc = MQTTClient_connect(client, &conn_opts)) != MQTTCLIENT_SUCCESS) {
        printf("Failed to connect, return code %d\n", rc);
        exit(-1);
    }
    //create device
    publish(client, "s/us", "100,C MQTT,c8y_MQTTDevice");
    //set hardware information
    publish(client, "s/us", "110,S123456789,MQTT test model,Rev0.1");
    //listen for operation
    MQTTClient_subscribe(client, "s/ds", 0);

    for (;;) {
        //send temperature measurement
        publish(client, "s/us", "211,25");
        sleep(3);
    }
    MQTTClient_disconnect(client, 1000);
    MQTTClient_destroy(&client);
    return rc;
}

```

Replace <<clientId>> , <<serverUrl>> , <<tenant_ID>> , <<username>> and <<password>> with your data.

The Cumulocity MQTT protocol supports both unsecured TCP and secured SSL connections (for example `tcp://mqtt.cumulocity.com:1883` or `ssl://mqtt.cumulocity.com:8883`), so as the <<serverUrl>> you can pick the one which fits for you. When using SSL remember to configure `MQTTClient_SSLOptions` and set it in the `MQTTClient_connectOptions` .

What does the code in `main` do?

- Configure an MQTT connection.
- Register a `on_message` callback function which will print incoming messages.
- Connect with Cumulocity via MQTT protocol.
- Create a new device with `C MQTT` name and `c8y_MQTTDevice` type.
- Update the device hardware information by putting a `"S123456789"` serial, a `"MQTT test model"` model and a `"Rev0.1"` revision.
- Subscribe to the static operation templates for the device - this will result in an `on_message` method call every time a new operation is created.
- Send temperature measurement every 3 seconds.

What does the code in `publish` do?

- Create a new MQTT message and set a payload.
- Publish message via MQTT protocol.
- Wait maximum 1 second for a message delivered ACK from the server.

Note that the subscription is established after the device creation, otherwise if there is no device for a given `clientId` the server will not accept it.

Build and run the application

To build the application, enter

```
$ gcc hello_mqtt.c -o hello_mqtt -lpaho-mqtt3c
```

To run the application, enter

```
$ ./hello_mqtt
Message '100,C MQTT,c8y_MQTTDevice' with delivery token 1 delivered
...
```

After starting the application, you should see a new device in the Device Management application, listed in **All devices**.

Additionally, if there will be a new operation created for this device (for example `c8y_Restart`), information about it will be printed to the console.

Improving the agent

Now that you have done your first step, check out the section [Hello MQTT](#) to learn more about Cumulocity MQTT and improve your application.

HELLO MQTT JAVA

In this tutorial, you will learn how to use the Java MQTT client with Cumulocity using pre-defined messages (called “static templates”).

Prerequisites

In order to follow this tutorial, check the following prerequisites:

- You have a valid tenant, a user and a password in order to access Cumulocity.
- Verify that you have Maven 3 and at least Java 7 installed.

```
$ mvn -v
Maven home: /Library/Maven/apache-maven-3.6.0
Java version: 1.8.0_201, vendor: Oracle Corporation, runtime: /Library/Java/JavaVirtualMachines/jdk1.8.0_201.jdk/Contents/Home/jre
Default locale: en_GB, platform encoding: UTF-8
OS name: "mac os x", version: "10.14.2", arch: "x86_64", family: "mac"
```

Maven can be downloaded from the [Maven website](#).

Developing the “Hello, MQTT world!” client

To develop a very simple “Hello, world!” MQTT client for Cumulocity, you must

- create a Maven project,
- add a dependency to the MQTT Java client library to the *pom.xml* (in this example we will use [Paho Java Client](#)),
- create a Java application,
- build and run the Java application.

Create a Maven project

To create a plain Java project with Maven, execute the following command:

```
$ mvn archetype:generate -DgroupId=c8y.example -DartifactId=hello-mqtt-java -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

This will create a folder *hello-mqtt-java* in the current directory with a skeleton structure for your project.

Add the MQTT Java client library

Edit the *pom.xml* in the *hello-mqtt-java* folder. Add a dependency to the MQTT Paho Java Client.

```
<dependency>
  <groupId>org.eclipse.paho</groupId>
  <artifactId>org.eclipse.paho.client.mqttv3</artifactId>
  <version>[1.2.1,)</version>
</dependency>
```

If you are using Java 9 or later, you must set the source and target as described at the [Apache Maven Compiler Plugin](#) page, adding the following code:

```
<properties>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

Create a Java application

Edit the *App.java* file located in the folder *hello-mqtt-java/src/main/java/c8y/example* with the following content:

```
package c8y.example;

import org.eclipse.paho.client.mqttv3.*;
import java.util.concurrent.*;

public class App {

    public static void main(String[] args) throws Exception {

        // client, user and device details
        final String serverUrl = "tcp://mqtt.cumulocity.com"; /* ssl://mqtt.cumulocity.com:8883 for a secure connection */
        final String clientId = "my_mqtt_java_client";
        final String device_name = "My Java MQTT device";
        final String tenant = "<<tenant_ID>>";
        final String username = "<<username>>";
        final String password = "<<password>>";

        // MQTT connection options
        final MqttConnectOptions options = new MqttConnectOptions();
        options.setUserName(tenant + "/" + username);
        options.setPassword(password.toCharArray());

        // connect the client to Cumulocity
        final MqttClient client = new MqttClient(serverUrl, clientId, null);
        ..
```



```

client.connect(options);

// register a new device
client.publish("s/us", ("100," + device_name + ",c8y_MQTTDevice").getBytes(), 2, false);

// set device's hardware information
client.publish("s/us", "110,S123456789,MQTT test model,Rev0.1".getBytes(), 2, false);

// add restart operation
client.publish("s/us", "114,c8y_Restart".getBytes(), 2, false);

System.out.println("The device '" + device_name + "' has been registered successfully!");

// listen for operations
client.subscribe("s/ds", new IMqttMessageListener() {
    public void messageArrived (final String topic, final MqttMessage message) throws Exception {
        final String payload = new String(message.getPayload());

        System.out.println("Received operation " + payload);
        if (payload.startsWith("510")) {
            // execute the operation in another thread to allow the MQTT client to
            // finish processing this message and acknowledge receipt to the server
            Executors.newSingleThreadScheduledExecutor().execute(new Runnable() {
                public void run() {
                    try {
                        System.out.println("Simulating device restart...");
                        client.publish("s/us", "501,c8y_Restart".getBytes(), 2, false);
                        System.out.println("...restarting...");
                        Thread.sleep(TimeUnit.SECONDS.toMillis(5));
                        client.publish("s/us", "503,c8y_Restart".getBytes(), 2, false);
                        System.out.println("...done...");
                    } catch (MqttException e) {
                        e.printStackTrace();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });
        }
    }
});

// generate a random temperature (10°-20°) measurement and send it every 7 seconds
Executors.newSingleThreadScheduledExecutor().scheduleWithFixedDelay(new Runnable() {
    public void run () {
        try {
            int temp = (int) (Math.random() * 10 + 10);

            System.out.println("Sending temperature measurement (" + temp + "°) ...");
            client.publish("s/us", new MqttMessage(("211," + temp).getBytes()));
        } catch (MqttException e) {
            e.printStackTrace();
        }
    }
}, 1, 7, TimeUnit.SECONDS);
}
}

```

Replace `serverUrl`, `clientId` and `device_name` as needed. Do not forget to specify the user credentials setting values for `tenant_ID`, `username` and `password`.

Cumulocity MQTT protocol supports both unsecured TCP and secured SSL connections (that is, `tcp://mqtt.cumulocity.com:1883` or `ssl://mqtt.cumulocity.com:8883`), so you can pick the one which fits for you and use it in `serverUrl`.

What does the code in `main` do?

- Configure the MQTT connection.
- Connect with Cumulocity via a MQTT protocol.
- Create a new device with a name (`device_name`) and a type (`c8y_MQTTDevice`).
- Update the device hardware information by putting a `"S123456789"` serial, a `"MQTT test model"` model and a `"Rev0.1"` revision.
- Subscribe to the static operation templates for the device and print all received operations to the console. In case of a `c8y_Restart` operation, simulate a device restart.
- Create a new thread which sends temperature measurement every 7 seconds.

Note that the subscription is established after the device creation, otherwise if there is no device for a given `clientId`, the server will not accept it.

Build and run the application

Use the following commands to build the application:

```
$ cd hello-mqtt-java
$ mvn clean install
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ hello-mqtt-java ---
[INFO] Building jar: /home/schm/Pulpit/hello-mqtt-java/target/hello-mqtt-java-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ hello-mqtt-java ---
[INFO] Installing /home/schm/Pulpit/hello-mqtt-java/target/hello-mqtt-java-1.0-SNAPSHOT.jar to /home/schm/.m2/repository/c8y/example/hello-mqtt-java/1.0-SNAPSHOT/hello-mqtt-java-1.0-SNAPSHOT.jar
[INFO] Installing /home/schm/Pulpit/hello-mqtt-java/pom.xml to /home/schm/.m2/repository/c8y/example/hello-mqtt-java/1.0-SNAPSHOT/hello-mqtt-java-1.0-SNAPSHOT.pom
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2.642 s
[INFO] Finished at: 2017-03-14T09:16:25+01:00
[INFO] Final Memory: 14M/301M
[INFO] -----
```

and this command to run it:

```
$ mvn exec:java -Dexec.mainClass="c8y.example.App"
...
[INFO]
[INFO] -----
[INFO] Building hello-mqtt-java 1.0-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ hello-mqtt-java ---
Received operation 510,123456789
```

After starting the application, you should see a new registered device in the Device Management application, listed in **All devices**. In the **Measurements** tab, you will see the temperature measurements being sent by your client.

Additionally, if there will be a new operation created for this device (for example `c8y_Restart`), information about it will be printed to the console.

Improving the agent

Now that you have done your first step, check out the section [Hello MQTT](#) to learn more about Cumulocity MQTT and improve your application.

HELLO MQTT JAVA WITH CERTIFICATES

In this tutorial, you will learn how to use the Java MQTT client with Cumulocity using X.509 certificates for authentication.

In the GitHub repository [cumulocity-examples](#), you can find a sample Java MQTT client using X.509 certificates and all necessary scripts

used in this tutorial.

Prerequisites

In order to follow this tutorial, check the following prerequisites:

- You have correctly configured the Java client based on the [Hello MQTT Java](#) tutorial.
- You have a valid tenant, a user and a password in order to access Cumulocity.
- You have a valid certificate. If you don't have it, follow the instructions in the next section to generate one.

To generate a valid certificate

If you don't have a valid certificate, you can generate one for testing purposes following the instructions in [Device certificates](#).

1. Download the scripts from the [cumulocity-examples](#) repository.
2. Create a root self-signed certificate (execute the script `00createRootSelfSignedCertificate.sh`) and upload it to your tenant. You can do it via [the Device Management application in the UI](#) or via [REST](#).
3. Create and sign the certificate (execute the script `01createSignedCertificate.sh`).
4. Move the certificates to keystore (execute the script `02moveCertificatesToKeystore.sh`). Additionally, if only the device leaf certificate is needed in the keystore, use the `-leafonly` option.
5. Finally, import the trusted certificate into keystore running the following command:

```
$ keytool -importcert -file c8y-mqtt-server.cer -keystore chain-with-private-key-iot-device-0001.jks -alias "Alias"
```

Developing the "Hello, MQTT world!" client with certificates

To develop a "Hello, world!" MQTT client for Cumulocity with certificates, you must

- copy the certificate and upload it to the platform,
- change the configuration in the MQTT client.

To copy and upload the certificate

Copy the certificate from the file `chain-iot-device-0001.pem` and upload it to the platform employing a POST request:

Endpoint: `/tenant/tenants/{tenantId}/trusted-certificates`

Authorization: Basic

Content-Type: `application/json`

Request body:

```
{
  "status": "ENABLED",
  "name": "sampleName",
  "autoRegistrationEnabled": "true",
  "certInPemFormat": "<<certificate in pem format>>"
}
```

To change the configuration

To change the configuration in the MQTT client, copy the file `chain-with-private-key-iot-device-0001.jks` into the resource folder and set the configuration. Note that the script employed (Step 4.) uses the password `changeit`. If you changed the value in the script, also do it for `KEYSTORE_PASSWORD` and `TRUSTSTORE_PASSWORD` in the following example.

```
// Configuration
private static final String KEYSTORE_NAME = "chain-with-private-key-iot-device-0001.jks";
private static final String KEYSTORE_PASSWORD = "changeit";
private static final String KEYSTORE_FORMAT = "jks";

private static final String TRUSTSTORE_NAME = "chain-with-private-key-iot-device-0001.jks";
private static final String TRUSTSTORE_PASSWORD = "changeit";
private static final String TRUSTSTORE_FORMAT = "jks";

private static final String CLIENT_ID = "iotdevice0001";
private static final String BROKER_URL = "<SSL URL of the platform>";

private MqttClient connect() throws MqttException {
    MqttClient mqttClient = new MqttClient(BROKER_URL, "d:" + CLIENT_ID, new MemoryPersistence());
    MqttConnectOptions options = new MqttConnectOptions();

    options.setCleanSession(true);

    Properties sslProperties = new Properties();
    sslProperties.put(SSLSocketFactoryFactory.KEYSTORE, getClass().getClassLoader().getResource(KEYSTORE_NAME).getPath());
    sslProperties.put(SSLSocketFactoryFactory.KEYSTOREPWD, KEYSTORE_PASSWORD);
    sslProperties.put(SSLSocketFactoryFactory.KEYSTORETYPE, KEYSTORE_FORMAT);
    sslProperties.put(SSLSocketFactoryFactory.TRUSTSTORE, getClass().getClassLoader().getResource(TRUSTSTORE_NAME).getPath());
    sslProperties.put(SSLSocketFactoryFactory.TRUSTSTOREPWD, TRUSTSTORE_PASSWORD);
    sslProperties.put(SSLSocketFactoryFactory.TRUSTSTORETYPE, TRUSTSTORE_FORMAT);
    sslProperties.put(SSLSocketFactoryFactory.CLIENTAUTH, true);

    options.setSSLProperties(sslProperties);
    mqttClient.setCallback(this);
    System.out.println("Connecting to the broker at " + BROKER_URL);
    mqttClient.connect(options);

    return mqttClient;
}
```

The device can now publish and subscribe as a standard device. Note that before the first connect no other actions are required, for example, creating a user. The user is created during the [auto registration](#) process.

INFO

You do not need to set a password, user or tenant for the MQTT client connecting using certificates. Cumulocity will recognize the tenant and the user by the provided certificate.

HELLO MQTT BROWSER-BASED

In this tutorial, you will learn how to use the browser-based MQTT client with Cumulocity using pre-defined messages (called "static templates").

Prerequisites

In order to follow this tutorial, check the following prerequisites:

- You have a valid tenant, a user, and a password in order to access Cumulocity.

Developing the "Hello, MQTT world!" client

To develop a very simple "Hello, world!" MQTT client for Cumulocity, you must

- create an HTML file and include the MQTT JavaScript client (in this example we will use [Paho JavaScript Client](#)),
- create a JavaScript application,

- run the application.

Create a JavaScript application

Create an HTML file (for example *hello_mqtt_js.html*) with the following content:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <title>Hello MQTT World</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <script src="https://cdnjs.cloudflare.com/ajax/libs/paho-mqtt/1.0.1/mqttws31.min.js"></script>
  <script src="main.js" defer></script>
</head>
<body>
  <div id="logger"></div>
</body>
</html>
```

Create a JavaScript file *main.js* with the following content:

```
// client, user and device details
var serverUrl = "ws://mqtt.cumulocity.com/mqtt"; /* wss://mqtt.cumulocity.com/mqtt for a secure connection */
var clientId = "my_mqtt_js_client";
var device_name = "My JS MQTT device";
var tenant = "<<tenant_ID>>";
var username = "<<username>>";
var password = "<<password>>";

var undeliveredMessages = [];
var temperature = 25;

// configure the client to Cumulocity
var client = new Paho.MQTT.Client(serverUrl, clientId);

// display all incoming messages
client.onMessageArrived = function (message) {
  log("Received operation " + message.payloadString + "");
  if (message.payloadString.indexOf("510") == 0) {
    log("Simulating device restart...");
    publish("s/us", "501,c8y_Restart");
    log("...restarting...");
    setTimeout(function () {
      publish("s/us", "503,c8y_Restart");
      log("...done...");
    }, 1000);
  }
};

// display all delivered messages
client.onMessageDelivered = function onMessageDelivered (message) {
  log("Message " + message.payloadString + " delivered");
  var undeliveredMessage = undeliveredMessages.pop();
  if (undeliveredMessage.onMessageDeliveredCallback) {
    undeliveredMessage.onMessageDeliveredCallback();
  }
};

function createDevice () {
  // register a new device
  publish("s/us", "100," + device_name + ",c8y_MQTTDevice", function () {
    // set hardware information
    publish("s/us", "110,S123456789,MQTT test model,Rev0.1", function () {
```

```

    publish('s/us', '114,c8y_Restart', function() {
        log('Enable restart operation support');
        //listen for operation
        client.subscribe("s/ds");
    })

    // send temperature measurement
    setInterval(function() {
        publish("s/us", '211,'+temperature);
        temperature += 0.5 - Math.random();
    }, 3000);
    });
});
}

// send a message
function publish (topic, message, onMessageDeliveredCallback) {
    message = new Paho.MQTT.Message(message);
    message.destinationName = topic;
    message.qos = 2;
    undeliveredMessages.push({
        message: message,
        onMessageDeliveredCallback: onMessageDeliveredCallback
    });
    client.send(message);
}

// connect the client to Cumulocity
function init () {
    client.connect({
        userName: tenant + "/" + username,
        password: password,
        onSuccess: createDevice
    });
}

// display all messages on the page
function log (message) {
    document.getElementById('logger').insertAdjacentHTML('beforeend', '<div>' + message + '</div>');
}

init();

```

Replace `serverUrl`, `clientId` and `device_name` as needed. Do not forget to specify the user credentials setting values for `tenant_ID`, `username` and `password`.

The Cumulocity MQTT protocol supports both unsecured TCP and also secured SSL connections (that is, `ws://mqtt.cumulocity.com/mqtt` or `wss://mqtt.cumulocity.com/mqtt`), so you can pick the one which fits for you and use it in `serverUrl`.

What does the code do?

- Configure the MQTT connection.
- Register `onMessageArrived` callback function which will display all incoming messages. In case of a `c8y_Restart` operation, simulate a device restart.
- Register `onMessageDelivered` callback function which will be called after a publish message has been delivered.
- After the page is fully loaded, the function `init` is called and it connects with Cumulocity via a MQTT protocol.
- When the connection is established, call a `createDevice` function.
- Create a new device with a name (`device_name`) and a type (`c8y_MQTTDevice`).
- Update the device hardware information by putting a `"S123456789"` serial, a `"MQTT test model"` model and a `"Rev0.1"` revision.
- Subscribe to the static operation templates for the device – this will result in `onMessageArrived` method call every time a new operation is created.
- Send a temperature measurement every 3 seconds.

Note that the subscription is established after the device creation, otherwise if there is no device for a given `clientId`, the server will

not accept it.

Run the application

Open the `hello_mqtt.js.html` file in a browser. You should see a new registered device in the Device Management application, listed in **All devices**. In the **Measurements** tab, you will see the temperature measurements being sent by your client.

Additionally, if there will be a new operation created for this device (for example `c8y_Restart`), related information will be displayed in the browser page.

Improving the agent

Now that you have done your first step, check out the section [Hello MQTT](#) to learn more about Cumulocity MQTT and improve your application.

HELLO MQTT NODE.JS

In this tutorial, you will learn how to use the Node.js MQTT client with Cumulocity using pre-defined messages (called “static templates”).

Prerequisites

In order to follow this tutorial, check the following prerequisites:

- You have Node.js and the package manager (npm) installed.
- You have a valid tenant, a user, and a password in order to access Cumulocity.

Developing the “Hello, MQTT world!” client

To develop a very simple “Hello, world!” MQTT client for Cumulocity, you must

- create a Node.js application,
- install the MQTT middleware (in this example we will use the library [MQTT.js](#)),
- run the application.

Create a Node.js application

Create the `package.json` file to list down the dependencies and other basic information about your application.

```
{
  "dependencies": {
    "mqtt": "*"
  },
  "scripts": {
    "start": "node app.js"
  }
}
```

Create the start script (`app.js`) specified in the `package.json` file with the following content:

```
// MQTT dependency https://github.com/mqttjs/MQTT.js
const mqtt = require("mqtt");

// client, user and device details
const serverUrl = "tcp://mqtt.cumulocity.com";
const clientId = "my_mqtt_nodejs_client";
const device_name = "My Node.js MQTT device";
const tenant = "<<tenant_ID>>";
const username = "<<username>>";
const password = "<<password>>";

var temperature = 25;

// connect the client to Cumulocity
const client = mqtt.connect(serverUrl, {
  username: tenant + "/" + username,
  password: password,
  clientId: clientId
});

// once connected...
client.on("connect", function () {
  // ...register a new device with restart operation
  client.publish("s/us", "100," + device_name + ",c8y_MQTTDevice", function () {
    client.publish("s/us", "114,c8y_Restart", function () {
      console.log("Device registered with restart operation support");
    });
  });

  // listen for operations
  client.subscribe("s/ds");

  // send a temperature measurement every 3 seconds
  setInterval(function () {
    console.log("Sending temperature measurement: " + temperature + "°");
    client.publish("s/us", "211," + temperature);
    temperature += 0.5 - Math.random();
  }, 3000);
});

console.log("\nUpdating hardware information...");
client.publish("s/us", "110,S123456789,MQTT test model,Rev0.1");
});

// display all incoming messages
client.on("message", function (topic, message) {
  console.log("Received operation " + message + "");
  if (message.toString().indexOf("510") !== 0) {
    console.log("Simulating device restart...");
    client.publish("s/us", "501,c8y_Restart");
    console.log("...restarting...");
    setTimeout(function () {
      client.publish("s/us", "503,c8y_Restart");
      console.log("...done...");
    }, 1000);
  }
});
});
```

Replace `serverUrl`, `clientId` and `device_name` as needed. Do not forget to specify the user credentials setting values for `tenant_ID`, `username` and `password`.

The Cumulocity MQTT protocol supports both unsecured TCP and secured SSL connections. No matter which connection type you select, your `serverUrl` should stay the same (like `mqtt.cumulocity.com`).

What does the code do?

- Configure the MQTT connection.

- When the connection is established, register a new device with a name (`device_name`) and a type (`c8y_MQTTDevice`).
- Add restart capabilities to the device.
- Subscribe to listen for operations.
- Send a random temperature measurement every 3 seconds.
- Update the device hardware information by putting a `"S123456789"` serial, a `"MQTT test model"` model and a `"Rev0.1"` revision.
- Listen to all incoming messages. In case of a `c8y_Restart` operation, simulate a device restart.

Note that the subscription is established after the device creation, otherwise if there is no device for a given `clientId`, the server will not accept it.

Run the application

Before running the application, the MQTT middleware must be installed. To achieve this, execute the following command:

```
$ npm install
```

Installation needs to be done only once. Afterwards, you only must execute the following command:

```
$ npm start
```

You should see a new registered device in the Device Management application, listed in **All devices**. In the **Measurements** tab, you will see the temperature measurements being sent by your client.

Additionally, if there will be a new operation created for this device (for example `c8y_Restart`), related information about it will be printed to the console.

Improving the agent

Now that you have done your first step, check out the section [Hello MQTT](#) to learn more about Cumulocity MQTT and improve your application.

HELLO MQTT PYTHON

In this tutorial, you will learn how to use the Python MQTT client with Cumulocity using pre-defined messages (called “static templates”).

Prerequisites

In order to follow this tutorial, check the following prerequisites:

- You have a valid tenant, a user, and a password in order to access Cumulocity.
- Verify that you have Python 3 installed:

```
$ python3 --version
Python 3.8.5
```

Python can be downloaded from www.python.org.

- Install the Python Paho client using your system's package manager or using pip:

```
$ pip install paho-mqtt
```

INFO

The above command installs Paho on your system. You may want to use [virtualenv](#) to install it only for this example.

INFO

On macOS you may need to execute `sudo easy_install pip` in case the `pip` command is not found.

Developing the “Hello, MQTT world!” client

To develop a very simple “Hello, world!” MQTT client for Cumulocity, you must

- create a Python script,
- run the script.

Create a Python script

Create a script file (for example `hello_mqtt.py`) with the following content:

```
# /usr/bin/env python3
# -*- coding: utf-8 -*-

import paho.mqtt.client as mqtt
import time, random, threading
import multiprocessing as mp

# client, user and device details
serverUrl = "mqtt.cumulocity.com"
clientId = "my_mqtt_python_client"
device_name = "My Python MQTT device"
tenant = "<<tenant_ID>>"
username = "<<username>>"
password = "<<password>>"

# task queue to overcome issue with paho when using multiple threads:
# https://github.com/eclipse/paho.mqtt.python/issues/354
task_queue = mp.Queue()

# display all incoming messages
def on_message(client, userdata, message):
    payload = message.payload.decode("utf-8")
    print(" < received message " + payload)
    if payload.startswith("510"):
        task_queue.put(perform_restart)

# simulate restart
def perform_restart():
    print("Simulating device restart...")
    publish("s/us", "501,c8y_Restart", wait_for_ack = True);

    print("...restarting...")
    time.sleep(1)

    publish("s/us", "503,c8y_Restart", wait_for_ack = True);
    print("...restart completed")

# send temperature measurement
def send_measurement():
    print("Sending temperature measurement...")
    temperature = random.randint(10, 20)
    publish("s/us", "211,{t}".format(temperature))

# publish a message
def publish(topic, message, wait_for_ack = False):
    QoS = 2 if wait_for_ack else 0
    message_info = client.publish(topic, message, QoS)
    if wait_for_ack:
```

```

    # wait_for_ack.
    print("> awaiting ACK for {}".format(message_info.mid))
    message_info.wait_for_publish()
    print("< received ACK for {}".format(message_info.mid))

# display all outgoing messages
def on_publish(client, userdata, mid):
    print("> published message: {}".format(mid))

# main device loop
def device_loop():
    while True:
        task_queue.put(send_measurement)
        time.sleep(7)

# connect the client to Cumulocity and register a device
client = mqtt.Client(clientId)
client.username_pw_set(tenant + "/" + username, password)
client.on_message = on_message
client.on_publish = on_publish

client.connect(serverUrl)
client.loop_start()
publish("s/us", "100," + device_name + ",c8y_MQTTDevice", wait_for_ack = True)
publish("s/us", "110,S123456789,MQTT test model,Rev0.1")
publish("s/us", "114,c8y_Restart")
print("Device registered successfully!")

client.subscribe("/ds")

device_loop_thread = threading.Thread(target = device_loop)
device_loop_thread.daemon = True
device_loop_thread.start()

# process all tasks on queue
try:
    while True:
        task = task_queue.get()
        task()
except (KeyboardInterrupt, SystemExit):
    print("Received keyboard interrupt, quitting ...")
    exit(0)

```

Replace `serverUrl`, `clientId` and `device_name` as needed. Do not forget to specify the user credentials setting values for `tenant_ID`, `username` and `password`.

Cumulocity MQTT protocol supports both unsecured TCP and secured SSL connections, so when configuring a port remember to use the correct one. No matter which connection type you select, your `serverUrl` should stay the same (like `mqtt.cumulocity.com`).

The above example uses a TCP connection. If you would like to use an SSL connection, remember to use the proper configuration from the Paho MQTT client. Further information can be found at www.eclipse.org.

What does the script do?

- Configure a MQTT connection.
- Register an `on_message` callback function which will print incoming messages. In case of a `c8y_Restart` operation, it will simulate a device restart.
- Register an `on_publish` callback function which will be called after a publish message has been delivered.
- Connect with Cumulocity via the MQTT protocol.
- Create a new device with a name (`device_name`) and a type (`c8y_MQTTDevice`).
- Update the device hardware information by putting a `"S123456789"` serial, a `"MQTT test model"` model and a `"Rev0.1"` revision.
- Subscribe to the static operation templates for the device – this will result in an `on_message` method call every time a new operation is created.
- Start the `device_loop_thread` which sends a temperature measurement every 7 seconds.
- Prepare a `task_queue`, which runs all tasks one by one.

What does the `publish` message do?

- Publish a given message about the given topic via MQTT.
- When publishing the message it uses QoS 2. So to be sure that the message was delivered, it will wait for server ACK (until the `on_publish` method is called with the matching message ID).

Note that the subscription is established after the device creation, otherwise if there is no device for a given `clientId`, the server will not accept it.

Run the script

To run the script just use the command:

```
$ python3 hello_mqtt.py
```

After starting the application you should see a new registered device in the Device Management application, listed in **All devices**. In the **Measurements** tab, you will see the temperature measurements being sent by your client.

Additionally, if there will be a new operation created for this device (for example `c8y_Restart`), information about it will be printed to the console.

Improving the agent

Now that you have done your first step, check out the section [Hello MQTT](#) to learn more about Cumulocity MQTT and improve your application.

MQTT SERVICE

FEATURE PREVIEW

This feature is in **Public Preview**. That is, it is not yet generally available and may be subject to change in the future.

REQUIREMENTS

To work with the MQTT Service, the following requirements must be met:

- The Cumulocity Messaging Service must be deployed in your Cumulocity environment.
- The Cumulocity MQTT Service must be deployed in your Cumulocity environment.
- Your tenant must be subscribed to the mqtt-service microservice. This may have been done automatically, depending on how your Cumulocity environment was configured. To check the subscription, open the Administration application and navigate to **Ecosystem > Microservices**. If you do not see the mqtt-service microservice listed, contact [product support](#) (for public environments) or your Cumulocity administrator (for dedicated environments) to request the subscription for your tenant.

The MQTT Service is a new MQTT endpoint implementation for Cumulocity that provides the following benefits:

- Sending and receiving arbitrary payloads on any MQTT topic. Note that the topics used by the Cumulocity [Core MQTT](#) implementation currently cannot be used with the MQTT Service.
- User-provided microservices can send and receive messages on MQTT topics, and map messages to and from the Cumulocity data model. The typical use case for such a microservice is to map between MQTT device payloads, and the Cumulocity REST and Notifications 2.0 APIs.
- Multi-tenancy support. A single endpoint serves multiple tenants and tenants are completely isolated from each other.
- Bi-directional TLS support. All MQTT traffic is encrypted and clients can authenticate using X.509 certificates.

The MQTT Service does not replace the existing [Core MQTT](#) capability of Cumulocity that supports sending device data already in the Cumulocity domain model directly into the platform. The new capability provided by the MQTT Service allows for easier integration of MQTT devices that cannot use the Cumulocity domain model. It also supports more flexible communication patterns between devices, applications, and the Cumulocity platform, controlled by user-provided microservices.

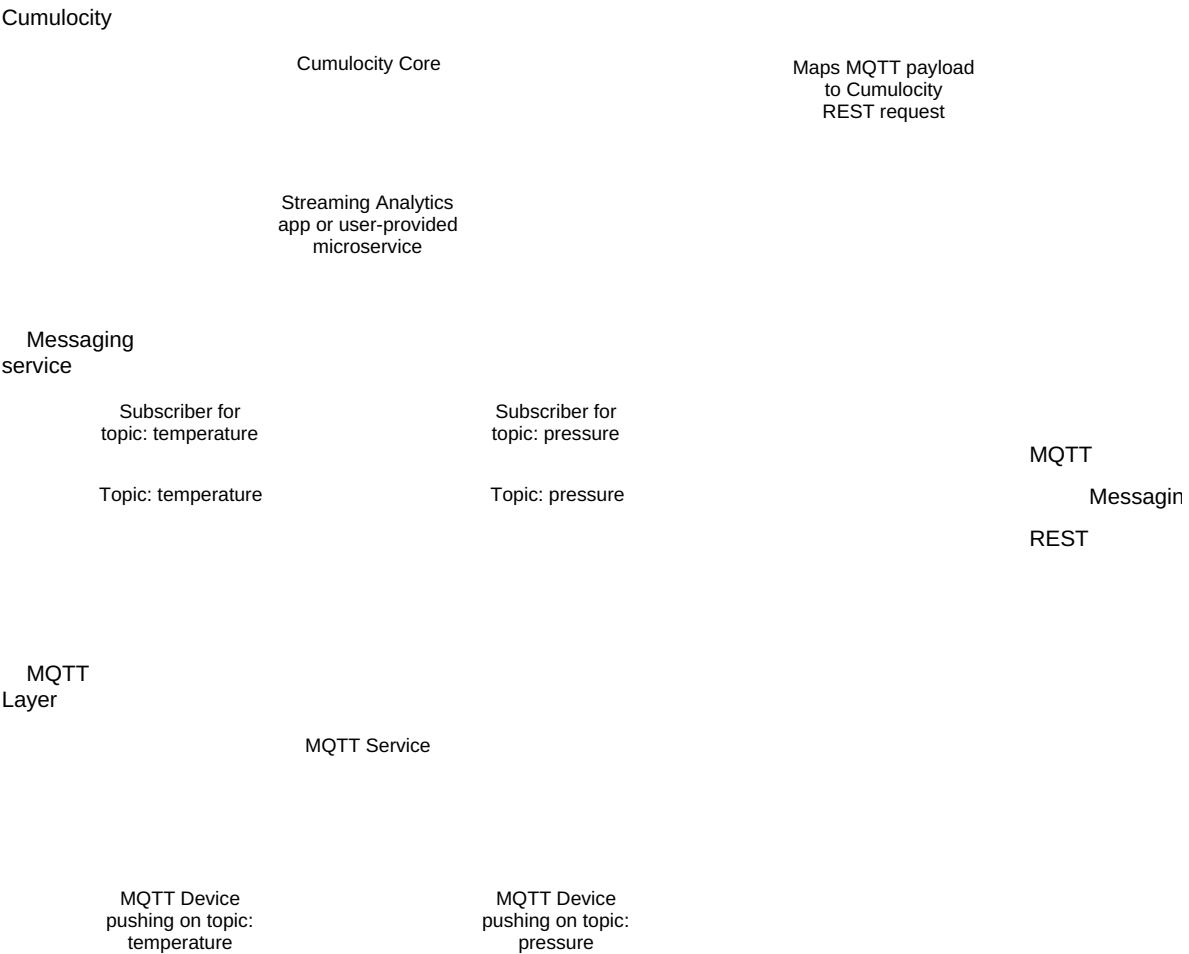
Device isolation is the default behaviour for the MQTT Service. Each MQTT client has its own private topic space and cannot directly receive messages from other clients, enhancing security and isolation between devices.

This documentation does not describe the basics of MQTT communication. If you are unfamiliar with MQTT, we recommend you to consult one of the numerous introductions on the internet. Some references can be found on the [MQTT website](#).

OVERVIEW

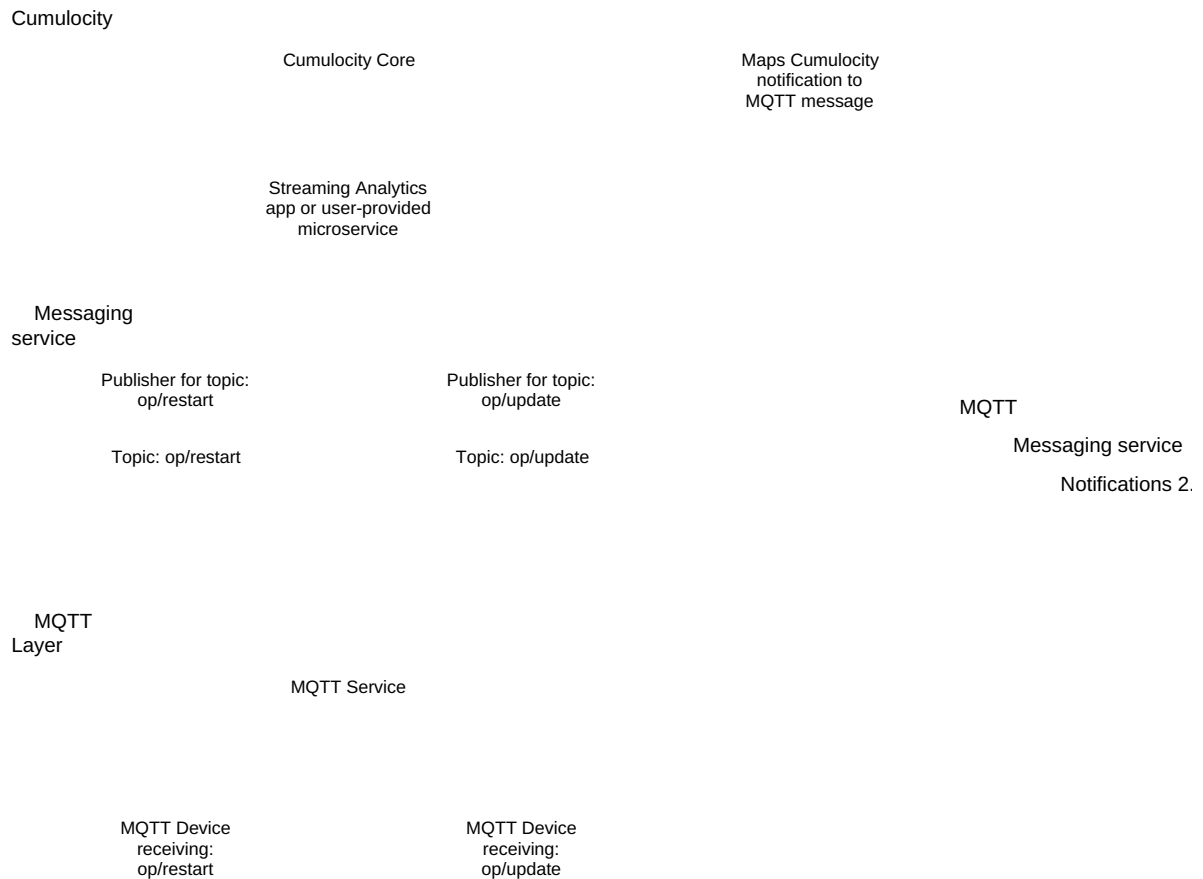
ARCHITECTURE

The MQTT Service works together with the Messaging Service to provide a framework for highly customizable and flexible MQTT message processing solutions. The diagram below illustrates how a message flows, starting from the device, through the Messaging Service, then to a user-provided microservice where it is converted to the Cumulocity JSON format and delivered to Cumulocity using the standard REST API.



All MQTT messages published to the MQTT Service are forwarded to the Messaging Service, where they are persisted, waiting to be consumed. A custom microservice or [Streaming Analytics app](#) that understands the topic and payload structure can consume the MQTT messages, and then translate and push them into Cumulocity.

Similarly, a custom microservice or Streaming Analytics app can send messages to devices, as shown in the diagram below. In this case, the user-provided microservice receives messages from Cumulocity through a Notifications 2.0 subscription. These messages are mapped to the payload structure used by the MQTT devices, then published to MQTT topics.



As with MQTT messages published by devices, messages published from a microservice will be forwarded to the Messaging Service, where they can be consumed by MQTT devices subscribed to the relevant topics.

Custom microservices may use the [Java client](#) to publish to or consume from MQTT topics. They can use the [Microservice SDK](#) to push data into Cumulocity.

MQTT SERVICE COMPARED TO CORE MQTT

The table below presents a basic comparison between the Cumulocity Core MQTT functionality and that of the MQTT Service.

	Core MQTT	MQTT Service
QoS	0, 1, 2	0, 1
Clean session	Starting with clean session is recommended	Starting with clean session is required
Retained flag	Not supported	Not supported
Last will	Supported	Supported
MQTT 5.0 features	Not supported	MQTT 5.0 clients can connect. Partial support for MQTT 5.0 features
Authentication	Basic and TLS device certificates	Basic and TLS device certificates
Scalability	Horizontal	Currently a single instance. Horizontal scaling will be available in the GA release

	Core MQTT	MQTT Service
Topic format	Determined by the SmartREST 2.0 protocol	Unrestricted. SmartREST topic names are reserved and cannot currently be used
Payload	Determined by the SmartREST 2.0 protocol	Unrestricted. The maximum message size is 128 KiB including all headers
Extensibility	Limited by SmartREST 2.0 custom templates	Streaming Analytics apps or custom mapping microservices can support arbitrary MQTT-based protocols
Message processors/consumers	Built-in message processor for each SmartREST 2.0 topic	Streaming Analytics apps or custom mapping microservices can support multiple processors for a topic
JSON via MQTT	Limited feature set	Streaming Analytics apps or custom mapping microservices can support arbitrary JSON payloads

MQTT PROTOCOL IMPLEMENTATION

This section covers some implementation details of the MQTT Service. The MQTT Service implementation supports clients connecting using MQTT versions 3.1, 3.1.1 and 5.0, although not all MQTT 5.0 protocol features are currently supported.

CONNECTING TO THE SERVICE

! IMPORTANT

MQTT Service requires clients to connect with clean session flag enabled, set to “1” (true), otherwise the client connection is rejected by the server.

MQTT connections to the MQTT Service must use TCP. Use your tenant domain as the target host for the connection, for example `{my-tenant}.cumulocity.com`.

Available ports:

	TCP
TLS	9883
no TLS	2883

Port 9883 (TLS) is the default port and should be used for secure, encrypted communication. Both one-way (server certificate only) and two-way (both client and server certificates) TLS are supported. When client certificates are not used, the server authenticates the client using standard username and password credentials. Port 2883 (no TLS) is not enabled in Cumulocity shared public environments due to the security risks of unencrypted traffic. To enable port 2883 in a dedicated environment, please contact [Product support](#).

TOPICS

MQTT Service topics are mapped to the Messaging Service subscriptions with identical names, including additional URL encoding. The Messaging Service subscriptions reliably store the topic messages for asynchronous processing. The messages stored on these subscriptions can be consumed using a dedicated [Java Client](#).

Topic restrictions

The MQTT Service does not impose any topic structure. There are just a few topic names which are reserved for historic purposes and future use, namely:

- All [SmartREST 2.0](#) related topics
- `error`
- `devicecontrol/notifications`

INFO

Wildcard topics (`+` , `#`) and system topics starting with `$` are not currently supported.

Other than these restrictions you are free to use any topic name which is compatible with the [MQTT specification](#).

Topic limits

The MQTT Service imposes several topic-related limits. See the [Service Quotas](#) section for details of the current limits in force.

There is a limit on the total number of topics that a single tenant can create. When the creation of a new topic, either by creating it via the client publishing a message or subscribing to a non-existent topic, would breach the topic limit the delivery of the packet is prevented.

The different MQTT protocol versions provide different feedback when this limit is exceeded.

MQTT 5 clients:

- Have access to the reason code and reason string describing the failure when using QoS 1 with acknowledgements, where the reason code is `QUOTA_EXCEEDED: 0x97`.

MQTT 3.1 and 3.1.1 clients:

- Clients only have access to the reason code describing the failure when using QoS 1 with acknowledgements and only for SUBSCRIBE packets, where the reason code is `0x80`.
- For PUBLISH packets, the client will be disconnected with no further information as per the MQTT specification.

In addition to the topic count, the MQTT Service also limits the size of the message backlog on each topic. The message backlog contains all messages that have been published on the topic but not yet received and acknowledged by all subscribers to the topic. When the backlog limit is reached, further attempts to publish to the topic will fail until some messages have been consumed.

Each message in a topic backlog also has a time-to-live (TTL) that starts at the moment the message is published. When the TTL of a message expires, that messages will be deleted from the backlog regardless of whether all subscribers have received it or not. MQTT clients do not receive any notification that messages have been discarded from a backlog due to TTL expiry.

Error topic

The MQTT Service provides clients the ability to review errors through messages received by subscribing to the error topic, `$debug/error`. When subscribing to the topic it will act as a per-client topic, meaning the client will only receive messages exclusively related to their client ID. For example, if a client was attempting to subscribe to a new topic, and the creation of the topic would exceed the topic limit, only that client would receive an error.

According to the MQTT 3.1.1 specification, if either the server or the client encounters a protocol violation, it must close the network connection on which it received the control packet which caused the violation.

In such instances MQTT clients must reconnect to be able to receive error messages from the error topic via the subscription. Error messages received after this reconnection are from the previous session. This can lead to confusion when attempting corrective actions. Therefore, we highly recommend you to build a microservice which uses the MQTT Service SDK to consume error messages, or use MQTT 5 for clients and make use of the reason codes feature.

Topic cleanup

The MQTT service will automatically remove topics which are no longer active. Topics are recognized as inactive when there are no subscriptions and the internal publisher to the topic is closed. The publisher is responsible for publishing the modified MQTT service

messages to the correct topic. The publishers live within a cache, where the publisher expires after one hour. Due to this it can take up to an hour after removing all subscriptions from a topic for it to be automatically deleted.

PAYLOAD

MQTT protocol messages map bidirectionally to the internal MQTT Service message format which includes the original payload and additional metadata fields. Assuming Java types, the packed message structure looks as follows:

MqttServiceMessage

Field name	Type	Description
payload	byte[]	MQTT payload
metadata	MqttServiceMetadata	Metadata from the MQTT message

MqttServiceMetadata

Field name	Type	Description
clientId	String	Unique MQTT client identifier, usually used as an external identifier
messageId	int	Unique MQTT message ID per client, available only with QoS 1 and 2
dupFlag	boolean	Indicates this message is a resend by the MQTT client
userProperties	Map	Reserved for future use of MQTT 5.0 features
payloadFormatIndicator	enum	Reserved for future use of MQTT 5.0 features
contentType	String	Reserved for future use of MQTT 5.0 features
correlationData	byte[]	Reserved for future use of MQTT 5.0 features
responseTopic	String	Reserved for future use of MQTT 5.0 features
topic	String	The name of the MQTT topic that the message was published by the client

The [Java Client](#) contains classes representing the above model.

Payload restrictions

The MQTT Service does not impose any specific payload format. All the incoming MQTT messages must meet the specification in terms of fixed and variable headers, but the payload for published messages is unrestricted. A Streaming Analytics app or a custom microservice will receive the exact same set of bytes that was sent by an MQTT device, and is responsible for converting these to a Cumulocity compatible format.

The size of the MQTT payload is limited to a maximum value that includes both the message header and body. The size of an MQTT packet header varies, but it will be at least 2 bytes. See the [Service Quotas](#) section for details of the current limit in force.

FEATURES

Authentication and authorization

The MQTT Service supports the following authentication methods:

- **Username and password** The MQTT username must include the tenant ID and username in the format `<tenantID>/<username>`.
- **Device certificates** For secure communication, devices must contain the entire chain of certificates leading to the trusted root

certificate, or if only the device certificate is provided, then the immediate issuer certificate must be uploaded to the platform's truststore. You can do this via [the Trusted certificates page in the UI](#) or via [REST](#). Moreover, the devices must contain the server certificate in their truststore.

If the trust anchor (that is, the trusted root or intermediate certificate) used to validate the device certificate is trusted by multiple tenants, the device must also specify the tenant ID in the **MQTT username** field. This ensures that the platform can correctly identify which tenant the device is attempting to connect to. While multi-tenant trust anchors are not currently supported in Cumulocity, this feature may be introduced in the future. If the tenant ID is provided, it must correspond to a tenant that trusts the given certificate; otherwise, the connection will be rejected.

ClientID

The MQTT **ClientID** field identifies the connected client. **ClientID** may consist of up to 128 alphanumeric characters. Each client connecting to the MQTT Service must have a unique client identifier, connecting a second client with the same identifier will result in the previous client's disconnection.

Quality of Service (QoS)

The MQTT Service implementation supports two levels of MQTT QoS:

- QoS 0: At most once:
 - The client sends the message once (fire and forget).
 - There is no response from the server.
 - There is no guarantee that subscribers will receive the message.
- QoS 1: At least once:
 - The client awaits server acknowledgment for each published message.
 - The client should re-send the message if there was no acknowledgement from the server.
 - It is guaranteed that subscribers will receive a message that was acknowledged by the server.
 - Subscribers may receive more than one copy of a message.
- QoS 2: Exactly once:
 - not supported

For subscriptions, the MQTT Service will deliver messages in the QoS that the client defined when subscribing to the topic (QoS 0 or 1).

Clean session

The MQTT Service **requires** the clean session flag to be set to "1" (true). Disabling clean session will result in client connections being rejected by the server.

Retained flag

The retained flag is currently ignored. Publishing data with the retained flag on the topic is allowed but has no practical difference to sending it without the flag.

Last will

In MQTT, the "last will" is a message that is specified at connection time and that is executed when the client loses the connection. Last will is fully supported by the MQTT Service, and as with any other publish messages you can use any unreserved topic and any payload.

RETURN CODES

The MQTT Service follows the MQTT specification for server responses. For example, if invalid credentials are sent in the **CONNECT** message, the server response **CONNACK** message contains the **0x05** return code. The return code can be treated similarly to REST API HTTP codes, such as 401.

MQTT 5.0 FEATURES

Clients can connect using version 5.0 of the MQTT protocol. Support for additional MQTT 5.0 features will be added in future releases.

MQTT TLS CERTIFICATES

Server certificates

The MQTT Service uses the same server certificates that are assigned to the main Cumulocity environment domain. It always sends these certificates during TLS handshake to devices. Moreover, Enterprise tenants are not able to customize those certificates via the SSL Management feature.

Device (client) certificates

Using device certificates with the MQTT Service shares the same requirements as outlined in [Device certificates](#).

If the trust anchor (that is, the trusted root or intermediate certificate) used to validate the device certificate is trusted by multiple tenants, the device must also specify the tenant ID in the **MQTT username** field. This ensures that the platform can correctly identify which tenant the device is attempting to connect to. For more information, see [Authentication and authorization](#).

Adding and trusting CA certificate

TLS trust anchors in the Cumulocity platform are defined per tenant. To use device certificates for authentication, the CA or intermediate certificate that signs the device certificates must be uploaded to the platform and added to the tenant's list of trusted certificates. You can do this via [the Trusted certificates page in the UI](#) or via [REST](#).

Additionally, ensure that the **Auto registration** option is enabled when adding certificates. This allows any device presenting a valid certificate to be automatically registered on the platform when it first connects.

Creating self-signed certificates

In order to self-sign the device certificates, the root CA certificate needs to be created. Using the OpenSSL CLI tool, create a private key and then generate a self-signed root certificate from it.

```
openssl genpkey -algorithm RSA -out ca.key
openssl req -x509 -new -nodes -key ca.key -sha256 -days 3650 -out ca.crt -subj "/C=UK/O=YourCompany/OU=YourOrg/CN=MQTTServiceCA"
```

Then create a private key for the device, generate the certificate signing request from this private key, and then sign the CSR.

```
openssl genpkey -algorithm RSA -out client.key
openssl rsa -in client.key -out client-key.pem -outform PEM
openssl req -new -key client.key -out client.csr -subj "/C=UK/O=YourCompany/OU=YourOrg/CN=mqtt-client"
openssl x509 -req -in client.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out client.crt -days 3650 -sha256
cat client.crt ca.crt > client-chain.pem
```

If you have more advanced requirements regarding certificate creation, see [Generating and signing certificates](#).

Using certificates

Once the CA certificate has been uploaded and trusted in Cumulocity, devices can authenticate using client certificates signed by your trusted CA. To connect using any MQTT client, use the previously generated client certificate and key. This example uses the Mosquitto MQTT client:

```
mosquitto_pub --cafile cumulocity.com.pem -d -q 1 \
-h "cumulocity.com" -p "9883" -i myclient \
-u t11101 \
-t "v1/devices/me/telemetry" \
--key client-key.pem \
--cert client-chain.pem \
-m '{"temperature":25}'
```

Explanation:

- `--cafile cumulocity.com.pem`: This file contains the CA certificate of Cumulocity's MQTT Service broker, used to validate the server's identity.
- `--key client-key.pem` and `--cert client-chain.pem`: These are your client certificate and private key, signed by your trusted CA.
- `-u t11101`: (Optional) Specifies the MQTT username, which must be your tenant ID as described in [Authentication and](#)

[authorization](#). In this example, `t11101` is the tenant ID.

Downloading the CA certificate (`cumulocity.com.pem`):

To download the Cumulocity MQTT Service broker's CA certificate:

1. Open `cumulocity.com` in a browser.
2. Click the padlock icon in the address bar and view the certificate details.
3. Download or export the root certificate, and save it as `cumulocity.com.pem`.

Alternatively, you can use `openssl` to retrieve and extract the certificate:

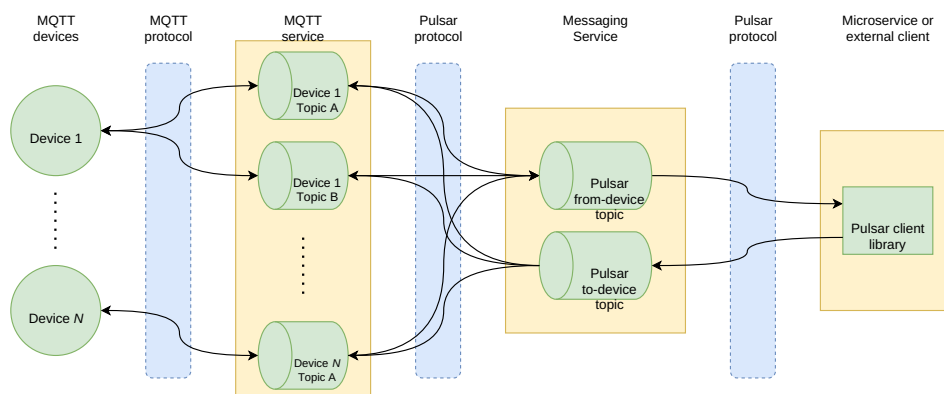
```
echo | openssl s_client -connect cumulocity.com:9883 -showcerts 2>/dev/null | \
sed -ne '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' > cumulocity.com.pem
```

INFO

Cumulocity uses certificates signed by well-known public CAs. Some clients (like Mosquitto) require explicitly providing the CA file, while others (like MQTTX) trust these certificates automatically.

CONNECTING MICROSERVICES AND APPLICATIONS

Cumulocity microservices and external applications can consume messages published by devices connected to the MQTT Service, and publish messages back to those devices. To do this, your microservice or external application connects to the Cumulocity Messaging Service, a deployment of [Apache Pulsar](#), and uses the Pulsar protocol to publish and consume MQTT messages. The diagram below shows the important interfaces and data flows used when interacting with the MQTT Service through Pulsar.



INFO

An *MQTT Service messaging client* is a software component that interacts with the MQTT Service through Pulsar. It can be deployed as a microservice hosted by the Cumulocity platform, or as part of an external application hosted outside the platform. This documentation refers to such a component simply as a *client*. If the implementation or behaviour differs depending on where the client is hosted, those differences are documented where relevant.

The MQTT Service implements *device isolation*, meaning that MQTT devices connected to the MQTT Service **cannot** communicate directly with each other using the MQTT protocol. All inter-device communication must be managed explicitly by the client, as shown in the diagram.

This documentation does not cover the publish-subscribe messaging concepts and architecture implemented by Pulsar, nor any features of the Pulsar client libraries beyond those needed to implement a simple MQTT Service client. To learn more about those subjects, refer to the [Pulsar product documentation](#).

CONNECTING TO THE MESSAGING SERVICE

To connect your client to the Messaging Service, you will need:

1. A [Pulsar client library](#).
2. The URL of the Messaging Service (Pulsar broker) in your Cumulocity environment.
3. Credentials for a user in your tenant with permission to access MQTT Service data on the Messaging Service.

Each of these prerequisites is explained in detail below.

Pulsar client library

Open-source Pulsar client libraries are available for a number of different languages and protocols. The example code in this documentation will use the [Java client library](#). Pulsar has strong cross-version compatibility. Use the latest version of your chosen client library regardless of the server version used by the Messaging Service. Integration with the MQTT Service does not require advanced Pulsar features that may only be available in the latest server version.

⚠ CAUTION

Currently only “basic” (username/password) authentication is supported for clients connecting to the Messaging Service through Pulsar. Therefore, you must ensure that your chosen Pulsar client library supports this authentication scheme.

Pulsar URL

For a microservice client, the URL should be obtained from the `C8Y_BASEURL_PULSAR` environment variable that will be passed to the microservice when it starts running. For an external application client, the URL has the general form `pulsar+ssl://<tenant_domain>:6651/`, where `<tenant_domain>` is the domain of your Cumulocity tenant, for example `my-tenant.cumulocity.com`. As implied by the `pulsar+ssl` protocol name, all external application client connections will use SSL/TLS security. Currently, only one-way TLS is supported. The server provides a certificate that the client can verify. Client certificates cannot be used. Implementing an external application client so that it reads the Pulsar URL from the `C8Y_BASEURL_PULSAR` environment variable makes it easier to develop a client that can be deployed as either a microservice or an external application.

Pulsar authentication

Authentication credentials identify both the Cumulocity tenant and the user within that tenant. Currently, only “basic” (username and password) authentication is supported for clients connecting to the Messaging Service through Pulsar. For a microservice client, you should use the credentials of the per-tenant [service user](#) that will be passed to the microservice when the tenant is subscribed to it. For an external application user, you can use the credentials of any tenant user with the appropriate authorization roles assigned, as described below. The username must be in the form `<tenantID>/<user>` where `<tenantID>` is the tenant ID (not the tenant name), and `<user>` is a user within that tenant. If two-factor authentication (TFA) is enabled for your tenant, your user must have the `devices` role assigned to disable the TFA check for that user. See [TFA Settings](#) for more information. Note that the `devices` role may be shown as “Device User” in the Cumulocity user interface.

Role-based access control

Pulsar client connections will be granted access to Messaging Service resources based on the roles and permissions assigned to the authenticated user. The following roles and permissions should be used for MQTT Service messaging clients:

Role and permission	Access granted
Mqtt service messaging topics, Read	Consume messages from MQTT devices connected to the MQTT Service
Mqtt service messaging topics, Update	Publish messages to MQTT devices connected to the MQTT Service

For microservice clients, the required permissions should be added to the `requiredRoles` section of the [microservice manifest](#), which will grant the requested permissions to the per-tenant service user. For example:

```
{
  "apiVersion": "v2",
  "name": "my-mqtt-service-client",
  "version": "1.0.0",
  ...
  "requiredRoles": [
    "ROLE_MQTT_SERVICE_MESSAGING_TOPICS_READ",
    "ROLE_MQTT_SERVICE_MESSAGING_TOPICS_UPDATE"
  ],
  ...
}
```

For external application clients, the required permissions should be configured for the authenticating user through the [Administration application](#).

Assign only the minimum permissions needed for the client to operate. For example, if your microservice only consumes messages, do not include the `ROLE_MQTT_SERVICE_MESSAGING_TOPICS_UPDATE` permission in the manifest.

Example code – connecting to the Messaging Service

The code snippet below shows how to use the Pulsar Java client library to connect to the Messaging Service with basic authentication. It assumes that the Pulsar URL is in the `C8Y_BASEURL_PULSAR` environment variable and that the tenant identifier, username and password are provided on the command line. Note that the client library will not actually attempt to connect to the Pulsar server immediately when the `PulsarClient` object is created. In the interest of brevity and clarity, this example does no error handling. A realistic implementation would need to handle exceptions thrown by the Pulsar client library methods.

```

package c8y.example.mqttservice;

import java.text.MessageFormat;
import java.nio.charset.StandardCharsets;

import org.apache.pulsar.client.api.Consumer;
import org.apache.pulsar.client.api.Message;
import org.apache.pulsar.client.api.MessageListener;
import org.apache.pulsar.client.api.Producer;
import org.apache.pulsar.client.api.PulsarClient;
import org.apache.pulsar.client.api.PulsarClientException;
import org.apache.pulsar.client.api.Schema;
import org.apache.pulsar.client.impl.auth.AuthenticationBasic;

public class SimplePulsarClient {
    public static void main(String[] args) throws Exception {
        // Validate command line.
        if (args.length != 2) {
            System.err.println("Usage: SimplePulsarClient <tenantID> <username>");
            System.err.println("The Pulsar URL will be read from the C8Y_BASEURL_PULSAR environment variable");
            System.err.println("The password will be read from the console");
            System.exit(-1);
        }

        // Collect all the configuration properties.
        final String pulsarUrl = System.getenv("C8Y_BASEURL_PULSAR");
        final String tenantID = args[0];
        final String username = args[1];
        final String password = new String(System.console().readPassword("Password for user %s/%s: ", tenantID, username));

        // Create the basic authentication credentials object.
        final AuthenticationBasic basicAuth = new AuthenticationBasic();
        basicAuth.configure(MessageFormat.format("{\"userId\":\"{0}/{1}\",\"password\":\"{2}\"", tenantID, username, password));

        // Create a Pulsar client using the basic authentication credentials.
        // The client will *not* try to connect and authenticate immediately.
        final PulsarClient client = PulsarClient.builder()
            .serviceUrl(pulsarUrl)
            .authentication(basicAuth)
            .build();
        System.out.println("Created Pulsar client");

        // The rest of the example will go here...
    }
}

```

MESSAGE PAYLOADS AND PROPERTIES

Pulsar messages consist of a *payload* and set of *properties*.

The payload is a sequence of zero or more bytes, identical to the payload of the MQTT **PUBLISH** message that the Pulsar message corresponds to. It is the client's responsibility to understand the format of the payloads produced and accepted by the MQTT devices it communicates with.

Pulsar message properties are name-value pairs, where both the name and the value are text strings. The properties recognised by the MQTT Service are listed in the table below. Messages received from MQTT devices will **always** include the properties marked as required, and may include any of the optional properties. Received messages will not include any properties other than those listed here. Messages published to MQTT devices **must** include all of the required properties, and may include any of the optional properties. If a published message includes any properties other than those listed here, those properties will be ignored by the MQTT Service.

Property name	Required	Value type and encoding	Purpose
topic	YES	String	MQTT topic name

Property name	Required	Value type and encoding	Purpose
<code>clientID</code>	YES ⁽¹⁾	String	MQTT client identifier
<code>tx.payloadFormatIndicator</code> ⁽²⁾	NO	Single byte with two permitted values, encoded as strings "0" and "1"	MQTT v5 Payload Format Indicator
<code>tx.contentType</code>	NO	String	MQTT v5 Content Type
<code>tx.responseTopic</code>	NO	String	MQTT v5 Response Topic
<code>tx.correlationData</code>	NO	Sequence of bytes, encoded as a Base64 string	MQTT v5 Correlation Data
<code>tx.userProperties.<name></code>	NO	String	MQTT v5 User Property with name <code>name</code> ⁽³⁾

Notes:

1. The `clientID` property can be omitted from a published message only in special case of a *broadcast* message, described below in [broadcast messages](#).
2. The `tx.` prefix indicates that a property is specific to a *transport*, in this case the MQTT Service. Other transports will define their own transport-specific properties, but all transports will use `topic` and `clientID`.
3. The MQTT version 5 specification allows a message to include more than one user property with the same name. This feature is **not** supported by the MQTT Service. If a device publishes a message containing multiple user properties with the same name, only one of these will be copied into the Pulsar message. It is undefined which property will be copied.

CONSUMING MESSAGES FROM MQTT DEVICES

All messages published by devices connected to the MQTT Service for a given tenant will be published to a *single* Pulsar topic, identified by the URL `persistent://<tenantID>/mqtt/from-device`. The topic URL can be broken down into 4 components:

Component	Description
<code>persistent</code>	Indicates that this is a persistent topic that will be preserved by the Messaging Service across component failures and restarts, to provide "at least once" delivery guarantees
<code><tenantID></code>	The Pulsar tenant ID, which will match the Cumulocity tenant ID
<code>mqtt</code>	The Pulsar namespace within the tenant, which will always be <code>mqtt</code> for the MQTT Service
<code>from-device</code>	The Pulsar topic within the namespace, which will always be <code>from-device</code> for message from devices connected to the MQTT Service

Your client will only be able to consume from this topic if the authenticated user has the "read" permission on the "Mqtt service messaging topics" role. The client will not be able to consume from any other topic.

The client identifier of the device that published the messages, and the MQTT topic it was published on, can be obtained from the message properties `clientID` and `topic` as described above. This means that your client **must** consume every message published by every device connected to the MQTT Service for the tenant, even those you are not interested in. Messages that are not of interest to the client can simply be acknowledged without further processing.

⚠ CAUTION

Your client **must** be trusted to safely handle every message published by every device connected to the MQTT Service in your tenant. If untrusted users have access to your tenant, these users should **not** be permitted to upload microservices, nor to connect external application clients to the Messaging Service. This recommendation also applies in the case of multiple customers, who do not mutually trust each other, sharing

a single tenant.

Durable subscriptions and message acknowledgement

Subscribing a consumer to a topic establishes a *durable subscription* to the topic. This means that the Messaging Service will retain messages published to the topic until they have been delivered to, and acknowledged by, a client. The subscription will remain until it is explicitly deleted. It will not be removed simply because the client is not currently running. Messages that are published while the client is disconnected will be available for it to consume when it reconnects. After consuming each message, the client **must** explicitly acknowledge it. Acknowledging a message tells the Messaging Service that the client has no further interest in it, allowing the message to be discarded. See the section on [best practices](#) below for more information on managing durable subscriptions correctly.

Example code – consuming messages

The code snippet below shows how to use the Pulsar Java client library to consume messages from the MQTT Service `from-device` topic. It extends the previous example that showed how to [set up the connection to the Pulsar server](#).

To consume messages from the topic, your client should create a Pulsar `Consumer` and subscribe it to the topic. The consumer should register a `MessageListener` callback that will be called whenever a new message arrives on the topic. The `MessageListener` implementation shows how to access the payload and properties of the received messages. For simplicity, the application messages in the example are simple text strings. However, the payload of the Pulsar message will always be an array of bytes, that must be converted to the format used by the application.

```
// Create a simple message listener that will log some details of
// each message received, when registered with a consumer.
final MessageListener<byte[]> listener = new MessageListener<byte[]>() {
    @Override
    public void received(Consumer<byte[]> consumer, Message<byte[]> message) {
        final String clientId = message.getProperty("clientId");
        final String topic = message.getProperty("topic");
        System.out.println(MessageFormat.format("Received message from MQTT device {0} on MQTT topic {1}", clientId, topic));
        System.out.println(MessageFormat.format("Message payload: {0}", new String(message.getValue(), StandardCharsets.UTF_8)));
        System.out.println(MessageFormat.format("Message properties: {0}", message.getProperties()));
        try {
            // Acknowledge the message.
            consumer.acknowledge(message);
        } catch (PulsarClientException e) {
            e.printStackTrace();
        }
    }
};

// Create a Pulsar consumer on the from-device topic for the tenant,
// using the listener defined above to process each message.
// This will trigger connection and authentication by the client.
final Consumer<byte[]> consumer = client.newConsumer(Schema.BYTES)
    .topic(MessageFormat.format("persistent://{0}/mqtt/from-device", tenantID))
    .subscriptionName("demoSubscription")
    .messageListener(listener)
    .subscribe();
System.out.println("Created Pulsar consumer");
```

PUBLISHING MESSAGES TO MQTT DEVICES

Any messages that your client wants to send to devices connected to the MQTT Service for a given tenant must be published to a *single* Pulsar topic, identified by the URL `persistent://{tenantID}/mqtt/to-device`. The components of the URL should be interpreted as described in [Consuming messages from MQTT devices](#) above.

Your client will only be able to publish to this topic if the authenticated user has the “update” permission on the “Mqtt service messaging topics” role. The client will not be able to publish to any other topic.

Messages published to the `to-device` topic are routed to connected MQTT devices using the two required message properties:

Property name	Purpose
<code>clientID</code>	Client identifier of the MQTT device that should receive the message
<code>topic</code>	Name of the MQTT topic that the message should be published to

If the `topic` property is empty or missing, the message will not be published to any MQTT client. The message will only be published to a device with an active subscription to the named MQTT topic. The message will only be published to a client that is connected at the time the MQTT Service processes the published message.

Successfully publishing a message to the Messaging Service does **not** mean that the message has been successfully delivered to any MQTT device. Onward publishing to MQTT devices happens *asynchronously* and without any feedback to the Pulsar client. Messages will be delivered to devices according to the MQTT protocol specification, using the QoS level of the MQTT subscription made by the device. However, because MQTT devices are required to use a *clean session* when connecting to the MQTT Service, messages published to a device while it is disconnected will not be delivered.

Broadcast messages

To enforce device-level isolation, messages are published **only** to the specific MQTT client identified by the `clientID` property, provided that client has an active subscription to the relevant MQTT topic. If the `clientID` property is not present, the message is broadcast to **all** connected MQTT clients with active subscriptions to that topic.

Broadcast publishing is potentially expensive when many clients are connected and may deliver messages to unexpected devices. Use broadcast only when the application must publish the same message to every device subscribed to a topic.

Message keys

To facilitate efficient delivery and correct ordering of messages sent to MQTT devices, clients **must** also set the *key* of a Pulsar message published to the `to-device` topic. The key should be set as follows:

- When the `clientID` message property is set, the key should have the same value as this property.
- When the `clientID` message property is **not** set, the key should have the same value as the `topic` message property.

Handling of invalid messages

Published messages that do not follow the rules for message properties and keys documented above will **not** be delivered to any MQTT device. In particular this applies to messages with the following invalid configuration:

- The message *key* is not set.
- The message *key* is set but does not match the `clientID` or `topic` property as described in [message keys](#).
- The `clientID` property is set but has an empty value.
- The `topic` property is not set, or it is set but has an empty value.

An alarm will be raised in the Cumulocity tenant when one of these invalid messages is detected and discarded. The rate of alarm sending is limited to avoid overloading the tenant with redundant alarms alerting about the same error on different messages.

A message with a non-empty `clientID` property referring to an MQTT device that is not currently connected is **not** considered to be invalid. However, this message will not be delivered to the device, even if it connects later, because of the requirement for devices to use a *clean session* when connecting. Similarly, a message published to a connected MQTT device that is not currently subscribed to the MQTT topic specified in the `topic` property is not considered to be invalid. In these situations, the message will not be delivered but no alarms will be raised.

Example code – publishing messages

The code snippet below shows how to use the Pulsar Java client library to publish messages to the MQTT Service `to-device` topic. It extends the previous examples that [set up the connection to the Pulsar server](#) and [created a message consumer](#).

To publish messages to the topic, your client should first create a Pulsar `Producer` associated with the topic. Then, the `Producer` can be used to create new `Message` objects that will be published to the topic. The example code shows how to correctly set the message properties and message key for messages targeted at a single device, and for “broadcast” messages. Again, the example assumes that the application messages are simple text strings, that must be converted to the byte array expected by the MQTT Service. For clarity, most error-handling code is omitted from the example. See [Handling Messaging Service errors](#) for advice on dealing with errors in a production client.

```

// Wrap all the operations that might fail after we create the
// durable subscription in a try-catch, so that we can delete the
// subscription if something goes wrong.
try {
    // Create a Pulsar producer on the to-device topic for the tenant.
    final Producer<byte[]> producer = client.newProducer(Schema.BYTES)
        .topic(MessageFormat.format("persistent://{0}/mqtt/to-device", tenantID))
        .create();
    System.out.println("Created Pulsar producer");

    // Publish a message to a single MQTT device.
    producer.newMessage()
        .property("clientId", "demoClient")
        .property("topic", "demoTopicB")
        .key("demoClient")
        .value("Message sent to a single device".getBytes(StandardCharsets.UTF_8))
        .send();
    System.out.println("Sent message to single device");

    // Publish a message to all MQTT devices subscribed to a topic.
    // Note that the "clientId" property is omitted here.
    producer.newMessage()
        .property("topic", "demoTopicB")
        .key("demoTopicB")
        .value("Message sent to all subscribed devices".getBytes(StandardCharsets.UTF_8))
        .send();
    System.out.println("Sent message to all subscribed devices");

    // Close the producer.
    producer.close();
}

```

MESSAGING SERVICE QUOTAS AND LIMITS

Messages published to a Pulsar topic are stored persistently by the Messaging Service until they have been delivered to, and acknowledged by, all interested consumers. For messages published to the `from-device` topic by the MQTT Service, the consumers are any clients that have created durable subscriptions on the topic. For messages published to the `to-device` topic by clients, the consumers are the instances of the MQTT Service that will deliver the messages to devices.

To optimize resource usage, the Messaging Service imposes storage limits and a message time-to-live (TTL) on persistently stored messages.

See the [service quotas](#) documentation for details on the default limits. These limits are configurable on a per-tenant basis. If your use case requires a different configuration, or if you have any questions or concerns, contact [product support](#).

Message backlog quota

Persistent messages are stored in a *backlog* until they are delivered to any interested consumers. The maximum size of the backlog is set by the *backlog quota limit*, which directly affects the number of messages that can be stored and therefore the resource consumption of the platform.

A separate backlog exists for each Pulsar topic, so for the MQTT Service the `from-device` and `to-device` topics for a tenant will each have their own independent backlog. The backlog is shared by all subscriptions on a topic. If the backlog quota limit is reached, no new messages can be added to the backlog until some older messages have been delivered, or deleted due to their TTL expiring.

If the backlog quota limit for the Pulsar `from-device` topic is reached, new MQTT `PUBLISH` packets from connected devices will be rejected. If the `PUBLISH` packet was sent with QoS level 0, the message will be lost. If the `PUBLISH` packet was sent with QoS level 1, the behaviour depends on the MQTT protocol version used by the device:

- For devices using MQTT version 3, the device will be disconnected.
- For devices using MQTT version 5, the device will receive a `PUBACK` packet with reason code `0x97`, *Quota exceeded*.

If the backlog quota limit for the Pulsar `to-device` topic is reached, clients calling the `Producer.send()` method, or its equivalent in the Pulsar library used by the client, will receive an appropriate exception or error response from the client library.

Message time-to-live

Any undelivered messages will be automatically deleted if they have been on the backlog for longer than the *time-to-live (TTL) limit*. This policy helps to limit overall resource usage and reduces the need to process outdated data after a prolonged disconnection of a consumer.

No undelivered message will ever be deleted from the backlog unless it reaches its TTL limit. Messages will always be delivered to the consumer in the order they were published to the topic.

BEST PRACTICES FOR RELIABLE MESSAGE DELIVERY FROM DEVICES

If a topic reaches its backlog quota limit, it stops accepting new messages and messages may be lost. To avoid this:

- Process and acknowledge messages from the `from-device` topic as quickly as possible. Every message **must** be explicitly acknowledged, even if the client is not interested in it. Do not acknowledge a message until processing is complete or the message has been stored securely for later processing. Acknowledged messages will not be re-delivered after a client failure or restart.
- Manage subscription lifecycles. Subscribing a consumer creates a *durable subscription* that remains until explicitly deleted. Messages published while the client is disconnected will be retained for the subscription and delivered when the client reconnects. Because subscriptions persist, a topic can reach its backlog quota even when no clients are running.
 1. Use the same subscription name each time the client connects. Avoid creating random subscription names on each run. That leaves inactive subscriptions accumulating and may exhaust the backlog.
 2. Explicitly delete subscriptions when they are no longer required. For example, when taking a client out of service for an extended period, call the consumer `unsubscribe()` method or use the Messaging Service [monitoring and management](#) interface to delete the subscription.

Example code – deleting the subscription

The code snippet below shows how to delete the subscription and close the other Pulsar client objects created by the earlier code examples.

```
finally {
    // Delete the durable subscription.
    // This is only necessary if messages should *not* be retained
    // on the topic while the client is disconnected.
    consumer.unsubscribe();
}

// Close the other Pulsar objects that we created.
consumer.close();
client.close();
```

HANDLING MESSAGING SERVICE ERRORS

The Cumulocity Messaging Service is a complex, distributed service running remotely from your client. In common with all distributed systems, perfect reliability cannot be guaranteed, and a client should be prepared to handle errors reported by the Pulsar client library. These errors can be split into two general categories:

1. Configuration or logical errors in the client implementation. Errors in this category are usually "fatal" and prevent the client from connecting to the Messaging Service, or publishing or consuming any messages. Some typical examples of this type of error include:
 - Attempting to connect with an incorrect Pulsar URL.
 - Using invalid authentication credentials.
 - Using the credentials of a user that is not authorized to access the Messaging Service.
 - Attempting to consume from the `to-device` topic, or publish to the `from-device` topic.
 - Attempting to publish to or consume from any other topic.
 - Attempting to publish incorrectly constructed messages. The most likely cause for this is attempting to publish a message with a payload that was not explicitly created as a byte array.
2. Transient errors in the Messaging Service. Errors in this category usually reflect a temporary issue with the Messaging Service server, that will be resolved either automatically or by administrator action. Some transient errors that a client may experience include:
 - Connections may be dropped when Messaging Service components are restarted during upgrades, or during unplanned outages of the Messaging Service. This will cause publish or consume operations to fail, and it may be necessary to re-connect, or re-establish the producer or consumer, before retrying the operation.

- Published messages will be rejected when the backlog quota limit on the `to-device` topic has been reached. See [reliable delivery best practices](#) for advice on avoiding this situation.
- Published messages may be rejected if other limits or quotas on the Messaging Service are reached.

If your client is using the Java client library, almost all errors will be reported as a `PulsarClientException` thrown by a client library method. In some very rare cases a `SchemaSerializationException` runtime error might also be thrown, if the client has not used the `Schema.BYTES` schema and byte array payloads exclusively. The `PulsarClientException` class has many sub-classes that allow a client to determine the cause of the error more precisely. Other client libraries will have similar language-specific error reporting mechanisms.

In general, it is not possible to recover from a fatal configuration or logic error in the client implementation. The client will need to be restarted after the error has been corrected. For transient errors, a strategy of retrying after a delay is usually appropriate. When an operation on a producer or a consumer has failed, it may be difficult to identify the exact root cause and the optimal response. A simple recovery approach that covers most scenarios is to delete the failed producer or consumer and create a new one before retrying the operation. This avoids cases where the producer or consumer cannot reconnect after an error. A more sophisticated strategy can tailor the response to the specific subclass of `PulsarClientException` thrown. Use an [exponential backoff](#) strategy to increase the delay between retries until the service recovers.

EXAMPLE CLIENT

A complete [example Java client](#) based on the code snippets above can be found in the [cumulocity-examples](#) repository. The `README.md` file provided with the example explains how to build and run it.

The examples repository also contains a simple [Python MQTT client](#) that can be used to simulate an MQTT device and test the operation of the Java client. See the `README.md` file included with the example for more details. Start the Python client first to ensure messages sent to a device are received, then start the Java client.

JAVA CLIENT

⚠ CAUTION

The MQTT Service Java SDK is deprecated and should not be used for new development. It has been replaced by direct connections to the Cumulocity Messaging Service.

Any clients using the Java SDK **must** be updated before the MQTT Service becomes Generally Available.

See [Connecting microservices and applications](#) for more information.

FREQUENTLY ASKED QUESTIONS

Q: How can I obtain device credentials for my MQTT devices?

A: The MQTT Service is not yet integrated with the Cumulocity device bootstrap process. This support is planned for a future release. In the meantime, follow the [Integration lifecycle](#) to bootstrap the device and obtain device credentials. Once the device credentials are obtained, the device can use them to connect to the MQTT Service.

Q: Does the MQTT Service support the SmartREST 2.0 protocol?

A: Not yet. Support for SmartREST 2.0 is planned for a future release.

Q: Why does the MQTT Service not use the standard MQTT ports 1883 and 8883?

A: Those ports are already used by Cumulocity Core MQTT. While both MQTT implementations are operating in parallel, the MQTT Service must use different ports.

Q: What other ways are there to map my MQTT device payloads to Cumulocity, other than a Streaming Analytics app or a custom

microservice?

A: One option is to use the [Dynamic Mapping Service for Cumulocity](#). This is a community-supported open-source component that allows many different payload formats and encodings to be mapped to the Cumulocity domain model. Mappings can be configured using a graphical UI or by writing JavaScript code.

REST

REST is a very simple and secure protocol based on HTTP(S) and TCP. It is today the de-facto Internet standard supported by all networked programming environments ranging from very simple devices up to large-scale IT. One of the many books introducing REST is [RESTful Web Services](#).

This section explains how to use Cumulocity's REST interfaces to integrate devices with Cumulocity. For general information on using REST interfaces and for information on developing applications on top of Cumulocity using REST refer to [Microservice SDK](#).

The description is closely linked to the Cumulocity OpenAPI Specification, which describes each interface in detail, as well as the fragment library:

- [REST implementation](#) is the reference for all general concepts.
- [Fragment library](#) specifies managed objects (devices) as well as for predefined sensor and control capabilities of devices.

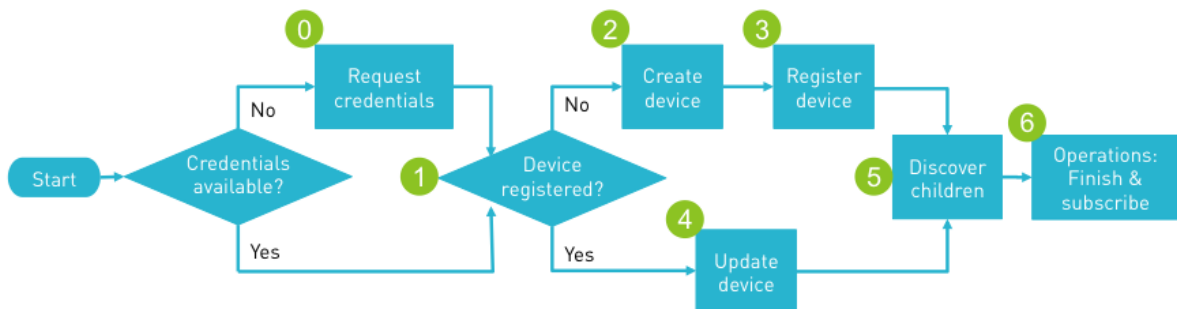
If you develop using Java ME/SE, JavaScript or C/C++, check the relevant chapters in this guide for even more convenient access to Cumulocity's functionality. Also, if you use any of the supported development boards, see the corresponding description in the Devices guide for more information.

INTEGRATION LIFECYCLE

The basic lifecycle for integrating devices into Cumulocity is discussed in [Interfacing devices](#). In this section, we will show how this lifecycle is implemented on REST level. The lifecycle consists of two phases, a startup phase and a cycle phase.

The startup phase connects the device to Cumulocity and updates the device data in the inventory. It also performs cleanup tasks required for operations. It consists of the following steps:

- [Step 0](#): Request device credentials, if not already requested.
- [Step 1](#): Check if the device is already registered.
- [Step 2](#): If not, create the device in the inventory and
- [Step 3](#): Register the device (create the identity).
- [Step 4](#): If yes, update the device in the inventory.
- [Step 5](#): Discover child devices and create or update them in the inventory.
- [Step 6](#): Complete operations that required a restart and subscribe to new operations.



The cycle phase follows. It continuously updates the inventory, writes measurements, alarms and events and executes operations as necessary. It can be considered to be the "main loop" of the device which is executed until the device shuts down. The loop consists of the following steps:

- [Step 7](#): Execute operations.
- [Step 8](#): Update inventory.
- [Step 9](#): Send measurements.
- [Step 10](#): Send events.
- [Step 11](#): Send alarms.



Reference models for the data can be found in the [fragment library](#).

STARTUP PHASE

Step 0: Request device credentials

Every request to Cumulocity must be authenticated, including requests from devices. If you want to assign individual credentials to devices, you can use the device credentials API to generate new credentials automatically. To do so, request device credentials at first startup through the API and store them locally on the device for further requests.

The process works as follows:

- Cumulocity assumes each device to have some form of unique ID. A good device identifier may be the MAC address of the network adapter, the IMEI of a mobile device or a hardware serial number.
- When you take a new device into use, you enter this unique ID into the device registration dialog in the tenant UI and start the device.
- Once started, the device will connect to Cumulocity and send its unique ID repeatedly. For this purpose, Cumulocity provides static bootstrap credentials that can be obtained by contacting [product support](#).
- You can accept the connection from the device in the device registration dialog in the tenant UI, in which case Cumulocity then sends generated credentials to the device.
- The device will store and use these credentials for all further requests.

From a device perspective, this request for credentials is a single REST request:

```
POST /devicecontrol/deviceCredentials
Content-Type: application/vnd.com.nsn.cumulocity.devicecredentials+json
Authorization: Basic <<Base64 encoded bootstrap credentials>>
{
  "id" : "0000000017b769d5"
}
```

The device issues this request repeatedly. While the user has not yet registered and accepted the device in the tenant UI, the request returns "404 Not Found." After the device has been accepted in the tenant UI, the following response is returned:

```

HTTP/1.1 200 OK
Content-Type: application/vnd.com.nsn.cumulocity.devicecredentials+json;charset=UTF-8;ver=0.9
Content-Length: ...
{
  "id": "0000000017b769d5",
  "self": "<<URL of new request>>",
  "tenantId": "test",
  "username": "device_0000000017b769d5",
  "password": "3rasfst4swfa"
}

```

The device can now connect to Cumulocity using the tenant ID, username and password. User alias is not supported for devices.

With the introduction of the concept of Enterprise tenants, it is no longer safe to assume the tenant name is the same as the tenant ID. The credentials request returns the tenant ID only. This cannot be used as the subdomain, combined with the domain name to provide a tenant URL that can be accessed with username only (and password). Access to the correct tenant can only be ensured by using the tenant ID and the username in authentication, for example, `<tenant ID>/<username>` with the password returned by the credentials request. In this case, the subdomain is irrelevant.

Request header should be:

```

Authorization: Basic <<Base64 encoded credentials <tenant ID>/<username>:<password> >>

```

For example, a credentials request for a device added to *xyz.cumulocity.com* could return a user ID, password and a tenant ID of "t123456789". The tenant ID "t123456789" cannot be used as a subdomain (that is, *t123456789.cumulocity.com*) for requests with the user ID and password - it will return "http 403". The tenant ID must be used with the user ID in the form "t123456789/", along with the password. The actual subdomain is then irrelevant. *t123456789.cumulocity.com* or *management.cumulocity.com* or even *anything.cumulocity.com* can be used.

Cumulocity uses the tenant ID specified with the user ID for full authentication and routing of the request to the correct tenant.

If the valid tenant URL is known (for example *xyz.cumulocity.com* as seen in the example above), then the username does not have to be prefixed by `<tenant ID>/` for authentication.

Step 1: Check if the device is already registered

The unique ID of the device is also used for registering the device in the inventory. The registration is carried out using the [Identity API](#). In the Identity API, each managed object can be associated with multiple identifiers distinguished by type. Types are, for example, "c8y_Serial" for a hardware serial, "c8y_MAC" for a MAC address and "c8y_IMEI" for an IMEI.

To check if a device is already registered, use a GET request on the identity API using the device identifier and its type. The following example shows a check for a Raspberry Pi with hardware serial "0000000017b769d5".

```

GET /identity/externalIds/c8y_Serial/raspi-0000000017b769d5 HTTP/1.1

HTTP/1.1 200 OK
Content-Type: application/vnd.com.nsn.cumulocity.externalid+json;charset=UTF-8;ver=0.9
...
{
  "externalId": "raspi-0000000017b769d5",
  "managedObject": {
    "id": "2480300",
    "self": "https://.../managedObjects/2480300"
  },
  "self": "https://.../identity/externalIds/c8y_Serial/raspi-0000000017b769d5",
  "type": "c8y_Serial"
}

```

Note that while MAC addresses are guaranteed to be globally unique, serial numbers for hardware may overlap across different hardwares. Hence, in the above example, we prefixed the serial number with a "raspi-" when registering the device (see Step 3).

In this case, the device is already registered and a status code of 200 is returned. In the response, a URL to the device in the inventory is returned in "managedObject.self". This URL can be used to work with the device later on.

If a device is not yet registered, a 404 status code and an error message is returned:

```
GET /identity/externalIds/c8y_Serial/raspi-0000000017b769d6 HTTP/1.1

HTTP/1.1 404 Not Found
Content-Type: application/vnd.com.nsn.cumulocity.error+json;charset=UTF-8;ver=0.9
...
{
  "error": "identity/Not Found",
  "info": "https://www.cumulocity.com/guides/reference/#error_reporting",
  "message": "External id not found; external id = ID [type=c8y_Serial, value=raspi-0000000017b769d6]"
}
```

Step 2: Create the device in the inventory

If Step 1 above indicated that no managed object representing the device exists, create the managed object in Cumulocity. The managed object describes the device, both its instance and metadata. Instance data includes hardware and software information, serial numbers, and device configuration data. Metadata describes the capabilities of the devices, including the supported operations.

To create a managed object, issue a POST request on the managed objects collection in the Inventory API. The following example creates a Raspberry Pi using the Linux agent:

```

POST /inventory/managedObjects HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.managedobject+json
Accept: application/vnd.com.nsn.cumulocity.managedobject+json
...
{
  "name": "RaspPi BCM2708 0000000017b769d5",
  "type": "c8y_Linux",
  "c8y_IsDevice": {},
  "com_cumulocity_model_Agent": {},
  "c8y_SupportedOperations": [ "c8y_Restart", "c8y_Configuration", "c8y_Software", "c8y_Firmware" ],
  "c8y_Hardware": {
    "revision": "000e",
    "model": "RaspPi BCM2708",
    "serialNumber": "0000000017b769d5"
  },
  "c8y_Configuration": {
    "config": "#Fri Aug 30 09:13:56 BST 2013\nc8y.log.eventLevel=INFO\n..."
  },
  "c8y_Mobile": {
    "imei": "861145013087177",
    "cellId": "4904-A496",
    "iccid": "89490200000876620613"
  },
  "c8y_Firmware": {
    "name": "raspberrypi-bootloader",
    "version": "1.20130207-1"
  },
  "c8y_Software": {
    "pi-driver": "pi-driver-3.4.5.jar",
    "pi4j-gpio-extension": "pi4j-gpio-extension-0.0.5.jar",
    ...
  }
}

HTTP/1.1 201 Created
Content-Type: application/vnd.com.nsn.cumulocity.managedobject+json;charset=UTF-8;ver=0.9
...
{
  "id": "2480300",
  "lastUpdated": "2013-08-30T10:12:24.378+02:00",
  "name": "RaspPi BCM2708 0000000017b769d5",
  "owner": "admin",
  "self": "https://.../inventory/managedObjects/2480300",
  "type": "c8y_Linux",
  "c8y_IsDevice": {},
  ...
  "assetParents": {
    "references": [],
    "self": "https://.../inventory/managedObjects/2480300/assetParents"
  },
  "childAssets": {
    "references": [],
    "self": "https://.../inventory/managedObjects/2480300/childAssets"
  },
  "childDevices": {
    "references": [],
    "self": "https://.../inventory/managedObjects/2480300/childDevices"
  },
  "deviceParents": {
    "references": [],
    "self": "https://.../inventory/managedObjects/2480300/deviceParents"
  }
}

```

The example above contains a number of metadata items for the device:

- "c8y_IsDevice" marks devices that can be managed using Cumulocity's Device Management application.
- "com_cumulocity_model_Agent" marks devices running a Cumulocity agent. Such devices will receive all operations targeted to themselves and their children for routing.
- "c8y_SupportedOperations" states that this device can be restarted and configured. In addition, it can carry out software and firmware updates.

For more information, refer to the [fragment library](#).

If the device could be successfully created, a status code of 201 is returned. If the original request contains an "Accept" header as in the example, the complete created object is returned including the ID and URL to reference the object in future requests. The returned object also include references to collections of child devices and child assets that can be used to add children to the device (see below).

Step 3: Register the device

After the new device has been created, it can now be associated with its built-in identifier as described in Step 1. This ensures that the device can find itself in Cumulocity after the next power-up.

Continuing the above example, we would associate the newly created device "2480300" with its hardware serial number:

```
POST /identity/globalIds/2480300/externalIds HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.externalid+json
Accept: application/vnd.com.nsn.cumulocity.externalid+json
...
{
  "type": "c8y_Serial",
  "externalId": "raspi-0000000017b769d5"
}

HTTP/1.1 201 Created
Content-Type: application/vnd.com.nsn.cumulocity.externalid+json;charset=UTF-8;ver=0.9
...
{
  "externalId": "raspi-0000000017b769d5",
  "managedObject": {
    "id": "2480300",
    "self": "https://.../inventory/managedObjects/2480300"
  },
  "self": "https://.../identity/externalIds/c8y_Serial/raspi-0000000017b769d5",
  "type": "c8y_Serial"
}
```

Step 4: Update the device in the inventory

If Step 1 above returned that the device was previously registered already, we must make sure that the inventory representation of the device is up to date with respect to the current state of the actual device. For this purpose, a PUT request is sent to the URL of the device in the inventory. Note, that only fragments that can actually change must be transmitted. (See [Cumulocity's domain model](#) for more information on fragments.)

For example, the hardware information of a device will usually not change, but the software installation may change. So it may make sense to bring the software information in the inventory up to the latest state after a reboot of the device:

```
PUT /inventory/managedObjects/2480300 HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.managedobject+json
...
{
  "c8y_Software": {
    "pi-driver": "pi-driver-3.4.6.jar",
    "pi4j-gpio-extension": "pi4j-gpio-extension-0.0.5.jar"
  }
}

HTTP/1.1 200 OK
```

❗ IMPORTANT

Do not update the name of a device from an agent! An agent creates a default name for a device so that it can be identified in the inventory, but users should be able to edit this name or update it with information from their asset management.

Step 5: Discover child devices and create or update them in the inventory

Depending on the complexity of the sensor network, devices may have child devices associated with them. A good example is home automation: You often have a home automation gateway that installs a multitude of different sensors and controls installed in various rooms of the household. The basic registration of child devices is similar to the registration of the main device up to the fact, that child devices usually do not run an agent instance (hence the "com_cumulocity_model_Agent" fragment is left out). To link a device with a child, send a POST request to the child devices URL that was returned when creating the object (see above).

For example, assume a child device with the URL "https://.../inventory/managedObjects/2543801" has already been created. To link this device with its parent, issue:

```
POST /inventory/managedObjects/2480300/childDevices HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.managedobjectreference+json
{
  "managedObject" : {
    "id" : "2543801"
  }
}
```

HTTP/1.1 201 Created

Finally, devices and references can be deleted by issuing a DELETE request to their URLs. For example, the reference from the parent device to the child device that we just created can be removed by issuing:

```
DELETE /inventory/managedObjects/2480300/childDevices/2543801 HTTP/1.1
```

HTTP/1.1 204 No Content

This does not delete the device itself in the inventory, only the reference. To delete the device, issue:

```
DELETE /inventory/managedObjects/2543801 HTTP/1.1
```

HTTP/1.1 204 No Content

This request will also delete all data associated with the device including its registration information, measurements, alarms, events and operations. Usually, it is not recommended to delete devices automatically. For example, if a device has just temporarily lost its connection, you usually do not want to lose all historical information associated with the device.

Step 6: Complete operations and subscribe

Each operation in Cumulocity is cycled through an execution flow. When an operation is created through a Cumulocity application, its status is PENDING, that means, it has been queued for executing but it hasn't executed yet. When an agent picks up the operation and starts executing it, it marks the operations as EXECUTING in Cumulocity. The agent will then carry out the operation on the device or its children (for example it will restart the device, or set a relay). Then it will possibly update the inventory reflecting the new state of the device or its children (for example it updates the current state of the relay in the inventory). Then the agent will mark the operation in Cumulocity as either SUCCESSFUL or FAILED, potentially indicating the error.



The benefit of this execution flow is that it supports devices that are offline and temporarily out of coverage. It also allows devices to support operations that require a restart – such as a firmware upgrade. After the restart, the device needs to know what it previously did and hence must query all EXECUTING operations and see if they were successful. Also, it needs to listen what new operations may be queued for it.

To clean up operations that are still in EXECUTING status, query operations by agent ID and status. In our example, the request would be:

```

GET /devicecontrol/operations?agentId=2480300&status=EXECUTING HTTP/1.1

HTTP/1.1 200 OK
Content-Type: application/vnd.com.nsn.cumulocity.operationcollection+json;charset=UTF-8;ver=0.9
...
{
  "next": "https://.../devicecontrol/operations?agentId=2480300&status=EXECUTING",
  "operations": [
    {
      "creationTime": "2013-08-29T19:49:15.239+02:00",
      "deviceId": "2480300",
      "id": "2593101",
      "self": "https://.../devicecontrol/operations/2480300",
      "status": "EXECUTING",
      "c8y_Restart": {
        }
      }
    ],
    "statistics": {
      "currentPage": 1,
      "pageSize": 2000
    },
    "self": "https://.../devicecontrol/operations?agentId=2480300&status=EXECUTING"
  }
}

```

The restart seems to have executed well – we are back after all. So let's set the operation to SUCCESSFUL.

```

PUT /devicecontrol/operations/2593101 HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.operation+json
{
  "status": "SUCCESSFUL"
}

HTTP/1.1 200 OK

```

Then, listen to new operations created in Cumulocity. The mechanism for listening to real-time operations in Cumulocity is described in the [Device control notification API](#) in the Cumulocity OpenAPI Specification and is based on the standard Bayeux protocol. First, a handshake is required. The handshake tells Cumulocity what protocols the agent supports for notifications and allocates a client ID to the agent.

```

POST /notification/operations HTTP/1.1
Content-Type: application/json
...
[ {
  "channel": "/meta/handshake",
  "version": "1.0"
} ]

HTTP/1.1 200 OK
...
[ {
  "minimumVersion": "1.0",
  "clientId": "139jhm07u1dlry92fdl63rmq2c",
  "supportedConnectionTypes": [
    "long-polling",
    "smartrest-long-polling",
    "websocket"
  ],
  "channel": "/meta/handshake",
  "version": "1.0",
  "successful": true
} ]

```

Afterwards, the device respectively the agent must subscribe to notifications for operations. This is done using a POST request with the ID of the device as subscription channel. In our example, the Raspberry Pi runs an agent and has ID 2480300:

```

POST /notification/operations HTTP/1.1
Content-Type: application/json
...
[ {
  "channel": "/meta/subscribe",
  "subscription": "/2480300",
  "clientId": "139jhm07u1dlry92fdl63rmq2c"
} ]

HTTP/1.1 200 OK
...
[ {
  "channel": "/meta/subscribe",
  "subscription": "/2480300",
  "successful": true
} ]

```

Finally, the device connects and waits for operations to be sent to it.

```

POST /notification/operations HTTP/1.1
Content-Type: application/json
...
[ {
  "connectionType": "long-polling",
  "channel": "/meta/connect",
  "clientId": "139jhm07u1dlry92fdl63rmq2c"
} ]

```

This request will hang until an operation is issued (that is, the HTTP server will not answer immediately) but will wait until an operation is available for the device (long polling).

Note that there might have been operations that were pending before we subscribed to new incoming operations. We must query these still. This is done after the subscription to not miss any operations between query and subscription. The technical handling is just like previously described for EXECUTING operations, but using PENDING instead:

```

GET /devicecontrol/operations?agentId=2480300&status=PENDING HTTP/1.1

```


CYCLE PHASE

Step 7: Execute operations

Assume now that an operation is queued for the agent. This will make the long polling request that we issued above return with the operation. Here is an example of a response with a single configuration operation:

```
HTTP/1.1 200 OK
...
[
  {
    "id": "139",
    "data": {
      "creationTime": "2013-09-04T10:53:35.128+02:00",
      "deviceId": "2480300",
      "id": "2546600",
      "self": "https://.../devicecontrol/operations/2546600",
      "status": "PENDING",
      "description": "Configuration update",
      "c8y_Configuration": { "config": "#Wed Sep 04 10:54:06 CEST 2013\n..." }
    },
    "channel": "/2480300"
  }, {
    "id": "3",
    "successful": true,
    "channel": "/meta/connect"
  }
]
```

When the agent picks up the operation, it sets it to EXECUTING status in Cumulocity using a PUT request (see above example for FAILED). It carries out the operation on the device and runs possible updates of the Cumulocity inventory. Finally, it sets the operation to SUCCESSFUL or FAILED depending on the outcome. Then, it will reconnect again to `/notification/operations` as described above and wait for the next operation.

The device should reconnect within ten seconds to the server to not lose queued operations. This is the time that Cumulocity buffers real-time data. The interval can be specified upon handshake.

Step 8: Update inventory

The inventory entry of a device usually represents its current state, which may be subject of continuous change. As an example, consider a device with a GPS chip. That device will keep its current location up-to-date in the inventory. At the same time, it will report location updates as well as events to maintain a trace of its locations. Technically, such updates are reported with the same requests as shown in Step 4.

Step 9: Send measurements

To create new measurements in Cumulocity, issue a POST request with the measurement. The example below shows how to create a signal strength measurement.

```

POST /measurement/measurements HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.measurement+json
...
{
  "source": { "id": "2480300" },
  "time": "2013-07-02T16:32:30.152+02:00",
  "type": "SignalStrength",
  "c8y_SignalStrength": {
    "rssi": { "value": -53, "unit": "dBm" },
    "ber": { "value": 0.14, "unit": "%" }
  }
}

HTTP/1.1 201 Created

```

Step 10: Send events

Similarly, use a POST request for events. The following example shows a location update from a GPS sensor.

```

POST /event/events HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.event+json
...
{
  "source": { "id": "2480300" },
  "text": "Location updated",
  "time": "2013-07-19T09:07:22.598+02:00",
  "type": "LocationUpdate",
  "c8y_Position": {
    "alt": 73.9,
    "lng": 6.151782,
    "lat": 51.211971
  }
}

HTTP/1.1 201 Created

```

Note that all data types in Cumulocity can include arbitrary extensions in the form of additional fragments. In this case, the event includes a position, but also self-defined fragments can be added.

Step 11: Send alarms

Alarms represent events that most likely require human intervention to be solved. For example, if the battery in a device runs out of energy, someone must visit the device to replace the battery. Creating an alarm is technically very similar to creating an event.

```

POST /alarm/alarms HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.alarm+json
Accept: application/vnd.com.nsn.cumulocity.alarm+json
...
{
  "source": { "id": "10400" },
  "text": "Tracker lost power",
  "time": "2013-08-19T21:31:22.740+02:00",
  "type": "c8y_PowerAlarm",
  "status": "ACTIVE",
  "severity": "MAJOR",
}

HTTP/1.1 201 Created
Content-Type: application/vnd.com.nsn.cumulocity.alarm+json
...
{
  "id": "214600",
  "self": "https://.../alarm/alarms/214600",
  ...
}

```

However, you most likely should not create an alarm for a device, if there is a similar alarm already active in the system. Creating many alarms may flood the user interface and may require users to manually clear all the alarms. This is an example for finding the active alarms of our Raspberry Pi from above:

```
GET /alarm/alarms?source=2480300&status=ACTIVE HTTP/1.1
```

In contrast to events, alarms can be updated. If an issue is resolved (for example the battery has been replaced, power has been restored), the corresponding alarm should be automatically cleared to save manual work. This can be done through a PUT request to the URL of the alarm. In the above example for creating an alarm, we used an "Accept" header to get the URL of the new alarm in the response. We can use this URL to clear the alarm:

```

PUT /alarm/alarms/214600 HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.alarm+json
...
{
  "status": "CLEARED"
}

HTTP/1.1 200 OK

```

If you are uncertain on whether to send an event or raise an alarm, you can simply just raise an event and let the user decide with a [Real-time rule](#) if they want to convert the event into an alarm.

REPLACING A PHYSICAL DEVICE

You can replace a physical device that is already connected to the Cumulocity platform while keeping its external ID and the data the device has collected. Do the following:

1. Turn off the old physical device.
2. [Register and bootstrap](#) the new device just like a regular device.
3. After the device has created its new managed object, turn the new physical device off.
4. Open the new device object in [Cumulocity's Device Management application](#) and look up the device owner and the device's external IDs.
5. Remove the external IDs from the device.
6. Open the old device in Cumulocity's Device Management application and change its owner to the one you looked up, and also add the external IDs you removed from the new device.
7. Remove the new device object that was created earlier but keep the device user.
8. Turn on the new physical device.

The new physical device sends its data to the existing managed object.

⚠ CAUTION

The above steps only work if the device is using standard device bootstrapping. Otherwise contact the device integrator or manufacturer.

ℹ INFO

If the device has child devices, their owners must also be updated.

DEVICE AUTHENTICATION

Devices can authenticate to the Cumulocity platform using:

- Device user credentials, that is, using the device username and password,
- Certificate Authentication, that is, using X.509 certificates over a defined REST endpoint protocol to procure the JWT session token on port 8443.

Mutual TLS (mTLS) is a security protocol that uses X.509 certificates for both client and server authentication in a communication session.

The mTLS protocol is commonly used to secure [connections](#) in web services, APIs, and other networked applications. When generating tokens using mTLS, the process involves the authentication of both the client and the server using X.509 certificates.

Retrieving device access tokens from the platform with certificates does not require the tenant ID, username and password. Authentication information will be obtained from the certificates. The device access token can be retrieved by sending only the device leaf certificate if an immediate issuer of the device certificate is uploaded to the trusted certificates list. If the uploaded trusted certificate is not an immediate issuer of the device certificate but belongs to the device's chain of trust, then the device must send the entire certificate chain in the `X-Ssl-Cert-Chain` to be authenticated successfully and retrieve the device access token. You can define which organization you trust by uploading the CA certificate (root/intermediate) to Cumulocity. For details, see [Trusted certificates](#). Alternatively, you can use the Certificate Authority feature. In this case, the root CA certificate is created by Cumulocity, along with a device certificate that is signed by this CA. For details, see [Certificate Authority](#).

JWT SESSION TOKEN RETRIEVAL

The devices can authenticate using X.509 certificates against Cumulocity by using the below endpoint only. In response, a JWT session token is issued by Cumulocity after successful authentication which can later be used to make subsequent requests.

The device access token API can be called by executing the following curl statement:

```
curl -v --cert domain-cert.pem --key domain-private-key.pem \
-H 'Accept: application/json' \
-H 'X-Ssl-Cert-Chain:<device certificate chain>' \
-X POST \
https://<Cumulocity tenant domain>:8443/devicecontrol/deviceAccessToken
```

Replace `<device certificate chain>` with your valid certificate chain when registering with Cumulocity. The header `X-Ssl-Cert-Chain` is not mandatory if you have an immediate issuer of the device certificate in Cumulocity.

```
curl -v --cert domain-cert.pem --key domain-private-key.pem \
-H 'Accept: application/json' \
-X POST \
https://<Cumulocity tenant domain>:8443/devicecontrol/deviceAccessToken
```

You will receive a response like that:

```
HTTP/1.1 200 Success
Content-Type: application/vnd.com.nsn.cumulocity.managedobject+json; charset=UTF-8; ver=0.9
...
{
  "accessToken": "eyJhbGciOiJSUWZlbnQVCj9.eyJktYTJmYy0x...S 04HPk3GQUd-fHyJ2oKSuetWFWpUSBPzJzl_73_3yaullplHorlSoQ"
}
```

A device token is used to access Cumulocity via REST calls. There is no need to send the certificate in subsequent requests once a device token is acquired. A device token lifetime can be configured using tenant options with a category of `oauth.internal` and a key of `device-token.lifespan.seconds`. The default value is 1 hour. The minimum allowed value is 5 minutes. Refer to the [Tenant API](#) in the Cumulocity OpenAPI Specification for more details.

It is recommended that the devices invalidate the session by explicitly calling the `logout` API before closing the HTTP connection. This will avoid any misuse of JWT session tokens generated. Here is the logout API. Refer to the [Users API](#) in the Cumulocity OpenAPI Specification for more details.

```
POST /user/logout
Accept: application/json
Content-Type: application/json
Authorization: Bearer "JWT Session token"
```

⚠ CAUTION

Only devices that are registered to use certificate authentication can retrieve a JWT session token using this endpoint. Once the device successfully authenticates using certificates (that is, by using its private key and the certificate chain), the device retrieves the JWT session token. This mTLS over HTTP endpoint can be leveraged only over this endpoint on port 8443.

REST CLIENT EXAMPLES

HELLO REST

This section gives a very basic example how to create a device representation in Cumulocity and subsequently how to send related measurement data.

All steps are performed by calling REST interfaces. Those REST calls are demonstrated by CURL statements that can be executed on command line.

Refer to [Using the REST interface](#) for a short introduction to CURL.

Prerequisites

In order to follow this tutorial, check if the following prerequisites are met:

- You have a valid tenant, user and password in order to access Cumulocity.
- The command line tool CURL is installed on your system.

Performing the REST calls

We will now perform a sequence of just two REST calls, which are described in detail next:

- Step 1: Create a new device in the inventory of Cumulocity

- Step 2: Transmit measurement data related to that device

In real world those steps are performed by the 'device agent'.

Step one is performed just once, when the device is connected to Cumulocity for the first time.

After that, actions related to that device can be performed by referencing the device by an internal ID which is returned when executing this step.

Create a new device

To create a new device in the inventory of Cumulocity the following REST request is needed:

```
POST /inventory/managedObjects HTTP/1.1
Content-Type: application/vnd.com.nsn.cumulocity.managedobject+json
Accept: application/vnd.com.nsn.cumulocity.managedobject+json
Authorization: Basic <<Base64 encoded credentials <tenant ID>/<username>:<password> >>
...
{
  "c8y_IsDevice" : {},
  "name" : "HelloWorldDevice"
}
```

This call can be done by executing the following curl statement:

```
curl -v -u <username>:<password> \
-H 'Accept: application/vnd.com.nsn.cumulocity.managedobject+json' \
-H 'Content-type: application/vnd.com.nsn.cumulocity.managedobject+json' \
-X POST \
-d '{"c8y_IsDevice":{},"name":"HelloWorldDevice"}' \
https://<Cumulocity tenant domain>/inventory/managedObjects
```

Replace `<username>` , `<password>` and `<tenant-ID>` with the appropriate credentials given to you when registering with Cumulocity.

The same credentials used to access the Cumulocity Web GUI can be used to execute the REST calls.

You will receive a response like that:

```
HTTP/1.1 201 Created
Content-Type: application/vnd.com.nsn.cumulocity.managedobject+json; charset=UTF-8; ver=0.9
Authorization: Basic <<Base64 encoded credentials <tenant ID>/<username>:<password> >>
...
{
  "id": "1231234"
  "lastUpdated": "2014-12-15T14:58:26.279+01:00",
  "name": "HelloWorldDevice",
  "owner": "<username>",
  "self": "https://.../inventory/managedObjects/1231234",
  "c8y_IsDevice": {},
  ...
}
```

When creating a device, Cumulocity generates an ID, which is needed in further calls in order to reference the device. You can find this ID as the "id" attribute-value pair in the response.

Send measurement data

After the device is created, we can send measurement data.

In our case, we will send a temperature measurement in the unit of Celsius which was collected on a certain time:

```

POST /measurement/measurements
Content-Type: application/vnd.com.nsn.cumulocity.measurement+json
Accept: application/vnd.com.nsn.cumulocity.measurement+json
...
{
  "c8y_TemperatureMeasurement": {
    "T": {
      "value": 21.23,
      "unit": "C"
    }
  },
  "time": "2014-12-15T13:00:00.123+02:00",
  "source": {
    "id": "1231234"
  },
  "type": "c8y_PTCMeasurement"
}

```

Replace the id value with the appropriate value you received in the first step.

Furthermore, you should update the time value to a recent timestamp in order to make it easy to find back the measurement on Cumulocity UI later.

Note the data format for timestamp values which is explained as `date-time` in the [Swagger/OpenAPI Specification](#).

```

curl -v -u <username>:<password> \
-H 'Accept: application/vnd.com.nsn.cumulocity.measurement+json' \
-H 'Content-type: application/vnd.com.nsn.cumulocity.measurement+json' \
-X POST \
-d '{"c8y_TemperatureMeasurement":{"T":{"value":21.23,"unit":"C"},"time":"2014-12-15T13:00:00.123+02:00","source":{"id":"1231234"},"type":"c8y_PTCMeasurement"}}' \
https://<Cumulocity tenant domain>/measurement/measurements/

```

The response to that request will look like this:

```

HTTP/1.1 201 Created
Content-Type: application/vnd.com.nsn.cumulocity.measurement+json; charset=UTF-8; ver=0.9
...
{
  "id": "4711",
  "self": "https://.../measurement/measurements/4711",
  "source": {
    "id": "1231234",
    "self": "https://.../inventory/managedObjects/1231234"
  },
  "time": "2014-12-15T12:00:00.123+01:00",
  "type": "c8y_PTCMeasurement",
  "c8y_TemperatureMeasurement": {
    "T": {
      "unit": "C",
      "value": 21.23
    }
  }
}

```

If you like to, you can repeat sending measurements. Before sending the request again, you should update the timestamp (value of attribute 'time') in order to create a time series.

Now you are done. Enter the Device Management application in the Cumulocity UI, select your device on the "All devices" page and switch to the "Measurements" tab. Here you can see your measurement data.

If you do not see data, you might need to change the filter setting to, for example, "last week" to include the timestamp you used in your submitted measurement.

Going further

The sequence of REST calls demonstrated here is just a shortened procedure of those described in [Device integration](#). The first step (creating a new device) is part of the 'startup phase', whereas step two (sending measurements) references to the 'cycle phase'.

Refer to the section on [Device integration](#) to get the necessary information required for implementing real-world agents.

HELLO X509 REST

In this section, we will learn how to generate a JWT token using mTLS protocol with Cumulocity. For authentication with Cumulocity we use X.509 certificates. To learn more about X.509 certificates, refer to [Device certificates](#). The Device access token API is only accessible on port 8443.

Prerequisites

In order to follow this tutorial, check the following prerequisites:

- Verify that you have Maven 3 and at least Java 8 installed.
- Verify that you have OpenSSL installed.

```
$ mvn -v
Maven home: /Library/Maven/apache-maven-3.6.0
Java version: 1.8.0_201, vendor: Oracle Corporation, runtime: /Library/Java/JavaVirtualMachines/jdk1.8.0_201.jdk/Contents/Home/jre
Default locale: en_GB, platform encoding: UTF-8
OS name: "mac os x", version: "10.14.2", arch: "x86_64", family: "mac"
```

Maven can be downloaded from the [Maven website](#).

Copying a Maven project

Copy a Java project which is configured with Maven from [cumulocity-examples](#) repository and execute the following command:

```
$ mvn clean install
```

This will compile the project.

Generating a valid certificate

If you don't have a valid certificate, you can generate one for testing purposes, following the instructions below.

1. Download the scripts from the [cumulocity-examples](#) repository.
2. Create a root self-signed certificate (execute the script *00createRootSelfSignedCertificate.sh*) and upload it to your tenant. You can do it via [the Device Management application in the UI](#) or via [REST](#).
3. Create and sign the certificate (execute the script *01createSignedCertificate.sh*).
4. Move the certificates to keystore (execute the script *02moveCertificatesToKeystore.sh*).
5. Download the public server key from the respective environment and import it into JKS using this command:

```
$ keytool -import -file platform.dev.c8y.io.crt -alias servercertificate -keystore truststore.jks
```

The following configuration is required before calling the device access token API using:

- KEYSTORE_NAME - The path to your keystore which contains the private key and the chain of certificates, which the device uses to authenticate itself.
- KEYSTORE_PASSWORD - The password created for keystore to use its private key.
- KEYSTORE_FORMAT - Either "JKS" or "PKCS12" depending on the file format. The path is provided by KEYSTORE_NAME.
- TRUSTSTORE_NAME - The path to your truststore which contains the certificate of the server.
- TRUSTSTORE_PASSWORD - The password to access the truststore.
- TRUSTSTORE_FORMAT - Either "JKS" or "PKCS12" depending on the file format. The path is provided by TRUSTSTORE.
- PLATFORM_URL - The URL of the platform.
- PLATFORM_MTLS_PORT - Port 8443 is available for device access token API
- DEVICE_ACCESS_TOKEN_PATH API - The endpoint responsible for the mTLS protocol.

- `LOCAL_DEVICE_CHAIN` - The whole chain in PEM format. This header is not mandatory, if you have uploaded the immediate issuer of the device certificate as trusted certificate in Cumulocity.

Changing the configuration

To change the configuration in the REST Java client, copy the file `chain-with-private-key-iot-device-0001.jks` into the resource folder and set the configuration. Note that the script employed (Step 4.) uses the password `changeit`. If you changed the value in the script, also do it for `KEYSTORE_PASSWORD` and `TRUSTSTORE_PASSWORD` in the following example.

```
private static final String KEYSTORE_NAME = "chain-with-private-key-iot-device-0001.jks";
private static final String KEYSTORE_PASSWORD = "changeit";
private static final String KEYSTORE_FORMAT = "jks";
private static final String TRUSTSTORE_NAME = "truststore.jks";
private static final String TRUSTSTORE_PASSWORD = "changeit";
private static final String TRUSTSTORE_FORMAT = "jks";
private static final String LOCAL_DEVICE_CHAIN = "-----BEGIN CERTIFICATE----- MIICQhNJJ0F/lfjm -----END CERTIFICATE-----";
private static final String PLATFORM_URL = "<URL of the platform>";
private static final String PLATFORM_MTLS_PORT = "8443";
```

The device can now generate a JWT token. Note that before the first connect no other actions are required, for example, creating a user. The user is created during the [auto registration](#) process.

INFO

You do not need to set a password, user or tenant for the REST java client using certificates. Cumulocity will recognize the tenant and the user by the provided certificate.

After filling in this data, the example client uses the provided data to retrieve the device access token to the specified platform using certificates.

In general, the mTLS protocol client uses the Java Secure Socket Extension, which is part of the Java Development Kit, to provide secure connections via SSL. JSSE provides the Java implementation of the SSL and TLS protocol which can be configured by developers using its classes. The documentation of the Java Secure Socket Extension shows how the SSL connection is established and provides some examples of customizing the implementation. The full document is available on the [official Oracle website](#). Cumulocity mTLS protocol supports secured SSL connections.

What does the code in `main` do?

- Configure the mTLS connection.
- Connect with Cumulocity via a mTLS protocol.
- Generate a JWT token after successful authentication using X.509 certificates.
- Using this JWT token the further rest operation can be done without any certificates.

Building and running the application

Use the following commands to build the application:

```

$ cd x509-rest-client
$ mvn clean install
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ x509-rest-client ---
[INFO] Building jar: /home/schm/Pulpit/device-jwt-rest-client/target/x509-rest-client-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ x509-rest-client ---
[INFO] Installing /home/schm/Pulpit/x509-rest-client/target/x509-rest-client-1.0-SNAPSHOT.jar to
/home/schm/.m2/repository/c8y/example/x509/x509-rest-client/1.0-SNAPSHOT/x509-rest-client-1.0-SNAPSHOT.jar
[INFO] Installing /home/schm/Pulpit/x509-rest-client/pom.xml to /home/schm/.m2/repository/c8y/example/x509/x509-rest-client/1.0-
SNAPSHOT/x509-rest-client-1.0-SNAPSHOT.pom
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2.642 s
[INFO] Finished at: 2017-03-14T09:16:25+01:00
[INFO] Final Memory: 14M/301M
[INFO] -----

```

and this command to run it:

```

$ mvn exec:java -Dexec.mainClass="c8y.example.x509.X509RestClient"
...
[INFO]
[INFO] -----
[INFO] Building x509-rest-client 1.0-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ x509-rest-client ---
access_token ="eyJhbGciOiJIU6IkpXVCJ9.eyJkYXRmYy0x...S 04HPk3GQUd-fHyJ2oKSuetWFWpUSBPzJzl_73_3yauIlplHorlSoQ"

```

OPC UA

INTRODUCTION

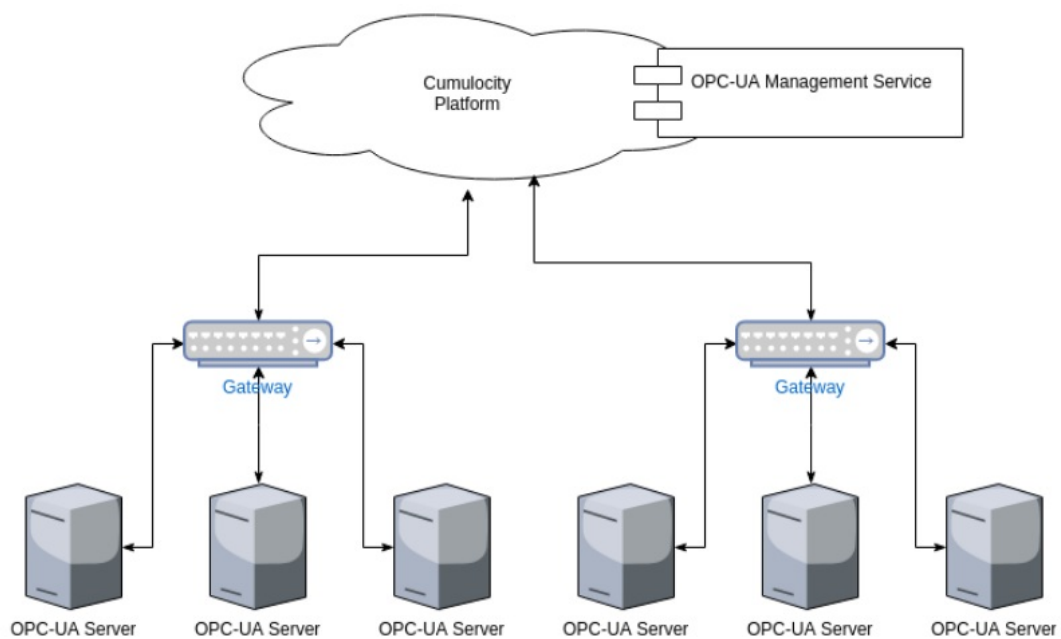


Toggle on the preview feature documentation to see upcoming changes to the existing functionality.

Show preview feature documentation

OPC Unified Architecture (OPC UA) is a standard pushed by the OPC Foundation for industry automation. The goal of OPC UA is to enable the communication between industrial devices. OPC UA is designed to work across technology boundaries (cross-platform). There are two components designed to accomplish this integration:

- OPC UA device gateway
- OPC UA management service



The OPC UA device gateway is a stand-alone Java program that communicates with OPC UA server(s) and the Cumulocity platform. It stores data into the Cumulocity database via REST. Additionally, C8Y commands are executed to perform various operations on the OPC UA servers.

The gateway must be registered as Cumulocity device in a specific tenant and the opcua-device-gateway must run in the users' environment.

❗ IMPORTANT

In order to use OPC UA, you must be subscribed to the "opcua-mgmt-service" microservice. If the "opcua-mgmt-service" microservice is not available in your tenant please contact [product support](#).

To download the gateway navigate to [Cumulocity resources](#).

The gateway requires Java 17 in order to run.

GATEWAY CONFIGURATION AND REGISTRATION

YAML file and spring profiles are used for the configuration of the gateway. A default configuration file is embedded in the gateway JAR file, so you only must set the properties which are different from the default.

! IMPORTANT

When editing the YAML file, make sure to provide valid indentations.

To run the gateway locally, the default settings should be overridden in a customized profile. To use the customized profile, create a YAML file which must follow the naming convention:

```
application-<<Profile_name>>.yaml
```

For example, to connect to a tenant, first a profile named *application-myTenant.yaml* will be created. The following properties will be added to the file:

```
C8Y:
  baseUrl: https://<<yourTenant>>.cumulocity.com
gateway:
  bootstrap:
    tenantId: <<yourTenantId>>
  identifier: Gateway_Device
  name: Gateway_Device
  db:
    # The gateway uses the local database to store platform credentials and local cache.
    # This parameter shows the location in which the local data should be stored.
    baseDir: C:/Users/<<userName>>/opcua/data
```

i INFO

Windows OS is used for the example.

THIN-EDGE.IO

The OPC UA gateway can also be registered and operated via [thin-edge.io](#). In contrast to the standalone mode, *thinEdge* configurations must be added to the YAML file:

Recommended configuration using the [thin-edge.io](#) Cumulocity proxy

i INFO

This requires to be run with the thin-edge.io version 1.7.1 or higher.

The recommended mode of integration uses the [thin-edge.io proxy](#). The local proxy of thin-edge.io exposes the Cumulocity API. By default, the proxy is available at <http://localhost:8001/c8y>.

Example configuration for the thin-edge.io proxy

```
C8Y:
  baseUrl: http://localhost:8001/c8y # Points to the thin-edge.io proxy
gateway:
  bootstrap:
    tenantId: <<yourTenantId>>
  identifier: Gateway_Device
  name: Gateway_Device
  db:
    # The gateway uses the local database to store platform credentials and local cache.
    # This parameter shows the location in which the local data should be stored.
    baseDir: C:/Users/<<userName>>/opcua/data
  thinEdge:
    enabled: true
    useHttpProxy: true
    deviceId: Thin-Edge_Device
```

With the configuration `gateway.thinEdge.enabled: true` you switch to the thin-edge.io mode. This means that the authentication and registration to the platform will be done via thin-edge.io. The OPC UA gateway is automatically registered and created as a subdevice under the thin-edge.io device defined with `gateway.thinEdge.deviceId`.

`gateway.thinEdge.useHttpProxy` is a switch that makes the opcua-device-gateway fully use the thin-edge.io proxy, including authentication to the platform. It requires `C8Y.baseUrl` to be set to the thin-edge.io proxy URL.

Example legacy thin-edge.io configuration (deprecated)

The legacy thin-edge.io mode recreates the authentication credentials for the thin-edge.io connection. This mode is now deprecated.

```
C8Y:
  baseUrl: https://<<yourTenant>>.cumulocity.com
gateway:
  bootstrap:
    tenantId: <<yourTenantId>>
  identifier: Gateway_Device
  name: Gateway_Device
  db:
    # The gateway uses the local database to store platform credentials and local cache.
    # This parameter shows the location in which the local data should be stored.
    baseDir: C:/Users/<<userName>>/opcua/data
  thinEdge:
    enabled: true
    # URL for the MQTT client to connect to the local thin-edge.io MQTT broker.
    mqttServerURL: tcp://<<thinEdge MQTT broker>>
    deviceId: Thin-Edge_Device
```

PREVIEW

MQTT FORWARDING MODE

The OPC UA gateway supports an MQTT Forwarding mode that can be used together with the thin-edge.io mode. In addition to the OPC UA gateway being registered as a child device of the thin-edge.io device and the OPC UA gateway using credentials provided by thin-edge.io, in MQTT Forwarding mode the OPC UA gateway also uses thin-edge.io to send the data it receives from OPC UA servers to Cumulocity. When using cyclic reads, the data received in a single cyclic read that is mapped to measurements, events, or custom actions can be batched into a single message.

The MQTT Forwarding mode uses the existing `thinEdge` configuration and introduces a number of additional configuration options to the YAML file:

```
C8Y:
  baseUrl: http://localhost:8001/c8y
gateway:
  bootstrap:
    tenantId: <<yourTenantId>>
  identifier: Gateway_Device
  name: Gateway_Device
  db:
    # The gateway uses the local database to store platform credentials and local cache.
    # This parameter shows the location in which the local data should be stored.
    baseDir: C:/Users/<<userName>>/opcua/data
  mappings:
    mergeCyclicRead: false
    mergedEventType: c8y_OpcuaEvent
    mergedMeasurementType: c8y_OpcuaMeasurement
  thinEdge:
    enabled: true
    useHttpProxy: true
    mqttServerURL: tcp://<<thinEdge MQTT broker>>
    deviceId: Thin-Edge_Device
    useForDataForwarding: true
    mqttAutomaticReconnect: true
    mqttCleanSession: true
    mqttConnectionTimeout: 30
    mqttKeepAliveInterval: 60
    mqttMaxInFlight: 1000
```

The configuration `gateway.thinEdge.useForDataForwarding` controls if MQTT Forwarding mode is enabled. The following configurations are optional and control the behavior of the MQTT client:

- `gateway.thinEdge.mqttServerURL` (default: `tcp://127.0.0.1:1883`) - URL for the MQTT client to connect to the local thin-edge.io MQTT broker.
- `gateway.thinEdge.mqttAutomaticReconnect` (default: `true`) - controls if the MQTT client will reconnect in case it loses connection to the MQTT server.
- `gateway.thinEdge.mqttCleanSession` (default: `true`) - controls if the MQTT client should remember state across sessions or start with a clean session.
- `gateway.thinEdge.mqttConnectionTimeout` (default: `30`) - connection timeout in seconds.
- `gateway.thinEdge.mqttKeepAliveInterval` (default: `60`) - keep alive interval in seconds.
- `gateway.thinEdge.mqttMaxInFlight` (default: `1000`) - maximum number of unacknowledged messages in the MQTT client. If this limit is reached, additional messages will fail.

For cyclic reads the configuration `gateway.mappings.mergeCyclicRead` can be enabled. The default is false. If this configuration is enabled cyclic reads mapped to measurements, events, or custom actions in a device protocol that use the same data reporting are merged into single messages. For measurements and events, the type can be controlled by the `gateway.mappings.mergedMeasurementType` and `gateway.mappings.mergedEventType` configuration. This is optional, and if not configured `c8y_OpcuaEvent` and `c8y_OpcuaMeasurement` respectively are used.

CONFIGURATION PROFILE LOCATION ON THE FILESYSTEM

The configuration profile can be stored either in the *same directory as the JAR file* or in a *default configuration directory*. Depending on the operating system, the following default configuration directories can be used:

```
Windows OS
  /C:/opcua/
Linux OS
  /etc/opcua/
  /etc/opcua/data
Mac OS
  /opt/opcua/
  /opt/opcua/data
```

The number of profiles you may have is not limited. To use a specific profile on runtime, the “--spring.profiles.active” JVM argument must be passed when running the gateway JAR file. For example, let's use the previously created profile. Start a terminal and use the following command:

```
java -jar opcua-device-gateway.jar --spring.profiles.active=default,myTenant
```

The command above will start a gateway with the default profile and it will override the default properties with the properties defined in the “myTenant” profile. The list of profiles must be provided as an ordered, comma-separated list. The default profile must always be the first profile in the list.

Optional: To specify your own configuration, Spring arguments can be used in your terminal to run the gateway JAR file. Multiple locations must be comma-separated. The configuration locations should be either YAML files or directories. In case of directories, they must end with “/”. For example:

```
java -jar opcua-device-gateway.jar --spring.config.location=file:<<location>>/.opcua/conf/application-myTenant.yaml,file:<<location>>/.opcua/conf/
```

If both arguments “--spring.config.location” and “--spring.profiles.active” are provided, the configuration locations should be directories instead of files. Otherwise, the profile-specific variants will not be considered.

ADDITIONAL CUSTOMIZATIONS

INFO

If no additional customizations are required, you can skip this section.

The following properties can be manually configured in the YAML file:

```
# Name of the application - this should not change
name: opcua-device-gateway
# Platform location and configuration
C8Y:
  # This is the base URL pointing to the Cumulocity platform. This must always be customized in an application profile.
  baseUrl: http://localhost
  # This is an internal setting of the Cumulocity SDK. It is set to true, because we typically
  # want to configure the Cumulocity SDK to always use the baseUrl provided during initialization.
  # Otherwise, the gateway would use the links in the `self` fragment of the core API responses as the host name.
  # This is helpful in deployment scenarios where the Cumulocity instance is
  # reachable only with an IP address.
  forceInitialHost: true

# HTTP proxy host for platform communication
# proxyHost: your.proxy.host

# HTTP proxy port for platform communication
# proxyPort: 8080

# Username for HTTP proxy authentication
# proxyUser: yourProxyUser

# Password for HTTP proxy authentication
# proxyPassword: yourProxyPassword

#
# Gateway-specific settings
#
gateway:
  # The version of the gateway - this is filled automatically during the build process - do not change this property
  version: ${project.version}
  # The following two properties will be set to the name of the user that is running the gateway unless it's overridden manually
```

```

identifier: mygateway
name: mygateway
# The gateway uses a local database to store platform credentials and a local cache. This setting tells
# where local data is stored.
db:
  baseDir: ${user.home}/.opcua/data
# These settings configure and enable/disable thin-edge.io mode (registration and operating OPC UA gateway via thin-edge.io).
thinEdge:
  # Enable thin-edge.io if the OPC UA gateway is running next to thin-edge.io and should use it to connect to Cumulocity.
  # Set enabled to false if the OPC UA gateway is running without thin-edge.io.
  enabled: false
  # MQTT Server URL of thin-edge.io (localhost).
  mqttServerURL: tcp://127.0.0.1:1883
  # Enable this if the MQTT client uses a single steady connection. Note that MQTT is only used to retrieve the JWT, which is dependent on how long
  # the JWT is valid. See https://cumulocity.com/guides/device-integration/mqtt/#jwt-token-retrieval.
  # We recommend you to use a steady connection only if the JWT is valid for a short time. If the JWT is valid for a longer time, the standard is one
  # hour. It is generally not recommended to have a steady MQTT connection.
  mqttSteadyConnection: false
  # The thin-edge.io deviceId must be changed, depending on the configured deviceId of the thin-edge.io certificate.
  deviceId: my-thin-edge-device
# These settings control the device bootstrap process of the gateway.
# In general, the default settings are sufficient, and should not be changed.
# Contact product support (https://cumulocity.com/guides/<latest-release>/additional-resources/contacting-support/).
# in case the bootstrap credentials are different.
bootstrap:
  # Tenant ID to be used for device bootstrap
  tenantId: management
  # Credentials for the device bootstrap user
  username: devicebootstrap
  password: <devicebootstrap user password>
  # When the gateway starts, it waits <delay> milliseconds before connecting to the platform and searching for
  # the device.
  delay: 5000
  # If set to true, the gateway will drop any stored device credentials and fetch new ones from the platform.
  force: false

# Scheduled tasks and thread pools configuration
# Only change the settings here if really necessary. Wrong scheduler configurations can
# disturb the gateway's operation.
scheduler:
  # Threadpool specific settings
  threadpool:
    # This setting corresponds to the size of the threadpool used for periodic tasks.
    size: 15
  # These settings control the threadpool of our internal task executor, which is used for generic background
  # execution and asynchronous tasks.
executor:
  threadpool:
    coreSize: 30
    maxSize: 60
  # The following settings control the settings of the device type mappings execution.
mappingExecution:
  # This section contains all settings related to external, custom-action execution.
  http:
    # Connection request timeout (milliseconds)
    connectionRequestTimeout: 3000
    # Connection timeout (milliseconds)
    connectionTimeout: 3000
    # Socket timeout (milliseconds)
    socketTimeout: 5000
    # Maximum number of connections via HTTP route
    maxPerRoute: 100
    # Maximum total size of the HTTP connection pool used for external, custom actions.
    maxTotal: 100
    # The inactivityLeaseTimeout setting defines a period, after which persistent connections to
    # the HTTP server must be reevaluated. See PoolingHttpClientConnectionManager for more information
    inactivityLeaseTimeout: 5000

```



```

inactivityLeaseTimeout: 50000 #ms
# Aggregate number of alarms if something goes wrong with the execution of external custom actions
failureAlarmAggregate: true
# How often is the alarm aggregation for failed external calls invoked?
failureAlarmFixedDelay: 15 # seconds
failureHandling:
# Whether a failed HTTP POST should be retried later or not. This can be overridden by the configuration in device type. Default is false
enabled: false
# Number of retries a failed HTTP POST will be resent
maxRetries: 5
# If retry is enabled, the exceptions of HTTP status codes can be provided here, comma separated. A HTTP POST which failed with one of these
codes will not be retried. This can be overridden by the configuration in the device type. Default is empty which means that all failed http posts will be
retried if enabled. Example: 400,500
noRetryHttpCodes:
# Minimum delay in seconds between two retries
retryDelay: 120
# Max queue size of the HTTP POST actions queue
maxQueueSize: 50000
# Worker thread (which performs the actual HTTP request) pool size
threadPoolSize: 200

# Threadpool configuration for the mapping execution
# Each value arriving in the gateway will be handled by one or more action handlers defined in the device type. Each handler will be executed in one
single thread.
# Hence, this threadpool must be large enough to cope with the parallel processing needs of values
# received from the OPC UA server.
threadpool:
size: 200

# Mapping-specific settings
mappings:

# In OPC UA, alarm severity is specified by an integer range between 0 and 1000. The alarmSeverityMap
# allows to configure how OPC UA severity is mapped into Cumulocity severity levels. The following is the default mappings:
# alarmSeverityMap:
# 1001: CRITICAL
# 801: CRITICAL
# 601: MAJOR
# 401: MINOR
# 1: WARNING

# Mapping synchronization interval
# The OPC UA gateway periodically fetches the OPC UA device types. With the following settings, this
# interval can be adjusted.

# Sync interval in milliseconds. The default is 43200000ms (12 hours)
syncInterval: 43200000

# Operation settings
operation:
# Default behavior that controls if the OPC UA gateway performs an address space scan when it connects the first time to an OPC UA server. Can
be overridden in the OPC UA Server config.
autoScanAddressSpace: true
# Validates if the nodes given for the operation belongs to device's address space. If the validation fails an alarm is created for the device. If disabled,
the opcua-device-gateway will execute the operation directly. The default is set to true.
validateDeviceOperationNodes: true
# Cyclic-Reader settings
cyclicRead:
# The cyclic readers use a dedicated threadpool to perform periodic read tasks.
threadpool:
# Allows the size of the threadpool for cyclic reads to be configured
size: 30
# How many nodes can be read at once for the cyclic read of the same device protocol, server, root node and the same parameters (rate, max-age).
defaultBulkSize: 1000

# OPC UA subscription settings: These settings allow global OPC UA configuration parameters
# for subscription-based data reporting

```

```

subscription:
# The reporting rate (in milliseconds) corresponds to the publishing rate for monitored items.
reportingRate: 100
# The maxKeepAliveCount specifies the maximum number of OPC UA reporting intervals with no data that
# can be skipped before the OPC UA server sends an empty response to the gateway, informing about
# a yet active, but idle OPC UA subscription.
maxKeepAliveCount: 200
# The lifeTimeCount specifies the maximum number of reporting intervals without a value being sent.
# After the lifetime count has exceeded, the subscription is terminated.
# Must be 3 times greater than maxKeepAliveCount
lifetimeCount: 600
# The notificationBufferSize defines how many monitored item values should be buffered to receive
# subscription notification data from the OPC UA server. The subscription reporting rate (publish interval)
# and the volume of sampling data should be taken into account to choose a suitable buffer size.
notificationBufferSize: 500
# The recreateFailedItems flag can be used to enable the feature of a subscription so that it automatically retries to create the monitored items
# if they fail due to error code Bad_NodeIdUnknown. It assumes that the NodeIds are correct, but it hasn't been added to the
# server's address space yet. The default value is false.
recreateFailedItems: false

# Subscription update settings
subscriptionUpdate:
# The subscription update interval controls how often the OPC UA gateway updates the subscription
# settings for connected OPC UA servers. Expects: Interval duration in milliseconds.
interval: 60000

# Server connectivity configuration
connectivity:
# If autoReconnect in the client configuration is set to false, the gateway tries to reconnect manually.
# triggerManualReconnectOnConnectionDrop can be used to stop the manual reconnect as well if set to false. The default value is true.
triggerManualReconnectOnConnectionDrop: true

# As a default, the OPC UA stack validates the endpoints returned by the OPC UA server. With this
# setting, the default can be toggled.
# This global setting can be individually overridden for each OPC UA server using the
# "validateDiscoveredEndpoints" configuration fragment.
# validateDiscoveredEndpoints: true

# Internal repository configurations
repositories:
# Interval in milliseconds describing how often the repositories are flushed to the platform
flushInterval: 10000
# Threadpool size for the event queue flushing
eventsThreadpool: 30
# Threadpool size for the alarm queue flushing
alarmsThreadpool: 30
# Threadpool for the measurement queue flushing
measurementsThreadpool: 60

# Maximum capacity. If a repository grows over this size, the OPC UA communication will be shut off!
maximumCapacity: 250000

# Re-enable threshold. If OPC UA communication has been disabled due to exceeding maximum capacity, this threshold
# controls when OPC UA communication is enabled again
reenableThresholdSize: 10

# The settings below describe platform-specific connection parameters.
platform:
inventory:
update:
# Default processing mode for inventory managed objects update to the Cumulocity platform.
defaultProcessingMode: QUIESCENT
# Processing mode for inventory update of the gateway device managed objects to the Cumulocity platform.
gateway:
processingMode: QUIESCENT
# Processing mode for inventory update of the OPC UA server device managed objects to the Cumulocity platform.
server:

```

```

server:
  processingMode: QUIESCENT
  # Processing mode for inventory update of value-map managed objects to the Cumulocity platform.
  valuemap:
    processingMode: QUIESCENT
  # Connection pool configuration
  connectionPool:
    # Overall maximum size of the connection pool
    max: 250
    # Max connections used for a single host
    perHost: 150

  # Gateway self-monitoring configuration

  # First, the gateway internally measures different metrics and populates them to the platform.
  # Second, the gateway actively checks if a server connection is active and working by regularly
  # browsing the root node of an OPC UA server.
  monitoring:
    # The interval below in milliseconds configures the frequency of this monitoring task.
    interval: 10000
    # The interval below in milliseconds configures how often we investigate the thread executor queue sizes to prevent overflow
    checkQueueSizes: 10000

  # The OPC UA gateway persists all latest values of an OPC UA server in a dedicated managed object,
  # the so-called value map. These value maps are locally kept on the device for a certain time
  # before being pushed to the platform, allowing for local aggregation of all last-seen values.
  valueMap:
    # The lifetime of a local value map in seconds
    lifeTime: 30

  # How often (in milliseconds) does the gateway check for changes in configured servers.
  # This setting controls how long it takes for the gateway to discover an added or a removed server
  childrenAddedOrRemoveCheck:
    interval: 15000

  # How often (in milliseconds and if enabled) the gateway reads pending operations from the platform.
  shortPolling:
    enabled: true
    fixedDelay: 15000

  # Time in days for which the certificate is valid.
  applicationIdentity:
    validityTime: 3650

  # Timeout scanning address space in minutes and a pause between retries in milliseconds
  scanAddressSpace:
    timeout: 1440
    retries: 5
    pauseMillisForRetry: 700

```

LOGGING

Custom logging configuration can be set during startup by passing the “-logging.config” JVM argument. For more info on how to set up custom logging settings, refer to the [“Logback” documentation](#). A sample logging configuration file may look like this:

```

<?xml version="1.0" encoding="UTF-8"?>
<configuration scan="true" scanPeriod="30 seconds">

  <include resource="org/springframework/boot/logging/logback/defaults.xml" />
  <appender name="FILE"
    class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>/${user.home}/opcua/log/device-gateway.log</file>
    <encoder>
      <pattern>${FILE_LOG_PATTERN}</pattern>
    </encoder>

    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
      <!-- rollover daily -->
      <fileNamePattern>${user.home}/opcua/log/device-agent-%d{yyyy-MM-dd}.%i.log
      </fileNamePattern>
      <timeBasedFileNamingAndTriggeringPolicy
        class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
        <maxFileSize>50MB</maxFileSize>
      </timeBasedFileNamingAndTriggeringPolicy>
      <maxHistory>5</maxHistory>
    </rollingPolicy>
  </appender>

  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>${CONSOLE_LOG_PATTERN}</pattern>
      <charset>utf8</charset>
    </encoder>
  </appender>

  <logger name="com.cumulocity.opcua.client.gateway" level="INFO" />
  <logger name="com.cumulocity" level="INFO" />
  <logger name="c8y" level="INFO" />

  <root level="INFO">
    <appender-ref ref="FILE" />
    <appender-ref ref="STDOUT" />
  </root>
</configuration>

```

DELETION OF GATEWAY

An OPC UA gateway can be associated with multiple OPC UA servers, and the servers can have multiple child devices connected to them. The cleanest approach to delete a gateway is to first delete the OPC UA server managed objects and all its child devices.

The server can be either deleted from the **OPC UA server** tab of the gateway (recommended way of deletion), or from the device list itself. If the server is deleted from the **OPC UA server** tab, then the server managed object and all the address space managed objects are deleted by the OPC UA management service, but the child devices associated with the server must be deleted separately.

On the other hand, if the server is deleted from the device list, then the child devices associated with the server can be deleted by selecting the checkbox **Also delete child devices of this device**. The deletion is detected by the gateway, and the address space managed objects are deleted for the corresponding server. If the gateway is offline, then the address space managed objects will not be removed.

The process of deletion is asynchronous for both cases, so it may take a while to completely remove all the associated managed objects. Thereafter, the gateway can be deleted from the list of devices along with the device user by selecting the checkbox **Also delete associated device owner** “device_<gateway_name>”.

If the gateway is directly deleted from the list of devices before deleting gateway's servers and devices of those servers, by selecting the checkbox **Also delete child devices of this device**, then the server managed object will be deleted, but the corresponding address space objects will not be deleted as they are not children of the gateway.

RUNNING THE GATEWAY

The gateway can run with either default or custom settings. To run the gateway run one of the commands below:

- Default settings and default logging configuration:

```
java -jar opcua-device-gateway.jar
```

- Custom settings and default logging configuration:

```
java -jar opcua-device-gateway.jar --spring.profiles.active=default,PROFILE_NAME
```

- Custom settings and custom logging configuration:

```
java -jar opcua-device-gateway.jar --spring.profiles.active=default,PROFILE_NAME --logging.config=file:PATH_TO_LOGBACK_XML
```

For example, using the profile from the previous section we are going to register the gateway. First, open the terminal and navigate to the location of the gateway.jar file. Next, enter the following command:

```
java -jar opcua-device-gateway.jar --spring.profiles.active=default,myTenant
```

ADJUSTING GATEWAY MEMORY SETTINGS

In certain scenarios it is required to adjust the memory settings of the gateway application. Examples for such scenarios are the integration of servers with very large address spaces or obtaining large amounts of data from servers using high sampling rates.

You can adjust the memory settings of the gateway like with any other Java program. Typically, it is sufficient to increase the initial heap size and the maximum heap size of the gateway process.

- Example: Run the gateway with a minimum heap size of 2 GB and a maximum heap size of 8 GB.

```
java -Xms2g -Xmx8g -jar opcua-device-gateway.jar
```

❗ IMPORTANT

Please adjust the memory settings according to the physical memory available on the gateway host. The maximum heap size must be set in a way that it doesn't consume more RAM than physically available to the gateway. Otherwise, the virtual memory management of the host operating system might start paging, resulting in reduced gateway performance.

PERFORMANCE TUNING FOR LARGE SETUPS

If you're running your setup with a large number of connected OPC UA servers the scan of these nodes can take a long time and may fail with the default settings. There are a lot of other problems like data collection and synchronisation, subscriptions, cyclic reads and flushing data. We recommend you to tune the following settings in the configuration YAML file (values are just examples of a known large setup):

```

gateway:

scheduler:
  threadpool:
    size: 300

executor:
  threadpool:
    coreSize: 600
    maxSize: 1200

```

If cyclic read (only subscription) is used we recommend

```

cyclicRead:
  threadpool:
    size: 3000

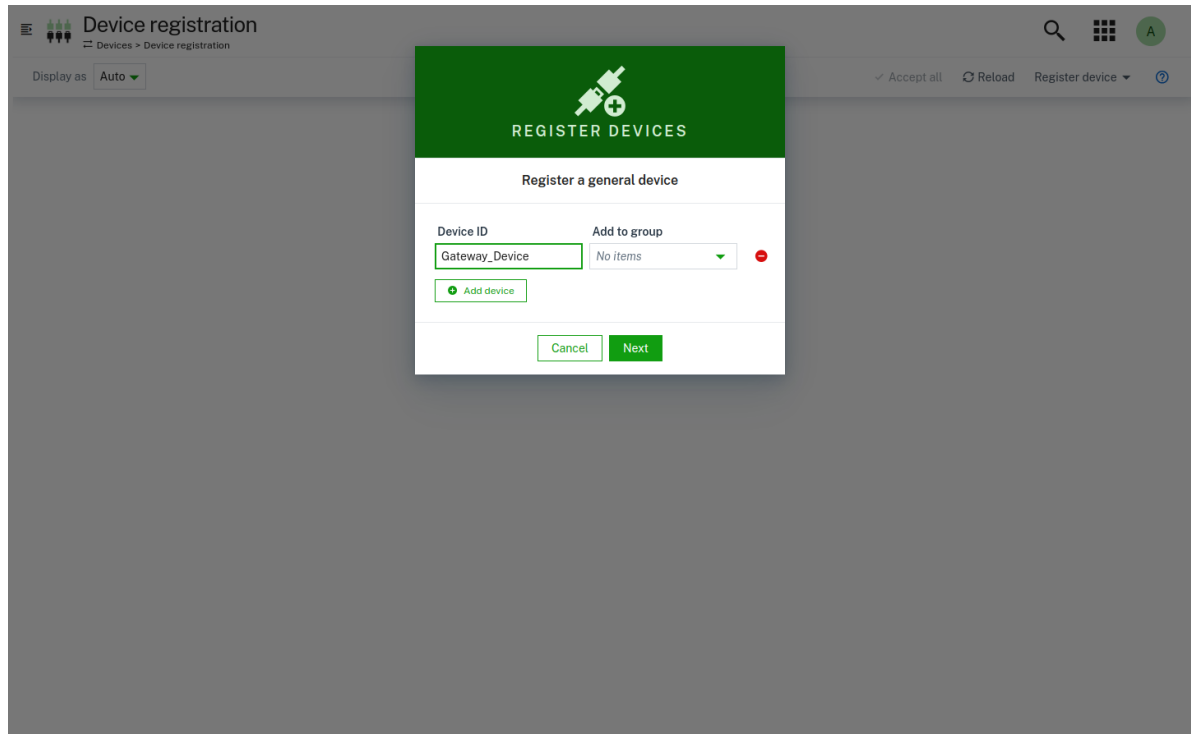
```

otherwise size 0 is perfect.



In general, a larger number of threads will increase your performance. To increase the number of threads, add more memory.

REGISTERING THE GATEWAY AS A CUMULOCITY DEVICE

1. Click **Registration** in the **Devices** menu of the navigator.
2. Click **Register device** at the right of the top bar and from the dropdown menu select **Single registration > General**.
3. Enter the Device ID (in our example it is "Gateway_Device") and then click **Next**.



4. Click **Accept** to complete the registration.

  **Device registration** 1 new device

Devices > Device registration

Display as

Auto

✓ Accept all Reload Register device ?

cucumbers1079558786-opcu...

STATUS

Pending acceptance

SECURITY TOKEN

e.g. A1e3gh5ds

Remove

Accept

CREATED ON

Mar 4, 2025, 12:20:38 PM

BY

admin

INFO

If you run the OPC UA Gateway in thin-edge.io mode, manual registration is not needed. thin-edge.io automatically registers the OPC UA Gateway at Cumulocity as a sub-device of its device.

GATEWAY DEVICE DETAILS

After the registration is completed, the gateway device will be created by the opcua-device-gateway.

In this section, only OPC UA specific information related to the tabs in the device details page will be explained. For more info on all tabs, see [Viewing device details](#).

The screenshot shows the 'cucumberOpcuaGateway' dashboard. The left sidebar contains navigation options: Info, Measurements, Alarms, Control, Availability, Events, OPC UA server, and Identity. The main area is divided into several widgets:

- DEVICE STATUS:** Shows 'Send connection: not monitored' and 'Push connection: inactive'. It includes a 'Required interval' of '--- minutes' and buttons for 'Replace' and 'Delete'.
- DEVICE AND COMMUNICATION:** Lists communication points like 'c8y_LocationUpdate' and 'c8y_UnavailabilityAlarm'.
- DEVICE DATA:** Displays metadata for device '341201', including its name 'cucumberOpcuaGateway', type 'c8y OPCUA_Device_Agent', last updated time, and creation time.
- ACTIVE, CRITICAL ALARMS:** Shows 'No alarms to display' with a link to user documentation.
- GROUP ASSIGNMENT:** Indicates 'Device not assigned' and provides a field to 'Select or search group' with an 'Assign' button.
- LOCATION:** A section at the bottom for location management.

CONNECTING THE GATEWAY TO THE SERVER

Next, establish a connection between the gateway and the OPC UA server.

1. In the **OPC UA server** tab of the respective gateway, click **Add server**.

The screenshot shows the 'OPC UA servers' configuration page. The left sidebar is expanded to 'DEVICE MANAGEMENT' > 'OPC UA server'. The main area shows a list of servers with one entry: 'Milo local' (opc.tcp://localhost:12686/CumulocityTestServer). The configuration details for this server are shown on the right:

- Server name:** Milo local
- Server connection:** Enabled (toggle)
- Connection status:** Connected (status)
- Server URL:** opc.tcp://localhost:12686/CumulocityTestServer
- Security mode:** NONE
- Security policy:** NONE
- Authentication:** Anonymous
- Advanced settings (highlighted with an orange box):**
 - Timeout:** 30 seconds
 - Status check interval:** 40 seconds
 - Auto scan address space:** Checked
 - Address scan type:** Full (radio button selected)

At the bottom, there are buttons for 'Add server', 'Cancel', 'Remove', and 'Save'.

2. Use the **Server connection** toggle, to enable or disable the server connection.
3. Enter the **Server URL** which is used to establish a connection between the server and the gateway.
4. Enter the **Timeout value** in seconds. The timeout value is calculated for each request. If the timeout value is exceeded the request will be unsuccessful.

5. Enter the **Status check interval** in seconds. The status check interval specifies how often the gateway actively checks if the server status has changed. These periodic checks are carried out by reading the *ServerStatus* variable on the OPC UA server.
6. Select the **Security mode** and **Security policy** depending on the server configuration. For more info, see [Security modes](#).
7. Select the desired authentication method. For more info, see [Authentication](#).
8. You can configure the mode of the OPC UA server address scan under **Advanced settings**. If automatic address scanning is enabled the gateway immediately starts the scan after the new server connection is saved.

CAUTION

Be cautious with the address scan type. If it is set to “full”, the gateway will scan the whole OPC UA server address space. Depending on the address space size and the speed of the OPC UA server this can take from minutes to days. We highly recommend you to set the address scan type to “partial” and configure only those node IDs you really want to scan. You can add further node IDs later and do a rescan in the **Address space** tab of the server object afterwards.

9. Click **Save**.

INFO

Once a connection is established, the servers will be located in the **Child devices** tab. In there, the servers will contain additional data such as access to the address space.

SECURITY MODES

The security mode settings tell the gateway how it should secure the connection between itself and the OPC UA server. When a mode other than NONE is selected, the gateway will auto-generate a self-signed application instance certificate and will use it to connect to the server. Possible security mode options are:

- NONE
- BASIC128RSA15_SIGN
- BASIC128RSA15_SIGN_ENCRYPT
- BASIC256_SIGN
- BASIC256_SIGN_ENCRYPT
- BASIC256SHA256_SIGN
- BASIC256SHA256_SIGN_ENCRYPT

INFO

The security modes have nothing to do with authorization or authentication! The security mode tells the gateway how the connection should be secured and whether the connection should be encrypted or not.

AUTHENTICATION

The authentication setting is used to authenticate and authorize the server user. It tells the gateway how to create a user identity and how to send it to the OPC UA server when establishing a connection.

The following authentication methods can be selected:

- Anonymous - Will only work when the OPC UA server allows such connections.
- Username/Password - With this setting the gateway will connect to the server as a specific user represented by a username and

password.

- Key-based authentication - The gateway will use an existing certificate to authenticate as a specific user. JKS keystore must be uploaded to Cumulocity as a binary with type "application/octet-stream". This keystore must follow the following rules:
 - It must be a Java keystore (JKS).
 - The keystore itself must be password-protected.
 - The keystore must contain a user certificate with the "opcuauser" alias.
 - The user certificate must be password-protected.

i INFO

The OPC UA gateway connects as an OPC UA client to the OPC UA server. If key-based authentication is used, the gateway uses a certificate and a corresponding private key to authenticate at the OPC UA server. Both certificate and private key must be stored in a keystore file, using the alias "opcuauser". This way, the gateway precisely can determine which certificate and private key must be used in case a keystore file should contain more data.

The keystore can be created via the following Java keytool command:

```
keytool -genkey -keyalg RSA -alias opcuauser -keystore keystore.jks -storepass passw0rd_a -validity 3600 -keysize 2048
```

With the above command, the key pass is set to the same value as the keystore password.

```
C:\ProgramData>keytool -genkey -keyalg RSA -alias selfsigned -keystore keystore.jks -storepass password -validity 360 -keysize 2048
What is your first and last name?
[Unknown]: John Doe
What is the name of your organizational unit?
[Unknown]: TestOpcua
What is the name of your organization?
[Unknown]: TestOpcua
What is the name of your City or Locality?
[Unknown]: Dusseldorf
What is the name of your State or Province?
[Unknown]:
What is the two-letter country code for this unit?
[Unknown]: DE
Is CN=John Doe, OU=TestOpcua, O=TestOpcua, L=Dusseldorf, ST=Unknown, C=DE correct?
[no]: yes
```

The keystore can be verified by using a tool like KeystoreExplorer. It can then be uploaded to Cumulocity as a binary and used in the server configuration.

The screenshot shows the 'cucumberOpcuaGateway' web interface. On the left is a sidebar with 'DEVICE MANAGEMENT' and navigation links: Home, Devices, Overviews, Groups, Device types, and Management. The 'Management' section is expanded, showing 'OPC UA server' and 'Identity'. The main panel is titled 'cucumberOpcuaGateway' and shows a breadcrumb: 'Devices > All devices > cucumberOpcuaGateway > OPC UA server'. Below this is a sub-header 'OPC UA servers' with a help icon. A message states: 'No servers found. Click below to add a new server.' To the right is a configuration form with the following fields:

- Server URL:**
- Timeout:** **seconds**
- Status check interval:** **seconds**
- Security mode:**
- Security policy:**
- Authentication:**
- Keystore password:**
- Certificate password:**
- Upload keystore:**

At the bottom of the form are four buttons: 'Add server' (green), 'Cancel' (green), 'Remove' (red), and 'Save' (green).

INFO

If you don't have the certificate trusted by your OPC UA server, the server will reject the connection. If you have problems trusting a certificate in your OPC UA server, contact your OPC UA server provider.

INFO

Beside the above authentication certificate, the device gateway also automatically creates a so-called application identity certificate to identify itself with the OPC UA server. This must be trusted by the OPC UA server as well.

CHILD DEVICES

All server connections are listed as child devices even if the servers are disconnected. To stop a server connection, either delete the server child device or disable/remove the connection from the **OPC UA server** tab.

The screenshot displays the 'cucumberOpcuaGateway' interface. On the left is a sidebar with 'DEVICE MANAGEMENT' and a menu including Home, Devices, Overviews, Groups, Device types, and Management. The main area shows the 'Child devices' tab with a table of 1 item. The table has columns: Sta..., Name, Model, Serial nu..., Registrati..., System ID, IMEI, and Alarms. The single entry is 'MyServer' with a registration date of 'Mar 4, 2025, 12:20:50 PM' and System ID '592201'. A bottom bar indicates 'powered by CUMULOCITY'.

Sta...	Name	Model	Serial nu...	Registrati...	System ID	IMEI	Alarms
	MyServer			Mar 4, 2025, 12:20:50 PM	592201		

ADDRESS SPACE

When you navigate to the child device of the gateway, the **Address space** tab shows the attributes and references of the address space node of the servers. The filter searches through the whole hierarchy to find “nodeId”, “browserName” or “displayName” of an attribute. In case of multiple “ancestorNodeIds”, you can click on the desired node to be redirected.

The address space is automatically scanned when a connection between the gateway and the server is established. The duration of the scan depends on the size of the address space. The address space information is stored locally once it is scanned and then used by this applying process. If the address space information is not yet available, for example, the address space has not been scanned, another scan will be triggered without synchronizing data into Cumulocity. Performing another address space operation will update the address space information.

In case a node cannot be read, the scan process skips this node and continues. An error entry is written to the opcua-device-gateway log file to provide information (more information available in debug level).

To manually scan the address space click **Rescan**. You can monitor the progress by checking the rescan operation in the **Control** tab.

The screenshot shows the 'DEVICE MANAGEMENT' interface. On the left is a sidebar with navigation options: Home, Devices, Registration, All devices, Map, Simulators, Availability, Overviews, Groups, Device types, and Management. The main area is titled 'Milo local' and shows a tree view of the device's address space. The tree is expanded to show the 'Address space' section, which includes a 'Filter...' search bar and a list of nodes. A 'Rescan' button is highlighted with a red box at the bottom of the tree. On the right, the 'Attributes' and 'References' sections are visible. The 'Attributes' section shows a table with columns 'ATTRIBUTE' and 'VALUE'. The 'References' section shows a table with columns 'ATTRIBUTE' and 'VALUE'.

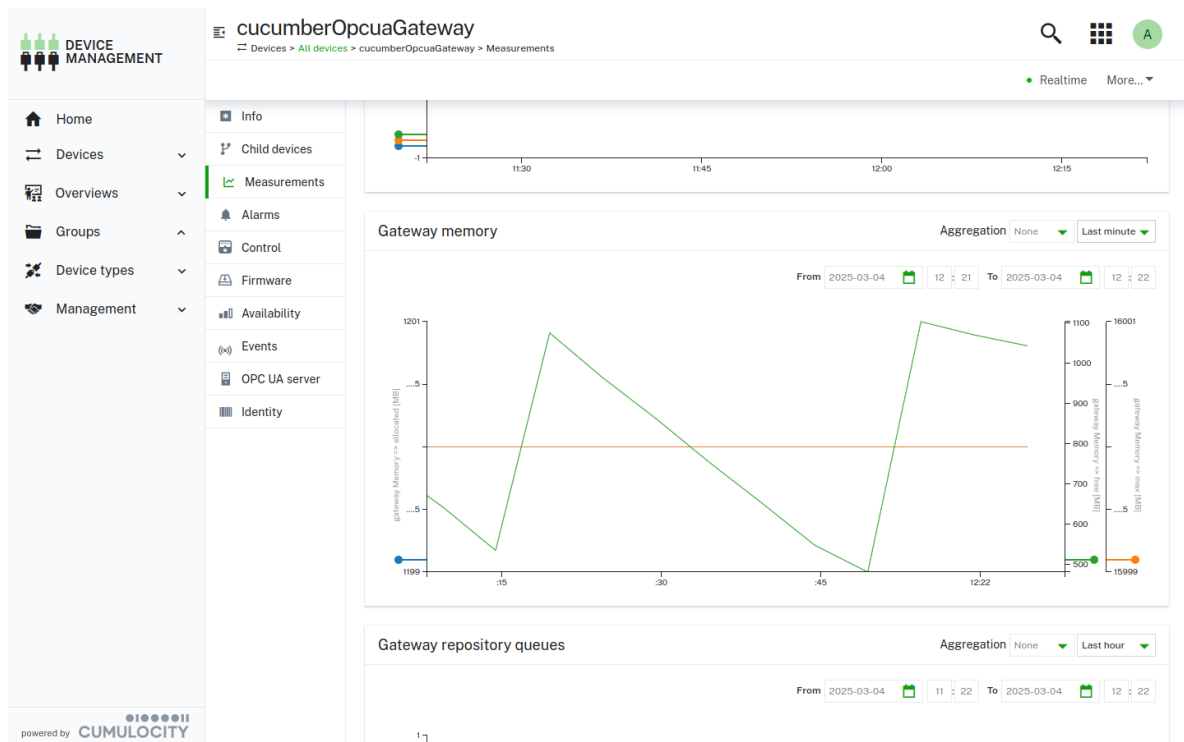
ATTRIBUTE	VALUE
absolutePath	[["O.Root"]]
ancestorNodeIds	[]
browseName	O.Root
description	
displayName	Root
eventNotifier	0
nodeClass	1
nodeClassName	Object
nodeId	i=84

ATTRIBUTE	VALUE
Organizes	Types
Organizes	Objects
Organizes	Views
HasTypeDefinition	Folder Type

MONITORING MEASUREMENTS

On the gateway device, the **Measurements** tab provides visualization of data in the form of charts. In total the gateway contains the following six charts:

Charts	Description
Connected servers	Provides the number of connected and disconnected servers.
Gateway active threads	Shows the number of active threads for the alarm/measurements/event flushes and for the executor. You can also see whether the threadpool size limit is not sufficient, based on the threadpool configurations in the gateway. If the maximum threadpool size is reached then any new activities which require a new thread will be blocked until a thread is available.
Gateway cyclic reads	Number of active cyclic reads done by the gateway. Cyclic reads are actively reading from the OPC UA server within an interval based on the configuration of the device protocol.
Gateway memory	Represents the "free", "max" and "allocated" memory values of the gateway.
Gateway repository queues	Before a thread is flushed it is first added to the queue. This chart shows how many threads are currently in the queue.
Server response time	Shows the response time of each currently connected server.



Monitoring measurement details

The following is the full list of monitoring measurements created by the gateway:

Chart	Measurement type	Measurement series	Unit	Description
Connected servers	c8y_connectedServers	connected servers	num	Number of connected servers
Connected servers	c8y_connectedServers	disconnected servers	num	Number of disconnected servers
Gateway active threads	c8y_gatewayActiveThreads	event_flush	threads	Number of active threads for event flushing
Gateway active threads	c8y_gatewayActiveThreads	alarm_flush	threads	Number of active threads for alarm flushing
Gateway active threads	c8y_gatewayActiveThreads	measurement_flush	threads	Number of active threads for measurement flushing
Gateway active threads	c8y_gatewayActiveThreads	event_flush_queued	threads	Number of queued threads for event flushing
Gateway active threads	c8y_gatewayActiveThreads	alarm_flush_queued	threads	Number of queued threads for alarm flushing
Gateway active threads	c8y_gatewayActiveThreads	measurement_flush_queued	threads	Number of queued threads for measurement flushing


Chart	Measurement type	Measurement series	Unit	Description
Gateway cyclic reads	c8y_gatewayCyclicReads	scheduled_reads	scheduled	Number of cyclic reads that have been scheduled
Gateway cyclic reads	c8y_gatewayCyclicReads	active_reads	threads	Number of active cyclic reads
Gateway cyclic reads	c8y_gatewayCyclicReads	avg_interval	ms	Average cyclic read rate overall
Gateway memory	c8y_gatewayMemory	max	MB	Gateway JVM max memory
Gateway memory	c8y_gatewayMemory	allocated	MB	Gateway JVM total allocated memory
Gateway memory	c8y_gatewayMemory	free	MB	Gateway JVM free memory
Gateway repository queues	c8y_gatewayRepositoryQueues	measurement_queue	measurements	Number of measurements currently in the queue
Gateway repository queues	c8y_gatewayRepositoryQueues	event_queue	events	Number of events currently in the queue
Gateway repository queues	c8y_gatewayRepositoryQueues	alarm_queue	alarms	Number of alarms currently in the queue
Server response time	c8y_serverResponseTime	response_time	ms	OPC UA server response time



MONITORING ALARMS




On the gateway device, the **Alarms** tab shows all alarms raised either on the gateway or on the servers.




There are three alarm types which can be raised:

- Connection loss - If the gateway fails to connect to the OPC UA server a critical alarm is raised.
- Gateway crash - If the gateway crashes or is abruptly shut down a major alarm is raised.
- No data arrived within interval - If the status interval check value in the OPC UA server configuration is exceeded a major alarm is raised.

 cucumberOpcuaGateway

Search  

All severities  No date filter  All alarm types 

 Clear all  Add location 

Info

Child devices

Measurements

Alarms

Control


Firmware


Availability


Events

OPC UA server

Identity

Alarms list Auto refresh 30 30 s 

 No data received from device within required interval.
Mar 4, 2025 12:22:08 PM cucumberOpcuaGateway

 No alarm selected
Select an alarm from the list to view its details.

Monitoring alarm details

The following is the full list of monitoring alarms created by the gateway:

Text	Alarm type	Severity	Note
EventRepository is constantly growing! Possible memory overflow which will result in gateway crash!	c8y_ua_GatewayQueueGrowth_EventRepository	CRITICAL	
AlarmRepository is constantly growing! Possible memory overflow which will result in gateway crash!	c8y_ua_GatewayQueueGrowth_AlarmRepository	CRITICAL	
MeasurementRepository is constantly growing! Possible memory overflow which will result in gateway crash!	c8y_ua_GatewayQueueGrowth_MeasurementRepository	CRITICAL	
Gateway crashed on last run! Please check the log files and memory dumps to see what caused this	c8y_ua_GatewayCrash	MAJOR	This alarm is also raised when the gateway process was not terminated gracefully
Failed to connect to server [{serverId}], reason: {reason}	c8y_ua_ServerConnectionFailed	CRITICAL	This alarm will be cleared by the gateway when the connection to the server has been established successfully
Connection dropped on server: {serverId}	c8y_ua_ConnectionDropped	CRITICAL	This alarm will be cleared by the gateway when the connection has been restored

MONITORING EVENTS

On the gateway device, the **Events** tab shows all events related to the gateway-server connection. Additionally, you can see when the gateway has started and when it ends.

The screenshot shows the 'Events' tab of the 'cucumberOpcuaGateway' interface. The sidebar on the left contains navigation links: Info, Child devices, Measurements, Alarms, Control, Firmware, Availability, Events (highlighted), OPC UA server, and Identity. The main content area displays a table of events. The table has three columns: Date, Event, and Source. Three events are listed, all dated 'Mar 4, 2025, 12:20:54 PM'. The events are: 'Connection established to server: 592201' (Source: MyServer), 'Server [592201] connected' (Source: MyServer), and 'Gateway [cucumbers1079558786-opcuaGateway, cucumberOpcuaGateway] started' (Source: cucumberOpcuaGateway). Above the table, there are filters for 'Date from', 'Date to', and 'Event type', followed by an 'Apply' button. At the top right, there are controls for 'Realtime', 'Reload', 'Add location', and a user profile icon.

Monitoring event details

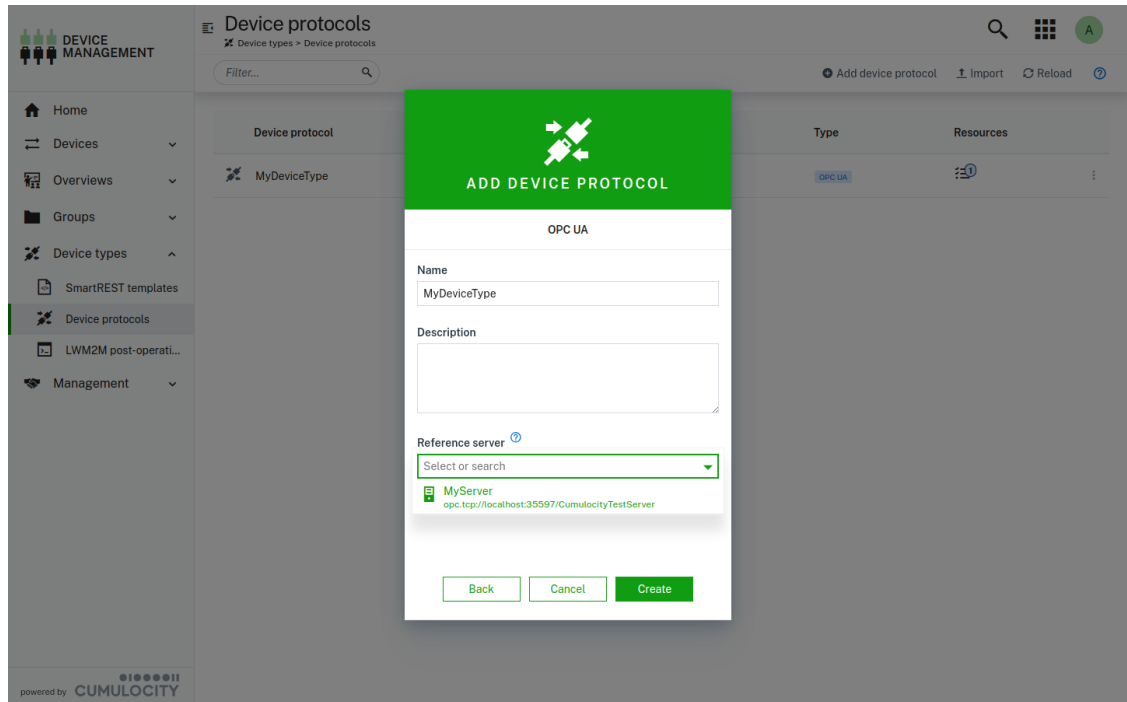
The following is the full list of monitoring events created by the gateway:

Text	Event type	Event source	Description
Gateway [{gateway identifier}, {gateway name}] started	c8y_ua_GatewayStarted	The gateway managed object	This event is created when the gateway has been started and authenticated with the Cumulocity platform
Connection established to server: {server ID}	c8y_ua_ConnectionEstablished	The server managed object	This event is created when the server connection is established - either first time or a reconnection
Server {server ID} connected	c8y_ua_ServerConnected	The server managed object	This event is created when server is connected successfully by the Connection Manager. This event is not created if it is a reconnection. This event is normally followed by an event of type c8y_ua_ConnectionEstablished
Server disconnected: {server ID}	c8y_ua_ServerDisconnected	The server managed object	This event is created when the server is disconnected proactively by the Connection Manager

DEVICE PROTOCOLS

ADDING A NEW DEVICE PROTOCOL

1. In the Device protocols page, click **New device protocol** in the top menu bar and select OPC UA as device protocol type.
2. In the resulting dialog box, enter a name and an optional description for the device protocol.
3. Optionally, a reference server can be selected. Selecting a reference server allows you to create device protocols based on the OPC UA model stored on an OPC UA server. This greatly simplifies the mapping process, as device protocols can be created based on OPC UA browse paths being actually present on the server.
4. Click **Create**.



INFO

Selecting a reference server will require you to select a reference node.

Once the device protocol is created, various configuration settings such as variables, data reporting and constraints can be applied. Initially, the device protocol will be inactive. When active, the gateway will scan the address space of all servers and will automatically apply the device protocol to all nodes which match the criteria. When the device protocol is configured, click **Save**.

ADDING A NEW VARIABLE

1. Click **Add variable** under the **Variables** section.
2. Enter the path and the name of the variable. The path can be the exact browse path or a regular expression of the browse path. If it is a regular expression, it must be wrapped inside `regex(...)`. For example: `regex(2:Objects)` or `regex(urn:test:namespace:Objects\\d)`. Note that the namespace index and the namespace URI are not part of the regular expression itself but they will be quoted as literal strings. When using a regular expression, keep in mind that it might be matching many nodes in the address space and resulting in unexpected incoming data. Our recommendation is to use it with great care and together with other exact browse paths in the same mapping if possible. For example, `["2:Objects"`, `"regex(2:MyDevice\\d)"`, `"..."]`
3. Select either the default or the custom data reporting. The default option uses the data reporting mechanism used in the device protocol. The custom option will let you configure a data reporting mechanism only for the current variable.
4. Additionally, different functionalities such as sending measurements, creating alarms, sending events and custom actions for each variable can be selected.
5. Click **Save** to save your settings.

The gateway has a scheduling job and after the variables are saved, the gateway will check whether the variables exist under the subtree of the node. Afterwards, for each node a child device of the server is created. The child devices will contain data based on the configuration of the device protocol. The node child devices will also be listed in the **All devices** page.

INFO

If no reference server was selected during the device protocol creation, the path should be given with a namespace URI representation. In the OPC UA server the index value can be taken from the namespace array. An example namespace URI representation for browse path "5:Counter1" would be:
<http://www.prosysopc.com/OPCUA/SimulationNodes:Counter1> . A node ID equal to "ns=5;s=Simulation" will have the following namespace representation
[nsu=http://www.prosysopc.com/OPCUA/SimulationNodes;s=Simulation](http://www.prosysopc.com/OPCUA/SimulationNodes;s=Simulation) . In both examples the fifth element of the server's namespace array has a value of
<http://www.prosysopc.com/OPCUA/SimulationNodes> .

The functionalities that can be enabled are the following:

Send measurement

Turn on **Send measurement** to specify a measurement.

Specify the following parameters:

- Enter the type of the measurement, for example, "c8y_AccelerationMeasurement".
- Series are any fragments in measurements that contain a "value" property, for example, "c8y_AccelerationMeasurement.acceleration".
- Specify the unit of the given measurement, for example, "m/s" for velocity.

All measurements which exceed the Java Long ranges for Long.Max_VALUE(9,223,372,036,854,775,807) or Long.MIN_VALUE(-9,223,372,036,854,775,807) are converted internal to double values with scientific notation (for example 9.223372036854778e+24) to ensure the storage in the database. This may result in a less precise rounded value.

Create alarm

Turn on **Create alarm** if you want to create an alarm out of the resource.

Specify the following parameters (all mandatory):

- Severity - One of CRITICAL, MAJOR, MINOR, WARNING
- Type
- Text

INFO

If the value of the mapped resource is "true" (in case of boolean), or a positive number (in case of integer/double), then the alarms are created in ACTIVE state. The alarm de-duplication prevents the creation of multiple alarms with same the source and type, thereby only incrementing the count of the existing alarm. The alarms will be CLEARED as soon as the value is changed to "false", or a number that is less than or equals to 0.

Send Event

Turn on **Send event** to send an event each time you receive a resource value.

Specify the following parameters:

- Enter the type of the event, for example, "com_cumulocity_model_DoorSensorEvent".
- Enter the text which will be sent, for example, "Door sensor was triggered". You can also get the resource value populated to the event text by defining the value placeholder:

Door sensor was triggered, event value: \${value}

The value will also be populated as a fragment of the created event, under a static fragment name as the following:

```
{
  "type": "com_cumulocity_model_DoorSensorEvent",
  "text": "Door sensor was triggered",
  "c8y_ua_DataValue": {
    "serverTimestamp": 132403410091850000,
    "sourceTimestamp": 132403410091850000,
    "value": {
      "value": 381632714
    },
    "statusCode": 0
  }
}
```

The node values under this static fragment that are numeric and exceed the Java long ranges for Long.Max_VALUE(9,223,372,036,854,775,807) or Long.MIN_VALUE(-9,223,372,036,854,775,807) are converted internally to double values with scientific notation (for example 9.223372036854778e+24) to ensure the storage in the database. This may result in a less precise rounded value. However, the node value that is populated for the value placeholder in the event text will have the actual value even if it exceeds the long range.

INFO

The measurements, events and alarms are added to a queue by the gateway, and they are flushed at once to create the respective elements. If the server is deleted, but there are still some items to be flushed, then the request is failed with a response code 403. Thereafter, the exception is handled by validating the existence of the source. If the source is missing then the elements will be removed from the queue.

Custom Actions

Custom actions are HTTP POST requests which the gateway will send to a defined custom URL. You can define custom headers and body template with the following placeholders available:

- `${value}`: value of specific node
- `${receivedTimestampInMs}`: Timestamp when the node value is received by the OPC UA device gateway in milliseconds
- `${serverId}`: ID of OPC-UA server
- `${nodeId}`: ID of source node
- `${deviceId}`: ID of source device

Below there is an example of a full device protocol that configures a custom action:

```
{
  "name": "My device protocol for HttpPost",
  "referencedServerId": "${serverId}",
  "referencedRootNodeId": "ns=2;s=HelloWorld/Dynamic",
  "enabled": true,
  "subscriptionType": {
    "type": "Subscription",
    "subscriptionParameters": {
      "samplingRate": 1000
    }
  },
  "applyConstraints": {
    "matchesNodeIds": [
      "ns=2;s=HelloWorld/Dynamic1"
    ]
  },
  "mappings": [
    {
      "browsePath": [
        "2:Double"
      ],
      "customAction": {
        "type": "HttpPost",
        "endpoint": "http://my.endpoint.local",
        "bodyTemplate": "{\"text\": \"I am coming from Http POST, value: ${value}\", \"type\": \"HttpPostMO\"}",
        "headers": {
          "Authorization": "Basic MYAUTHCREDENTIALS==",
          "Content-Type": "application/json"
        }
      }
    }
  ]
}
```

PREVIEW

DEVICE PROTOCOL BEHAVIOR IN MQTT FORWARDING MODE

If **MQTT Forwarding mode** is enabled, the configured functionalities in device protocols behave differently:

Send measurement (MQTT Forwarding mode)

Measurements are sent to the MQTT topic `te/<identifier>/m/<measurement-type>` of thin-edge.io. As thin-edge.io only supports measurement values in measurements, the measurement units configured in the device protocol are ignored. Also, the standard fragments of the OPC-UA Gateway and additionally configured static fragments are not populated.

Create alarm (MQTT Forwarding mode)

Alarms are created by sending them to the `te/<identifier>/a/<alarm-type>` MQTT topic of thin-edge.io. Alarms are cleared by sending an empty message to the same topic. The logic to clear and deduplicate alarms is unchanged.

Send event (MQTT Forwarding mode)

Events are sent to the `te/<identifier>/e/<event-type>` MQTT topic of thin-edge.io.

Custom actions (MQTT Forwarding mode)

Custom actions in MQTT Forwarding mode use the same body template mechanism. The endpoint of the custom action should be a valid MQTT topic. Headers are ignored.

Send measurement (MQTT Forwarding mode, merging enabled)

Merging measurements is only supported for cyclic reads but not for subscriptions. Variables coming in the same cyclic read (meaning they use the same data reporting in a device protocol) are sent as a single, multi-value measurement to the `te/<identifier>/m/<measurement-type>` of thin-edge.io. The configured type and series for the variable will be used as fragment and series in the measurement. If multiple variables are configured for the same type, they will be consolidated into a single fragment.

The measurement type of the measurement is `c8y_OpcuaMeasurement` unless `gateway.mappings.mergedMeasurementType` in the configuration has been set to a different type.

Send event (MQTT Forwarding mode, merging enabled)

Merging events is only supported for cyclic reads but not for subscriptions. Variables coming in the same cyclic read (meaning they use the same data reporting in a device protocol) are sent as a single event with multiple fragments to the `te/<identifier>/e/<event-type>` of thin-edge.io. The configured type for the variable is used as the fragment name. While it is allowed to use the same event type for multiple variables in a device protocol, here this will result in variables overwriting each other and hence generally discouraged.

The event type of the event is `c8y_OpcuaEvent` unless `gateway.mappings.mergedEventType` in the configuration has been set to a different type.

Custom actions (MQTT Forwarding mode, merging enabled)

Merging custom actions is only supported for cyclic reads but not for subscriptions. Variables coming in the same cyclic read (meaning they use the same data reporting in a device protocol) are sent as a single measurement. For each variable, the body template of the configured custom action is applied, and the results are sent as the string representation of a JSON array.

The expectation is that all custom actions use the same endpoint which is used as the MQTT topic to send the message to. If this is not the case, the endpoint from the first configured variable of the device protocol is used.

MONITORING EVENTS FOR DEVICE PROTOCOL APPLICATION

When a device protocol has been applied to or un-applied from a node, a monitoring event is generated as the following:

Device type has been applied

- Event type - `c8y_ua_DeviceTypeApplied`
- Event text - *Device type: {device type ID} is applied to root node: {root node ID} of server: {server ID}*
- Event source - The server managed object

DEVICE MANAGEMENT

cucumberOpcuaGateway

Devices > All devices > cucumberOpcuaGateway > Events

Date from Date to Event type Apply Realtime Reload Add location

Home Devices Overviews Groups Device types Management

Info Child devices Measurements Alarms Control Firmware Availability Events OPC UA server Identity

Date	Event	Source
Mar 4, 2025, 12:22:16 PM	Device type: 491852 is applied to root node: ns=2;s=HelloWorld/Dynamic2 of server: 592201	MyServer
Mar 4, 2025, 12:22:17 PM	Device type: 491852 is applied to root node: ns=2;s=HelloWorld/Dynamic5 of server: 592201	MyServer
Mar 4, 2025, 12:22:17 PM	Device type: 491852 is applied to root node: ns=2;s=HelloWorld/Dynamic8 of server: 592201	MyServer
Mar 4, 2025, 12:22:17 PM	Device type: 491852 is applied to root node: ns=2;s=HelloWorld/Dynamic of server: 592201	MyServer
Mar 4, 2025, 12:22:16 PM	Device type: 491852 is applied to root node: ns=2;s=HelloWorld/Dynamic1 of server: 592201	MyServer
Mar 4, 2025, 12:22:16 PM	Device type: 491852 is applied to root node: ns=2;s=HelloWorld/Dynamic4 of server: 592201	MyServer
Mar 4, 2025, 12:22:16 PM	Device type: 491852 is applied to root node: ns=2;s=HelloWorld/Dynamic7 of server: 592201	MyServer
Mar 4, 2025, 12:22:15 PM	Device type: 491852 is applied to root node: ns=2;s=HelloWorld/Dynamic3 of server: 592201	MyServer

powered by CUMULOCITY

Device type has been un-applied

- Event type - c8y_ua_DeviceTypeUnapplied
- Event text -
 - If the device type has been un-applied from all nodes on the server: *Device type: {device type ID} is un-applied from all nodes of server: {server ID}*
 - If the device type has been un-applied from a specific node on the server: *Device type: {device type ID} is un-applied from root node: {root node ID} of server: {server ID}*
 - If all device types have been un-applied for the server: *All device types are un-applied for server: {server ID}*
- Event source - The server managed object

DEVICE MANAGEMENT

cucumberOpcuaGateway

Devices > All devices > cucumberOpcuaGateway > Events

Date from Date to Event type Apply Realtime Reload Add location

Home Devices Overviews Groups Device types Management

Info Child devices Measurements Alarms Control Firmware Availability Events OPC UA server Identity

Date	Event	Source
Mar 4, 2025, 12:22:40 PM	Device type: 491852 is un-applied from root node: ns=2;s=HelloWorld/Dynamic4 of server: 592201	MyServer
Mar 4, 2025, 12:22:40 PM	Device type: 491852 is un-applied from root node: ns=2;s=HelloWorld/ScalarTypes of server: 592201	MyServer
Mar 4, 2025, 12:22:40 PM	Device type: 491852 is un-applied from root node: ns=2;s=HelloWorld/Dynamic6 of server: 592201	MyServer
Mar 4, 2025, 12:22:40 PM	Device type: 491852 is un-applied from root node: ns=2;s=HelloWorld/Dynamic7 of server: 592201	MyServer
Mar 4, 2025, 12:22:40 PM	Device type: 491852 is un-applied from root node: ns=2;s=HelloWorld/Dynamic3 of server: 592201	MyServer
Mar 4, 2025, 12:22:40 PM	Device type: 491852 is un-applied from root node: ns=2;s=HelloWorld/Dynamic8 of server: 592201	MyServer
Mar 4, 2025, 12:22:40 PM	Device type: 491852 is un-applied from root node: ns=2;s=HelloWorld/Dynamic9 of server: 592201	MyServer
Mar 4, 2025, 12:22:40 PM	Device type: 491852 is un-applied from root node: ns=2;s=HelloWorld/Dynamic2 of server: 592201	MyServer

powered by CUMULOCITY

Custom action retry mechanism on external server failure

If a custom action fails, a retry mechanism will be processed. This is configured in the application YAML file, and the queues will be stored in the event repository.

Queues are collections of failed custom actions, including the complete HTTP request of this custom action. Each entry of the queue is one failed custom action.

A background scheduler task will retry each queue up to the number of *maxRetries*. If *maxRetries* is reached the queue will be stored as a permanently failed queue in the event repository.

All elements of the queue are retried after the retry delay has passed. In effect, the count of the elements in the queue is decreases with each successful retried custom action.

This mechanism can be configured in the configuration section `gateway.mappingExecution.http.failureHandling` (see also [Additional customization](#)) using the following properties:

Setting	Default	Value format	Example	Description
enabled	true	boolean (true/false)	true	Activate or deactivate the fail over for custom actions
maxRetries	5	number	5	Number of retries for failed queues. If the maximum is reached the queue is saved as permanently failed and never tried again. Default is 5.
noRetryHttpCodes	empty	Comma - separated list of HTTP response codes	400,500	If retries are enabled (<code>failureHandling.enabled=true</code>), this setting allows retries to be skipped for certain HTTP response codes. In the given example requests that have received a <code>400 Bad Request</code> or a <code>500 Internal server error</code> response will not be retried.
retryDelay	120	number	120	Minimum delay in seconds between two retries of the same request.

DATA REPORTING

There are three data reporting mechanisms which can be applied to read all mapped browse paths:

- **None** - The gateway will not read values automatically. The mappings will be applied only when manual read operations are performed on mapped nodes.
- **Cyclic Read** - The gateway reads values from mapped nodes at specified interval rates in milliseconds. The minimum allowed rate is 50 milliseconds.
- **Subscription** - The gateway retrieves values by using OPC UA's own subscription mechanism.

Possible parameters:

- **Sampling interval** (required) - Defines a time interval individually for each mapped node. This is the rate at which the server checks the data source for changes.
- **Queue size** - The size of the queue where it holds the samples before reporting. If you wish to record samples at a faster rate than reporting interval, you will also need to reserve a longer queue size, to be able to keep all the samples in the server. The reporting interval is defined for the gateway and the value is configurable with the YAML file (see gateway.subscription.reportingRate).
- **Discard** - Select whether to discard the **oldest** or the **newest** item if the samples are exceeding the queue size.
- **Data change trigger**:
 - **Status** - Triggers notification if node's status has changed.
 - **Status/Value** - Triggers notification if node's status or value has changed.
 - **Status/Value/Timestamp** - Triggers notification if node's status, value or timestamp has changed.
- **Deadband filter** - Deadband filter makes notified data values to be filtered.
 - **None** - No filter will be applied. This option is selected by default.
 - **Absolute** - Contains the absolute change in a data value which causes the generation of a notification. This parameter applies only to variables with any number data type.
 - **Percent** - The value is defined as the percentage of the EU range. It applies only to analog items with a valid EU range property. This range is multiplied with the deadband value and is then compared to the actual value change in order to determine the need for a data change notification.

! IMPORTANT

Very low interval rates (for example 50 ms) for cyclic read and subscription types will result in huge amounts of data being created.

APPLYING CONSTRAINTS

Specifying auto-apply constraints allows you to limit the scope in which the device protocols are applied, for example by specifying a set of possible servers or node IDs. If no constraints are set, device protocols are applied at any fitting location on the OPC UA server.

The following constraints can be applied:

- **Limit device protocol to a set of servers** - Limit the device protocols to a particular set of servers. This is useful if you want to

have 1 device type for each OPC UA server. Simply click on the dropdown menu and select the desired servers.

- **Limit device protocol scope in the address space** - Limit the scope to servers which have the entered path in their address space.
- **Limit device protocol to servers with a certain fragment** - The device protocol will only be available to the servers which have the entered fragment.
- **Limit device protocol to specific root nodes ID** - A list of “root” node IDs (from which your browsePath is defined) to which the device protocol should be applied. For example, if there is only one server and the device protocol is applied to two node IDs, two child devices of the server will be created. Note that if the device protocol variables do not exist in the root nodes, the device protocol will not be applied to the root node server.

MyDeviceType

Device types > Device protocols > MyDeviceType

ID: 491852
Date created: Mar 4, 2025, 12:22:13 PM
Last update: Mar 4, 2025, 12:22:13 PM
Fieldbus version: 4

MyDeviceType *e.g. My protocol description*

☒ Active When active, the gateway scans the address space of all servers and automatically applies this device protocol to all nodes matching the criteria.

Data reporting

☐ None ☒ Cyclic read ☐ Subscription

Reading interval: 1000 msec

Auto apply constraints

Specifying auto-apply constraints allows you to limit the scope where the device protocols are applied, for example by specifying a set of possible servers or node IDs. If no constraints are set, device protocols are applied at any fitting location on the OPC UA server.

☒ Limit device protocol to a set of servers
Select server IDs from list

☒ Limit device protocol scope in the address space
e.g. /objects/devices/.*

☒ Limit device protocol to servers with a certain fragment
e.g. c8y_SomeServerMarker

☐ Limit device protocol to specific root nodes ID

Save

powered by CUMULOCITY

BAD STATUSCODE HANDLING

The gateway creates an alarm with status WARNING for the corresponding OPC UA server managed object when the received DataValue of a node has an abnormal status code (such as BAD or UNCERTAIN). The alarm text in the Cumulocity platform will provide the necessary information such as the node ID, and the StatusCode information (code, value, and description).

The type of the alarm is constructed as `c8y_ua_DataValueAbnormal_<nodeId>_<StatusCode>_<value>`. This alarm due to the abnormal status code for a node is created once, and the counter is not incremented in the next read operations. The alarm will be cleared by the OPC UA device gateway when the status code is back to normal in the following read operations. The full list of abnormal alarms are stored in the server managed object under the `c8y_ua_DataValueAbnormalAlarms` fragment.

REST APIS

While the Cumulocity user interface for OPC UA provides an easy and visual way to configure and build your OPC UA solution, the OPC UA management microservice gives you the possibility to do it via RESTful web service.

The full API definitions can be found at `/service/opcua-mgmt-service/swagger-ui.html`.

OPC UA SERVER RESOURCES

Connect a new OPC UA server to the gateway

Endpoint

POST /service/opcua-mgmt-service/gateways/{gatewayId}/servers

Description

Connect a new OPC UA server to the gateway or update the existing server with a new configuration.

Payload

```
{
  "name": "My Server",
  "config": {
    "securityMode": "NONE",
    "serverUrl": "opc.tcp://127.0.0.1:53530/OPCUA/SimulationServer",
    "autoScanAddressSpace": true
  }
}
```

Payload data structure explained:

Field	Type	Mandatory	Description
id	string	yes/no	String. Id of the OPC UA server in case of updating an existing server. When connecting a new server, this must not be provided.
name	string	yes	Server managed object name.
requiredInterval	integer	no	How frequently the server is expected to send data to the Cumulocity platform.
config	<i>ServerConnectionConfig</i>	yes	Connection configuration to the OPC UA server.

Data structure for ServerConnectionConfig:

Field	Type	Mandatory	Description
securityMode	string	yes	String enum, mandatory. Security mode for connection to OPC UA server. Possible values: <code>NONE</code> , <code>BASIC128RSA15_SIGN</code> , <code>BASIC128RSA15_SIGN_ENCRYPT</code> , <code>BASIC256_SIGN</code> , <code>BASIC256_SIGN_ENCRYPT</code> , <code>BASIC256SHA256_SIGN</code> , <code>BASIC256SHA256_SIGN_ENCRYPT</code> .
serverUrl	string	yes	String, mandatory. OPC UA server URL.
autoScanAddressSpace	boolean	no	boolean. Whether the gateway should scan the address space automatically the first time it connects. Default is true.
rescanCron	string	no	String. Regular expression that defines how the address space rescanning should be scheduled. If this is not set, no auto-rescan will be triggered.
autoReconnect	boolean	no	Boolean. Whether the gateway should try to reconnect to the OPC UA server once the connection drop is detected. Default is true.

Field	Type	Mandatory	Description
timeout	integer	no	Integer. Define the default communication timeout that is used for each synchronous service call. This value is set to each service request and the OPC UA gateway will wait for the response message for that long.
statusCheckInterval	integer	no	Integer. Define the status check interval, that is, how often the server status is read. Default is 3 (seconds).
maxResponseMessageSize	long	no	Integer. Define the maximum size, in bytes, for the body of any response message from the server. Default is 50.000.000 (50 MB). To make it unlimited, set this to 0.
targetConnectionState	string	no	String enum. Possible values: <code>enabled/disabled</code> . Whether the connection to the target OPC UA server is enabled.
userIdentityMode	string	no	User identity. Possible values: <code>Anonymous</code> , <code>UserName</code> , <code>Certificate</code> , <code>IssuedToken</code> . Default is <code>Anonymous</code> .
userName	string	yes/no	Authentication username when user identity mode is <code>UserName</code> .
userPassword	string	no	Authentication password when user identity mode is <code>UserName</code> . Set the value in order to change the authentication password. If it is not set, the previously stored authentication password is preserved.
keystoreBinaryId	string	yes/no	If the user identity mode is <code>Certificate</code> , this is the binary object ID of the uploaded keystore.
keystorePass	string	no	If the user identity mode is <code>Certificate</code> , this is the password of the uploaded keystore.
certificatePass	string	no	If the user identity mode is <code>Certificate</code> , this is the password of the private key embedded in the keystore.
cyclicReadBulkSize	integer	no	For cyclic reads, this defines how many nodes can be read at once from the OPC UA server. This is applicable only for nodes resulting from the application with the same device type, matching root node and sharing the same reading parameters (rate and max age). Default is 1000 as defined in the application YAML file.
alarmSeverityMappings	map<string, string>	no	<p>Alarm severity mappings from the OPC UA event severity to the Cumulocity alarm severity. This is applicable only for UAAlarmCreation. The key of this map is the lower bound value of the OPC UA event severity in the range. The value of this map is the expected severity of the alarm being created. For example, to map the OPC UA severity of the range 200-400 to a <i>MINOR</i> Cumulocity alarm, put this entry to the map: <code>"200": "MINOR"</code> .</p> <p>If this is given, it will override the alarm severity mappings that are specified in the configuration YAML file.</p> <p>Note that, if the <i>severity</i> field for alarm mapping is provided, this <i>alarmSeverityMappings</i> will have no effect.</p> <p>Example: <code>"201": "WARNING",</code> <code>"401": "MINOR",</code> <code>"601": "MAJOR",</code> <code>"801": "CRITICAL"</code></p> <p>Additionally, the <i>Severity</i> attribute should be added in the subscribed attributes (uaEventMappings -> attributes) in the device type in order to get the severity value of the OPC UA event.</p>

Field	Type	Mandatory	Description
alarmStatusMappings	map<string, string>	no	<p>The status of an alarm in Cumulocity is defined by multiple conditions on OPC UA servers. For example, if the value of <code>AcknowledgedState</code> node is "Aked" and <code>ConfirmedState</code> is "Confirmed", then the status of the alarm in Cumulocity is expected as ACKNOWLEDGED. They might vary with different servers as well. This field enables the user to configure the desired conditions (based on the information retrieved from the event type nodes of the OPC UA server) while creating alarms via UA event mappings (this is not applicable for OPC UA data value alarm creation). The example below shows that the keys of the map are the user-defined expressions and the value represents their corresponding desired status of the alarm. The variables that can be used in the expressions are the selected attributes provided in the subscription definition of the device type. It can be written down either by using the relevant node names (for example <code>EnabledState.text == 'Enabled'</code>), or the qualified browse name with namespace index (for example <code>['0:EnabledState'].text == 'Enabled'</code>). If the variables are not provided in the subscribed attributes (uaEventMappings -> attributes), they are considered null. If the alarm status is explicitly provided in the alarm mapping (uaEventMappings -> alarmCreation) of the device type, these alarm status mappings have no effect. The Spring Expression Language(SpEL) has been used to parse these conditions, but only boolean expressions are allowed. See the alarmStatusMappings example below the table.</p>

INFO

There are three alarm statuses in Cumulocity, namely ACTIVE, ACKNOWLEDGED, and CLEARED. If the user-defined conditions overlap and as a result more than one alarm status is realized during the alarm creation, then the status is selected based on priority. ACTIVE has the highest priority, followed by ACKNOWLEDGED and then CLEARED status with the least priority. If the expression could not be evaluated then the gateway logs a warning and the alarm status is assumed as ACTIVE. The alarm status is also assumed as ACTIVE, if the default status is not specified, and the parameters do not match any other defined condition.

subscribeModelChangeEvent	boolean	no	<p>The subscription to model change event can be enabled/disabled using this property. Default value is "false" (disabled), which means any change in the address space nodes of the OPC UA server in runtime will not automatically be updated in the address space of Cumulocity. This property must be explicitly set to "true" to detect and persist the address space changes on runtime.</p>
---------------------------	---------	----	--

Field	Type	Mandatory	Description
validateDiscoveredEndpoints	boolean	no	<p>The protocol stack of the OPC UA gateway validates whether the endpoint it is connecting to is present in the list of endpoints returned by the OPC UA server. By default, this validation is enabled (true). However, a setting in the server configuration can override the global gateway configuration setting, which can be configured using <code>gateway.connectivity.validateDiscoveredEndpoints</code>.</p> <p>We strongly recommend you to keep this validation enabled unless there are compelling reasons to disable it. Disabling the endpoint validation should only be done when absolutely necessary.</p>

alarmStatusMappings example

```
{
  "alarmStatusMappings": {
    "[0:ActiveState].text == 'Active' and [0:AckedState].text != 'Acknowledged'": "ACTIVE",
    "[0:ActiveState].text == 'Active' and [0:AckedState].text == 'Acknowledged'": "ACKNOWLEDGED",
    "[0:ActiveState].text == 'Inactive'": "CLEARED",
    "default": "ACTIVE"
  }
}
```

Alarms status changed by OPC UA server

If events operated on the OPC UA server change their status, these changes can be reflected as internal alarms.

To catch these events and convert them into internal alarms, a UA event mapping with the alarmCreation definition in device protocol and alarmStatusMappings in server configuration are required.

For better performance an in-memory map is used to store the alarm type and the internal representation. These values are also stored on the filesystem and survive a possible crash or restart of the gateway. When the alarm is cleared then its entry is removed from the in-memory map.

Get all servers of a gateway device

Method

GET /service/opcua-mgmt-service/gateways/{gatewayId}/servers

Parameters

Field	Field type	Description
gatewayId	Path variable	Managed object ID of the gateway that should connect to the OPC UA server.

Example response

```
[
  {
    "id": "25197",
    "gatewayId": "800",
    "name": "My OPC UA server",
    "requiredInterval": 1,
    "config": {
      "securityMode": "NONE",
      "serverUrl": "opc.tcp://127.0.0.1:12686/CumulocityOPCUAServer",
      "userIdentityMode": "Anonymous",
      "timeout": 30,
      "autoReconnect": true,
      "statusCheckInterval": 40,
      "targetConnectionState": "enabled"
    },
    "namespaceTable": [
      "http://opcfoundation.org/UA",
      "urn:cumulocity:opcua:test:server:ee8ff646-cc83-4a1f-ad29-97356c496ef0",
      "urn:cumulocity:opcua:test:server"
    ],
    "c8y_Availability": {
      "lastMessage": "2020-08-27T12:43:22.585+0000",
      "status": "UNAVAILABLE"
    },
    "c8y_Connection": {
      "status": "DISCONNECTED"
    },
    "c8y_RequiredAvailability": {
      "responseInterval": 1
    }
  }
]
```

Delete and disconnect an OPC UA server

Endpoint

DELETE /service/opcua-mgmt-service/servers/{serverId}

Description

Delete the OPC UA server managed object. Once the DELETE request is received by the OPC UA management service, the specified server along with all its address space nodes created in the Cumulocity platform will be deleted. The service will retain all the child devices of the server, and their corresponding data, which were created by the device protocols.

Parameters

Field	Field type	Description
serverId	Path variable	Managed object ID of the OPC UA server.

Response

200 OK

ADDRESS SPACE RESOURCES

Get an address space node by ID

Endpoint

GET /service/opcua-mgmt-service/servers/{serverId}/address-spaces/get

Description

Get a node in the server address space specified by the given node ID. The node ID must be URL encoded.

Parameters

Parameter	Parameter Type	Description
serverId	Path variable	Integer, mandatory. OPC UA server managed object ID.
nodeId	Query param	Mandatory. Node ID of the node to get.
withUri	Query param	Boolean, default is false. Whether the result should use address space URI instead of index.

Example

Endpoint: `GET /service/opcua-mgmt-service/servers/10/address-spaces/get?nodeId=i%3D84`

```
{
  "nodeId": "i=84",
  "nodeClass": 1,
  "nodeClassName": "Object",
  "browseName": "0:Root",
  "browsePath": null,
  "displayName": "Root",
  "description": "The root of the server address space.",
  "references": [
    {
      "referenceId": "i=35",
      "targetId": "i=87",
      "referenceLabel": "Organizes",
      "targetLabel": "Views",
      "targetBrowseName": "Views",
      "inverse": false,
      "hierarchical": true
    },
    {
      "referenceId": "i=40",
      "targetId": "i=61",
      "referenceLabel": "HasTypeDefinition",
      "targetLabel": "FolderType",
      "targetBrowseName": "FolderType",
      "inverse": false,
      "hierarchical": false
    }
  ],
  "attributes": {
    "eventNotifier": 0
  },
  "absolutePaths": [
    [
      "0:Root"
    ]
  ],
  "ancestorNodeIds": [
    []
  ]
}
```

Get children of a given node

Endpoint

`GET /service/opcua-mgmt-service/servers/{serverId}/address-spaces/children`

Description

Get all child nodes of the given node specified by the node ID in the server address space. The node ID must be properly URL encoded.

Parameters

Parameter	Parameter Type	Description
serverId	Path variable	Integer, mandatory. OPC UA server managed object ID.
nodeId	Query param	Mandatory. Node ID of the node to get.
withUri	Query param	Boolean, default is false. Whether the result should use address space URI instead of index.

Example

Endpoint: `GET /service/opcua-mgmt-service/servers/10/address-spaces/children?nodeId=i%3D84`

```
[
  {
    "nodeId": "i=86",
    "nodeClass": 1,
    "nodeClassName": "Object",
    "browseName": "0:Types",
    "browsePath": null,
    "displayName": "Types",
    "description": "The browse entry point when looking for types in the server address space.",
    "references": [
      {
        "referenceId": "i=40",
        "targetId": "i=61",
        "referenceLabel": "HasTypeDefinition",
        "targetLabel": "FolderType",
        "targetBrowseName": "FolderType",
        "inverse": false,
        "hierarchical": false
      },
      {
        "referenceId": "i=35",
        "targetId": "i=86",
        "referenceLabel": "OrganizedBy",
        "targetLabel": "Types",
        "targetBrowseName": "Types",
        "inverse": true,
        "hierarchical": true
      }
    ],
    "attributes": {
      "eventNotifier": 0
    },
    "absolutePaths": [
      [
        "0:Root",
        "0:Types"
      ]
    ],
    "ancestorNodeIds": [
      [
        "i=84"
      ]
    ]
  }
]
```

Browse a node

Endpoint

GET /service/opcua-mgmt-service/servers/{serverId}/address-spaces/browse

Description

Browse a node from a base node following the given browse path. This basically searches for a node with relative browse path to the other node.

Parameters

Parameter	Parameter Type	Description
serverId	Path variable	Integer, mandatory. OPC UA server managed object ID.
nodeId	Query param	Node ID of the node to browse from. Default is root node (i=84).

Parameter	Parameter Type	Description
browsePath	Query param	Mandatory. Browse path to browse from the give node. Browse path can be a single param or an array to represent a path from the given node to the target node. To specify the browsePath as an array, repeat the browsePath query. Example: <code>.../browse?browsePath=L1&browsePath=L2</code> .
withUri	Query param	Boolean, default is false. Whether the result should use address space URI instead of index.

Example

Endpoint: `GET /service/opcua-mgmt-service/servers/10/address-spaces/browse?nodeId=i%3D84&browsePath=Objects`

```
[
  {
    "nodeId": "i=85",
    "nodeClass": 1,
    "nodeClassName": "Object",
    "browseName": "0:Objects",
    "browsePath": null,
    "displayName": "Objects",
    "description": "The browse entry point when looking for objects in the server address space.",
    "references": [
      {
        "referenceId": "i=35",
        "targetId": "i=2253",
        "referenceLabel": "Organizes",
        "targetLabel": "Server",
        "targetBrowseName": "Server",
        "inverse": false,
        "hierarchical": true
      },
      {
        "referenceId": "i=35",
        "targetId": "ns=2;s=Cumulocity",
        "referenceLabel": "Organizes",
        "targetLabel": "Cumulocity",
        "targetBrowseName": "2:Cumulocity",
        "inverse": false,
        "hierarchical": true
      }
    ],
    "attributes": {
      "eventNotifier": 0
    },
    "absolutePaths": [
      [
        "0:Root",
        "0:Objects"
      ]
    ],
    "ancestorNodeIds": [
      [
        "i=84"
      ]
    ]
  }
]
```

DEVICE TYPE RESOURCES

These resources provide the APIs for manipulating device types.

Creating a new device type

Endpoint

POST /service/opcua-mgmt-service/device-types

Sample payloads

- Measurement mappings using subscription

```
{
  "name": "My device type",
  "enabled": true,
  "mappings": [
    {
      "browsePath": [
        "2:Dynamic",
        "2:Double"
      ],
      "measurementCreation": {
        "unit": "T",
        "type": "MyMeasurementType",
        "fragmentName": "MyMeasurement",
        "series": "MySeries"
      }
    }
  ],
  "subscriptionType": {
    "type": "Subscription",
    "subscriptionParameters": {
      "samplingRate": 5000
    }
  }
}
```

- Event mappings using cyclic read

```
{
  "name": "My device type",
  "enabled": true,
  "mappings": [
    {
      "browsePath": [
        "2:Dynamic",
        "2:Integer"
      ],
      "eventCreation": {
        "type": "MyEventType",
        "text": "My event with value ${value}"
      }
    }
  ],
  "subscriptionType": {
    "type": "Cyclicread",
    "rate": 5000
  }
}
```

- Alarm mappings using subscription

```
{
  "name": "My device type",
  "enabled": true,
  "mappings": [
    {
      "browsePath": [
        "2:Dynamic",
        "2:Boolean"
      ],
      "alarmCreation": {
        "type": "MyAlarm",
        "severity": "MAJOR",
        "text": "Heads up, the level is high!"
      }
    }
  ],
  "subscriptionType": {
    "type": "Subscription",
    "subscriptionParameters": {
      "samplingRate": 5000
    }
  }
}
```

- UA events mappings into alarm and event

```
{
  "name": "My device type",
  "enabled": true,
  "uaEventMappings": [
    {
      "browsePath": [
        "Server"
      ],
      "eventTypeId": "i=2041",
      "attributes": [
        "Message",
        "Severity"
      ],
      "alarmCreation": {
        "text": "ALARM! message: ${0}",
        "severity": "MAJOR",
        "status": "ACTIVE",
        "type": "c8y_myEvent_alarm_${1}"
      }
    }
  ]
}
```

Full payload data structure explained:

Field	Type	Mandatory	Description
name	string	yes	Device type name.
enabled	boolean	no	Whether the device type is enabled and should be applied to the connected servers or not. Default is false.
description	string	no	
mappings	array<Mapping>	no	Define the mappings from OPC UA data into Cumulocity measurements, events and alarms.

Field	Type	Mandatory	Description
uaMappings	<i>array<UAMapping></i>	no	Define the mappings from OPC UA alarms and events into Cumulocity alarms and events.
referencedNamespaceTable	array	no	Reference namespace table if known. This is then used to convert the browse paths with namespace index into namespace URL. This is to make sure that the mappings are still the same even when the namespace index gets changed.
subscriptionType	<i>SubscriptionType</i>	no	Define the mechanism how to collect data from the OPC UA servers. There are two mechanisms that can be used: Cyclic read and subscription. This is not mandatory however if the device type is enabled and no subscription type is specified, the device will not be applied to any node.
processingMode	string	no	Define the Cumulocity processing mode for incoming data. Refer to HTTP usage > Process mode in the Cumulocity OpenAPI Specification for more information. Possible values: PERSISTENT, TRANSIENT, QUIESCENT, CEP. Default is PERSISTENT. Note that for the alarm mappings, only the PERSISTENT mode is supported regardless what is being given here.
overriddenSubscriptions	<i>OverriddenSubscription</i>	no	While the subscriptionType defines how data can be collected from the OPC UA server, this option allows you to override the mechanism for particular browse paths. For example you can have a subscription applied globally with a sampling rate of 1000ms but you can apply sampling rate of 500ms for particular browse paths.
applyConstraints	<i>ApplyConstraint</i>	no	Limit the places in the address space where the device type should be applied.

Data structure for Mapping

Field	Type	Mandatory	Description
name	string	no	
browsePath	array	yes	The browse path. The path can be the exact browse path or a regular expression of the browse path. Each browse path in the list of mappings must be unique. Duplication is not allowed by the API. If it is a regular expression, it must be wrapped inside <code>*regex(...)*</code> . For example: <code>`regex(2:Objects)`</code> or <code>`regex(urn:test.namespace:Objects\\d)`</code> . Note that the namespace index and the namespace URI are not part of the regular expression itself but they will be quoted as literal strings. When using a regular expression, keep in mind that it might be matching many nodes in the address space and resulting in unexpected incoming data. Our recommendation is to use it with great care and together with other exact browse paths in the same mapping if possible. For example, <code>`["2:Objects", "regex(2:MyDevice\\d)", "..."]`</code>
measurementCreation	<i>MeasurementCreation</i>	no	Mappings for measurement.
eventCreation	<i>EventCreation</i>	no	Mappings for event.

Field	Type	Mandatory	Description
alarmCreation	<i>AlarmCreation</i>	no	Mappings for alarm. If the value of the mapped resource is "true" (in case of boolean), or a positive number (in case of integer/double), then the alarms are created in ACTIVE state. The alarm de-duplication prevents the creation of multiple alarms with the same source and type, thereby only incrementing the count of the existing alarm. The alarms will be CLEARED as soon as the value is changed to "false", or a number that is less than or equals to 0.
customAction	<i>HttpPostAction</i>	no	Mappings for custom action. Only HTTP POST is supported so far.

Data structure for *UAMapping*

Field	Type	Mandatory	Description
browsePath	array	yes	The browse path.
eventTypeId	string	yes	The event type ID.
attributes	array	yes	Selectable event attributes. The nodeId of the event source is added by default as the last selected attribute by the OPC UA device gateway. If <i>alarmSeverityMappings</i> are defined, also the <i>Severity</i> attribute must be added to the attributes. If <i>alarmStatusMappings</i> are defined, also the variables used in the expression must be added to the attributes.
eventCreation	<i>UAEventCreation</i>	no	Mappings for event.
alarmCreation	<i>UAAAlarmCreation</i>	no	Mappings for alarm.

Data structure for *SubscriptionType*

Field	Type	Mandatory	Description
type	string	yes	Subscription type. Possible values: Subscription, CyclicRead, None.
subscriptionParameters	<i>SubscriptionParameters</i>	yes/no	In case the subscription type is <i>Subscription</i> , this is required. This defines the OPC UA subscription configuration, such as sampling rate, queue size, and more.
cyclicReadParameters	<i>CyclicReadParameter</i>	yes/no	In case the subscription type is <i>CyclicRead</i> , this is required. This defines the cyclic read configuration, such as rate.

Data structure for *OverriddenSubscription*

Field	Type	Mandatory	Description
browsePath	array	yes	The browse path.
subscriptionType	<i>SubscriptionType</i>	yes	The custom subscription type that overrides the global one.

Data structure for *ApplyConstraints*

Field	Type	Mandatory	Description
matchesServerIds	array	no	Limit the servers by server managed object ID where the device type should be applied.
serverObjectHasFragment	string	no	Limit the servers by their custom fragment where the device type should be applied.
matchesNodeIds	array	no	Limit the nodes in the server address space where the device type should be applied.
browsePathMatchesRegex	string	no	Regular expression of the browse paths where the device type should be applied.
serverHasNodeWithValues	<i>ServerNodeValues</i>	no	Limit the servers which have particular nodes with given values.

Data structure for *MeasurementCreation*

Field	Type	Mandatory	Description
type	string	yes	Measurement type.
series	string	no	Measurement series. If this is omitted, it will be automatically generated by the gateway.
unit	string	yes	Measurement unit.
fragmentName	string	no	Measurement fragment name. If this is omitted, it will be automatically generated by the gateway.
staticFragments	array	no	Static fragments that should be populated to the measurement.
overriddenProcessingMode	string	no	Custom processing mode applied to the measurement to be created. Possible values: PERSISTENT, TRANSIENT, QUIESCENT, CEP. Default: PERSISTENT.

Data structure for *EventCreation*

Field	Type	Mandatory	Description
type	string	yes	Event type.
text	string	yes	Event text. This event text can be parameterized by the value of the subscribed node by using the placeholder: <code>\${value}</code> .
staticFragments	array	no	Static fragments that should be populated to the measurement.
overriddenProcessingMode	string	no	Custom processing mode applied to the event to be created. Possible values: PERSISTENT, TRANSIENT, QUIESCENT, CEP. Default: PERSISTENT.

Data structure for *AlarmCreation*

Field	Type	Mandatory	Description
type	string	yes	Alarm type.

Field	Type	Mandatory	Description
text	string	yes	Alarm text.
severity	string	yes	Alarm severity. Possible values: WARNING, MINOR, MAJOR, CRITICAL.
staticFragments	array	no	Static fragments that should be populated to the alarm.
overriddenProcessingMode	string	no	Custom processing mode applied to the alarm to be created. Possible values: PERSISTENT.

Data structure for *HttpPostAction*

Field	Type	Mandatory	Description
endpoint	string	yes	Endpoint of the HTTP POST request.
headers	map<string, string>	no	HTTP headers of the HTTP request.
bodyTemplate	string	yes	Template of the request body. This can be parameterized by the following placeholders: <code>\${value}</code> : Data value of the OPC UA node. <code>\${serverId}</code> : OPC UA server managed object ID. <code>\${nodeId}</code> : ID of the node where the data is coming from. <code>\${deviceId}</code> : Managed object ID of the source manage object.
retryEnabled	boolean	no	Whether a failed HTTP POST should be retried or not. This overrides the configuration in the gateway. If this is not provided, the configuration in the gateway will be taken.
noRetryHttpCodes	array<integer>	no	Array of HTTP POST status exceptions by which the failed HTTP POST should not be retried if enabled. Example: [400, 500]. Note that, if this is null or missing, the exceptions will be taken from the gateway configuration. If this is provided, even with an empty array, the configuration in the gateway is disregarded.

Data structure for *UAEventCreation*

This has exactly the same fields as *EventCreation*, however the *text* and *type* field can be parameterized with different parameters.

Field	Type	Mandatory	Description
text	string	yes	Event text. This event text can be parameterized by the data value of selected attributes. Put <code>\${i}</code> to parameterize it by the data value of attribute at index <i>i</i> . The index starts from 0, so put <code>\${0}</code> to take the first attribute, <code>\${1}</code> to select second attribute, and so on.
type	string	yes	Event type. This event type can be parameterized by the data value of selected attributes. Put <code>\${i}</code> to parameterize it by the data value of attribute at index <i>i</i> . The index starts from 0, so put <code>\${0}</code> to take the first attribute, <code>\${1}</code> to select second attribute, and so on.

Data structure for *UAAAlarmCreation*

Field	Type	Mandatory	Description
text	string	yes	Alarm text. This alarm text can be parameterized by the data value of selected attributes. Put <code>\${i}</code> to parameterize it by the data value of attribute at index <code>i</code> . The index starts from 0, so put <code>\${0}</code> to take the first attribute, <code>\${1}</code> to select second attribute, and so on.
type	string	yes	Alarm type. This alarm type can be parameterized by the data value of selected attributes. Put <code>\${i}</code> to parameterize it by the data value of attribute at index <code>i</code> . The index starts from 0, so put <code>\${0}</code> to take the first attribute, <code>\${1}</code> to select second attribute, and so on.
severity	string	no	For UAAAlarmCreation, the severity is optional. If this is not provided, the severity of the alarm will be mapped using the severity mappings specified in the default gateway configuration YAML file or in the server configuration.
status	string	no	Alarm status. The possible values are ACTIVE, ACKNOWLEDGED, CLEARED. If this is given, the <i>alarmStatusMappings</i> setting in <i>ServerConnectionConfig</i> is ignored.

Data structure for *SubscriptionParameter*

Field	Type	Mandatory	Description
samplingRate	integer	yes	Subscription sampling rate in milliseconds. Minimum allowed value is 50.
deadbandType	string	no	Possible values: Percent, Absolute.
deadbandValue	double	no	If the <i>deadbandType</i> is <i>Percent</i> , this ranges from 0 to 100. If the <i>deadbandType</i> is <i>Absolute</i> , this can be any double value.
queueSize	integer	no	Subscription queue size.
dataChangeTrigger	string	no	Default value is StatusValue. Possible values: Status, StatusValue, StatusValueTimestamp.
discardOldest	boolean	no	Default is true. When this is true and the reported data is exceeding the queue size, the oldest elements in the queue will be discarded. If this is false, the newer elements will be discarded.
ranges	string	no	When the subscribed node is array type, you can provide the data range you want to get. For example: "0:3" to get elements from index 0 to 3 from the array.

Data structure for *CyclicParameter*

Field	Type	Mandatory	Description
rate	integer	yes	Cyclic read rate in milliseconds. Minimum allowed value is 50.

Data structure for *ServerNodeValues*

Field	Type	Mandatory	Description
matchAll	array	no	A collection of conditions and they must all be matched.

Field	Type	Mandatory	Description
matchOneOf	array	no	A collection of conditions and at least one of them must be matched.
Data structure for <i>MatchingNode</i>			
Field	Type	Mandatory	Description
nodeId	string	yes	The node ID to match against.
valueMatchesOneOf	array	no	A collection of possible values of the node, in string representation. If this is omitted, the gateway only checks for the existence of the node by given node ID.

Get all OPC UA device types

Endpoint

GET /service/opcua-mgmt-service/device-types

Payload

The endpoint returns a JSON array of all OPC UA device types.

Get a single device type

Endpoint

GET /service/opcua-mgmt-service/device-types/{deviceTypeId}

Payload

A JSON representation of the device type with the given ID if it exists. If not, an error message is returned.

Response codes

200 OK

404 Not found

Updating a device type

Endpoint

PUT /service/opcua-mgmt-service/device-types/{deviceTypeId}

Payload

The payload of updating a device type is exactly the same as the payload of creating it. Please note that partial update is not supported. All information must be provided in the update request and will completely override the existing device type.

Deleting a device type

Endpoint

DELETE /service/opcua-mgmt-service/device-types/{deviceTypeId}

Success response

204 No Content

OPERATIONS

Cumulocity operations is the interface that is used to tell the gateway what to do and how to do it. This section describes all operations that are currently supported by the gateway.

SCANNING THE ADDRESS SPACE

This operation triggers importing address space for a specific OPC-UA server. The server's ID is passed as a device ID. The gateway will scan the entire address space of the server and persist a twinned representation of the address space in the Cumulocity platform.

```
POST /devicecontrol/operations/

{
  "deviceId": "<server-device-Id>",
  "c8y_ua_command_ScanAddressSpace": {
    "skipSync": false
  },
  "description": "Import address space from root node"
}
```

The twinned address space information is persisted in the Cumulocity inventory. It is internally used to support address space browsing and to define device protocols. Hence this operation is always triggered if a new server is added to the platform.

Once the device gateway knows the address space, it uses it to handle different logics, for example applying device protocols to nodes. So if you already have the address space scanned once and stored in Cumulocity, you might want the device gateway to learn one more time about server's address space without synchronizing data into Cumulocity. To achieve that, provide `"skipSync": true`.

To do a partial address space scan, you can provide the `nodeIds` property which contains all node IDs to be scanned. The subaddress space starting from those nodes as well as the ancestor nodes are persisted both in the Cumulocity inventory, unless `skipSync` is set to `true`, and in the local address space file of the gateway.

```
POST /devicecontrol/operations/

{
  "deviceId": "<server-device-Id>",
  "c8y_ua_command_ScanAddressSpace": {
    "nodeIds": [
      "ns=3;i=1005",
      "ns=3;i=1001",
      "ns=3;i=1004"
    ],
    "skipSync": false
  },
  "description": "Import address space from node(s) ns=3;i=1005,ns=3;i=1001,ns=3;i=1004. Triggered by custom operation"
}
```

Available arguments for `c8y_ua_command_ScanAddressSpace` :

Field	Type	Mandatory	Description
nodeId	string	no	Scan of the address space starts from this node. If not provided, full scan is done starting from the root node.
skipSync	boolean	no	If set to true, the address space nodes will not be synchronized to Cumulocity Inventory API. Default is false.



INFO

We do not recommend you to directly work with the persisted address space data structures in the Cumulocity inventory, as these might change in the future. Use the endpoints of the management service to interact with the OPC UA address space.

READING THE VALUE OF A NODE/NODES

This operation reads the value attribute of specific node or list of nodes. The `deviceId` of the operation can be either the OPC UA Server device ID or the generated device ID (child device of OPC UA Server).

```
POST /devicecontrol/operations/

{
  "deviceId": "<server-or-child-device-Id>",
  "c8y_ua_command_ReadValue": {
    "nodes": ["NODE_ID"],
    "timestampsToReturn": "Neither"
  },
  "description": "read value"
}
```

Available arguments for `c8y_ua_command_ReadValue` :

Field	Type	Mandatory	Description
nodes	string array	yes	Array of IDs of the nodes to execute the operation
ranges	string	no	The index ranges of a subset of the multi-dimension array from the read attribute. The syntax is according to the OPC UA specification and will be transformed to NumericRange.
<div>NumericRange: <dimension> [',' <dimension>] <dimension>: <index> [':' <index>]</div> <p>Example values to define the range for a 1D array is "0:1", for a 2D array is "0:1,0:1"</p>			
maxAge	double	no	The maximum age used for the read. If the server does not have a value that is within the maximum age, it shall attempt to read a new value from the data source. If maxAge is set to 0, the server shall attempt to read a new value from the data source. Default is 0.
timestampsToReturn	string	no	Time stamps to return for the read attributes in the operation result. Available options are "Source", "Server", "Both", "Neither". Default is "Both".
expirationTime	dateTime	no	Expiration time to execute the operation. The operation is executed if it is before the given expiration time. Otherwise, the operation will fail. In this case, "Operation expired" is returned as failure reason.

The result of this operation will contain output in the following format:

```
{
  "results": {
    "ns=2;s=MyLevel": {
      "13": {
        "value": {
          "value": 77.0
        },
        "statusCode": 0,
        "sourcePicoSeconds": 0,
        "serverPicoSeconds": 0
      }
    }
  }
}
```

READING ALL ATTRIBUTES OF A NODE

This operation returns all attributes of specific node. The `deviceId` of the operation can be either the OPC UA Server device ID or the generated device ID (child device of OPC UA Server).

```
{
  "deviceId": "<server-or-child-device-Id>",
  "c8y_ua_command_ReadNodeAttributes": {
    "node": "ns=2;s=MyEnumObject"
  },
  "description": "Read node attributes"
}
```

Available arguments for `c8y_ua_command_ReadNodeAttributes` :

Field	Type	Mandatory	Description
node	string	yes	ID of the node to execute the operation
expirationTime	dateTime	no	Expiration time to execute the operation. The operation is executed if it is before the given expiration time. Otherwise, the operation will fail. In this case, "Operation expired" is returned as failure reason.

The result may differ depending on the node type.

```
{
  "Value": {
    "value": 1
  },
  "DataType": "ns=2;s=MyEnumType",
  "ValueRank": -1,
  "AccessLevel": 3,
  "UserAccessLevel": 3,
  "MinimumSamplingInterval": -1.0,
  "Historizing": false,
  "DisplayName": "MyEnumObject",
  "WriteMask": 0,
  "UserWriteMask": 0
}
```

READING AN ATTRIBUTE

This operation supports to read one or more attributes of one or more nodes. This includes support of the range parameter to read a single element or a range of elements when the attribute value is an array. The `deviceId` of the operation can be either the OPC UA Server device ID or the generated device ID (child device of OPC UA Server).

```
{
  "deviceId": "<server-or-child-device-id>",
  "c8y_ua_command_ReadAttribute": {
    "nodes": ["ns=3;s=FloatArray"],
    "attribute": "13"
  },
  "description": "Read attribute from ns=3;s=FloatArray"
}
```

Available arguments for `c8y_ua_command_ReadAttribute` :

Field	Type	Mandatory	Description
nodes	string array	yes	Array of IDs of the nodes to execute the operation
attribute	string	yes	The ID of the attribute according to the OPC UA specification
ranges	string	no	The index ranges of a subset of the multi-dimension array from the read attribute. The syntax is according to the OPC UA specification and will be transformed to NumericRange.
<div>NumericRange: <dimension> [',' <dimension>] <dimension>: <index> ['.' <index>]</div> <p>Example values to define the range for a 1D array is "0:1", for a 2D array is "0:1,0:1"</p>			
maxAge	double	no	The maximum age used for the read. If the server does not have a value that is within the maximum age, it shall attempt to read a new value from the data source. If maxAge is set to 0, the server shall attempt to read a new value from the data source. Default is 0.
timestampsToReturn	string	no	Time stamps to return for the read attributes in the operation result. Available options are "Source", "Server", "Both", "Neither". Default is "Both".
expirationTime	dateTime	no	Expiration time to execute the operation. The operation is executed if it is before the given expiration time. Otherwise, the operation will fail. In this case, "Operation expired" is returned as failure reason.

The result may differ depending on the node type.

```
{
  "results": {
    "ns=3;s=FloatArray": {
      "13": {
        "value": {
          "value": [1.0, 2.0, 3.0, 4.0, 5.0]
        },
        "statusCode": 0,
        "sourceTimestamp": 1566572540173,
        "sourcePicoseconds": 0,
        "serverTimestamp": 1566573849897,
        "serverPicoseconds": 0
      }
    }
  }
}
```

Example operation with ranges fragment:

```
{
  "description": "Read attribute from ns=3;s=FloatArray",
  "deviceId": "<server-device-Id>",
  "c8y_ua_command_ReadAttribute": {
    "nodes": ["ns=3;s=FloatArray"],
    "attribute": "13",
    "ranges": "0:1"
  }
}
```

The result may differ depending on the node type.

```
{
  "results": {
    "ns=3;s=FloatArray": {
      "13": {
        "value": {
          "value": [1.0, 2.0]
        },
        "statusCode": 0,
        "sourceTimestamp": 1566572540173,
        "sourcePicoseconds": 0,
        "serverTimestamp": 1566574513935,
        "serverPicoseconds": 0
      }
    }
  }
}
```

READ COMPLEX

This operation reads many attributes from many nodes at single call. The `deviceId` of the operation can be either the OPC UA Server device ID or the generated device ID (child device of OPC UA Server).

```
{
  "deviceId": "<server-or-child-device-Id>",
  "c8y_ua_command_ReadComplex": {
    "nodeAttrs": {
      "ns=2;s=MyEnumObject": {
        "13": "",
        "11": ""
      }
    }
  },
  "description": "Read complex"
}
```

Available arguments for `c8y_ua_command_ReadComplex` :

Field	Type	Mandatory	Description
-------	------	-----------	-------------

Field	Type	Mandatory	Description
nodeAttrs	map<string, map<string, string>>	yes	Map with ID of the node and inner map with ID of the attribute and the index range. The index ranges defines a subset of the multi-dimension array from the read attribute. The syntax is according to the OPC UA specification and will be transformed to NumericRange. <div> NumericRange: <dimension> [',' <dimension>] <dimension>: <index> ['<index>] </div> Example values to define the range for a 1D array is "0:1", for a 2D array is "0:1,0:1". Empty string ("") can be given to not define any range.
maxAge	double	no	The maximum age used for the read. If the server does not have a value that is within the maximum age, it shall attempt to read a new value from the data source. If maxAge is set to 0, the server shall attempt to read a new value from the data source. Default is 0.
timestampsToReturn	string	no	Time stamps to return for the read attributes in the operation result. Available options are "Source", "Server", "Both", "Neither". Default is "Both".
expirationTime	dateTime	no	Expiration time to execute the operation. The operation is executed if it is before the given expiration time. Otherwise, the operation will fail. In this case, "Operation expired" is returned as failure reason.

HISTORIC READ

This operation reads history values and applies the mappings except of alarm mappings. The `deviceId` of the operation can be either the OPC UA Server device ID or the generated device ID (child device of OPC UA Server).

```
{
  "deviceId": "<server-or-child-device-id>",
  "c8y_ua_command_HistoricReadOperation": {
    "nodeId": "ns=2;s=MyLevel",
    "processMappings": true,
    "dateFrom": "2019-06-13T10:43:00+02:00",
    "dateTo": "2019-06-13T10:52:00+02:00",
    "tagType": "TAG",
    "batchSize": 500
  },
  "description": "Historic read"
}
```

Available arguments for `c8y_ua_command_HistoricReadOperation` :

Field	Type	Mandatory	Description
nodeId	string	yes	ID of the node to execute the operation
dateFrom	dateTime	yes	The values are read starting from this time
dateTo	dateTime	yes	The values are read until this time

Field	Type	Mandatory	Description
ranges	string	no	<p>The index ranges of a subset of the multi-dimension array from the read attribute. The syntax is according to the OPC UA specification and will be transformed to NumericRange.</p> <div> <p>NumericRange: <dimension> [, ' <dimension>] <dimension>: <index> [: ' <index>]</p> </div> <p>Example values to define the range for a 1D array is "0:1", for a 2D array is "0:1,0:1"</p>
batchSize	integer	no	Batch size for each history read call to the OPC UA server. Default is 200.
processMappings	boolean	no	If set to false then the read values will not be processed based on the device protocol mapping. Default is true. Note that any data created from historic data using a device protocol will carry the same processing mode as specified in the mapping.
tagType	string	no	Possible tagType values are "TAG" and "NO_TAG". "TAG" appends "_Historic" for both the mapping types and for the measurement mappings. Default is "TAG".

HISTORIC DATA BINARY UPLOAD

This operation reads historic values and only saves those values to a file which can be retrieved using the binary API. The `deviceId` of the operation can be either the OPC UA Server device ID or the generated device ID (child device of OPC UA Server).

```
{
  "deviceId": "<server-or-child-device-Id>",
  "c8y_ua_command_HistoricDataUploadOperation": {
    "nodeId": "ns=2;s=MyLevel",
    "dateFrom": "2019-01-03T09:53:00+02:00",
    "dateTo": "2019-06-13T18:53:00+02:00",
    "chunkSize": 1,
    "compress": true,
    "batchSize": 150000
  },
  "description": "Upload history data"
}
```

The binary file representations, which can be queried using binary API, are created with the type "c8y_ua_HistoricData" and an operationId with the value of the operation with which it has been generated.

Available arguments for `c8y_ua_command_HistoricDataUploadOperation` :

Field	Type	Mandatory	Description
nodeId	string	yes	ID of the node to execute the operation
dateFrom	dateTime	yes	The values are read starting from this time
dateTo	dateTime	yes	The values are read until this time

Field	Type	Mandatory	Description
ranges	string	no	The index ranges of a subset of the multi-dimension array from the read attribute. The syntax is according to the OPC UA specification and will be transformed to NumericRange. <div> NumericRange: <dimension> ['<dimension>] <dimension>: <index> ['<index>] </div> Example values to define the range for a 1D array is "0:1", for a 2D array is "0:1,0:1"
batchSize	integer	no	Batch size for each history read call to the OPC UA server. Default is 100000.
chunkSize	integer	no	The maximum file size in Mb for the output binary file. For each batch, the files can be divided based on this limit.
compress	boolean	no	If set the false the output chunks are compressed. Default is false.

READ FILE

Prerequisites:

- Open and Read methods for the file node must be implemented on server side, either as the children of the file node itself or as the children of the data type node

With this operation, a file can be downloaded from the OPC UA server at the given fileId. The `deviceId` of the operation can be either the OPC UA Server device ID or the generated device ID (child device of OPC UA Server).

```
{
  "deviceId": "<server-or-child-device-Id>",
  "c8y_ua_command_ReadFileOperation": {
    "fileNodeId": "ns=2;s=sampleFile",
    "bufferSize": <bufferSize>
  },
  "description": "Read sample file"
}
```

Available arguments for `c8y_ua_command_ReadFileOperation` :

Field	Type	Mandatory	Description
fileNodeId	string	yes	ID of the node to execute the operation
bufferSize	long	no	Maximum value can be 10 MB. The default size, if not set in the request, is 1MB. This will not limit the size of the file to be read. If the size is bigger, multiple read operations are triggered.
skipResetPosition	boolean	no	If set to true then the position to read the file is reset before reading the file. Default is false.

After the downloaded file has been read successfully (see **Control** tab of the device) it is available in **Management > Files repository** in the Administration application for download to local file system.

Alternatively, you can check the binary folder by using the binary API like this:

```
{{url}}/inventory/binaries
```

This returns a JSON response like this:

```
{
  "self": "http://<tenant-domain>/inventory/binaries?pageSize=5&currentPage=1",
  "managedObjects": [
    {
      "owner": "<device-owner>",
      "type": "ua-file-type",
      "lastUpdated": "2021-05-17T14:33:21.074Z",
      "name": "ns=2;s=sampleFile",
      "self": "http://<tenant-domain>/inventory/binaries/2351",
      "id": "2351",
      "c8y_IsBinary": "",
      "length": 13268,
      "contentType": "application/octet-stream"
    }
  ],
  "statistics": {
    "totalPages": 1,
    "currentPage": 1,
    "pageSize": 5
  }
}
```

Now download is possible with the self link provided inside the managedObjects section of the JSON response.

For further information, refer to [binaries API](#) in the Cumulocity OpenAPI Specification.

WRITE VALUE

This operation writes values to the node/nodes. The `deviceId` of the operation can be either the OPC UA Server device ID or the generated device ID (child device of OPC UA Server).

```
{
  "deviceId": "<server-or-child-device-id>",
  "c8y_ua_command_WriteValue": {
    "values": {
      "ns=3;s=LocalizedText": {
        "value": "This is a localized text"
      },
      "ns=3;s=Double": {
        "value": "3.14159"
      }
    }
  },
  "description": "Write values to different nodes"
}
```

Available arguments for `c8y_ua_command_WriteValue` :

Field	Type	Mandatory	Description
values	map<string, rangedValue>	yes	Map with ID of the node to execute the operation and RangedValue to set
expirationTime	dateTime	no	Expiration time to execute the operation. The operation is executed if it is before the given expiration time. Otherwise, the operation will fail. In this case, "Operation expired" is returned as failure reason.

Available arguments for type `rangedValue` (used as map value in `c8y_ua_command_WriteValue.values`):

Field	Type	Mandatory	Description
value	string	yes	Value to set to the node attribute
ranges	string	no	The index ranges of a subset of the multi-dimension array. The syntax for the ranges is according to the OPC UA specification and will be transformed to <code>NumericRange</code> .

```
NumericRange: <dimension> [',' <dimension>]
<dimension>: <index> [':' <index>]
```

Example values to define the range for a 1D array is "0:1", for a 2D array is "0:1,0:1"

WRITE ATTRIBUTE

This operation is similar to the previous one, but instead of writing to the value attribute, this operation writes attributes' values to any writable attributes. The following example writes two different attributes to two different nodes. The `deviceId` of the operation can be either the OPC UA Server device ID or the generated device ID (child device of OPC UA Server).

```
{
  "deviceId": "<server-or-child-device-id>",
  "c8y_ua_command_WriteAttribute": {
    "values": {
      "ns=3;s=LocalizedText": {
        "attribute": "13",
        "value": "This is a localized text"
      },
      "ns=3;s=Double": {
        "attribute": "13",
        "value": "3.14159"
      }
    }
  },
  "description": "Write attributes' values to different attributes of different nodes"
}
```

Available arguments for `c8y_ua_command_WriteAttribute`:

Field	Type	Mandatory	Description
values	map<string, attributeRangedValue>	yes	Map with ID of the node to execute the operation and <code>AttributeRangedValue</code> to set
expirationTime	dateTime	no	Expiration time to execute the operation. The operation is executed if it is before the given expiration time. Otherwise, the operation will fail. In this case, "Operation expired" is returned as failure reason.

Available arguments for type `attributeRangedValue` (used as map value in `c8y_ua_command_WriteAttribute.values`):

Field	Type	Mandatory	Description
attribute	string	yes	ID of the attribute according to the OPC UA specification

Field	Type	Mandatory	Description
value	string	yes	Value to set to the node attribute
ranges	string	no	The index ranges of a subset of the multi-dimension array. The syntax for the ranges is according to the OPC UA specification and will be transformed to NumericRange.

```
NumericRange: <dimension> [',' <dimension>]
<dimension>: <index> [':' <index>]
```

Example values to define the range for a 1D array is "0:1", for a 2D array is "0:1,0:1"

Example operation with ranges fragment:

```
{
  "deviceId": "<server-device-Id>",
  "c8y_ua_command_WriteAttribute": {
    "values": {
      "ns=3;s=FloatArray": {
        "attribute": "13",
        "ranges": "0:1",
        "value": "2.0,4.0"
      }
    }
  },
  "description": "Write attribute value to array attribute"
}
```

GET METHOD DESCRIPTION

This operation reads the description of a method node. The `deviceId` of the operation can be either the OPC UA Server device ID or the generated device ID (child device of OPC UA Server).

```
{
  "deviceId": "<server-or-child-device-Id>",
  "c8y_ua_command_GetMethodDescriptionOperation": {
    "nodeId": "ns=2;s=MyMethod"
  },
  "description": "get method description"
}
```

Available arguments for `c8y_ua_command_GetMethodDescriptionOperation` :

Field	Type	Mandatory	Description
nodeId	string	yes	ID of the node to execute the operation

The result describes a method, it's parent object, input and output arguments.

```

{
  "nodeId": "ns=2;s=MyMethod",
  "name": "MyMethod",
  "parentNodeId": "ns=2;s=MyDevice",
  "parentName": "MyDevice",
  "inputArguments": [{
    "name": "Operation",
    "description": "The operation to perform on parameter: valid functions are sin, cos, tan, pow",
    "dataType": "String",
    "dataTypeId": "i=12"
  },
  {
    "name": "Parameter",
    "description": "The parameter for operation",
    "dataType": "Double",
    "dataTypeId": "i=11"
  }
  ],
  "outputArguments": [{
    "name": "Result",
    "description": "The result of 'operation(parameter)'",
    "dataType": "Double",
    "dataTypeId": "i=11"
  }]
}

```

CALL METHOD

This operation calls the method on the OPC UA server. It requires complete input arguments with an additional “value” fragment. The **deviceId** of the operation can be either the OPC UA Server device ID or the generated device ID (child device of OPC UA Server).

```

{
  "deviceId": "<server-or-child-device-Id>",
  "c8y_ua_command_CallMethodOperation": {
    "request": {
      "nodeId": "ns=2;s=MyMethod",
      "arguments": [{
        "name": "Operation",
        "description": "The operation to perform on parameter: valid functions are sin, cos, tan, pow",
        "dataType": "String",
        "dataTypeId": "i=12",
        "value": "pow"
      },
      {
        "name": "Parameter",
        "description": "The parameter for operation",
        "dataType": "Double",
        "dataTypeId": "i=11",
        "value": "5"
      }
    ]
  }
},
  "description": "call method"
}

```

Available arguments for **c8y_ua_command_CallMethodOperation** :

Field	Type	Mandatory	Description
request	methodRequest	yes	Request to send to the OPC UA Server

Field	Type	Mandatory	Description
expirationTime	dateTime	no	Expiration time to execute the operation. The operation is executed if it is before the given expiration time. Otherwise, the operation will fail. In this case, "Operation expired" is returned as failure reason.

Available arguments for type methodRequest (used in `c8y_ua_command_CallMethodOperation.request`):

Field	Type	Mandatory	Description
nodeId	string	yes	ID of the node to execute the operation
arguments	list<methodArgument> >	no	List of arguments for the method request
objectNodeId	string	no	The nodeId of the Object or ObjectType that is the source of a HasComponent reference (or subtype of HasComponent reference) to the method
parseResponse	boolean	no	If set to true, the value is converted to JSON and the actual value is stored in the rawValue fragment in response. Default is true

Available arguments for type methodArgument (used in `c8y_ua_command_CallMethodOperation.request.arguments`):

Field	Type	Mandatory	Description
name	string	no	Name of the method argument
description	string	no	Description of the method argument
dataType	string	yes	Data type of the method argument
dataTypeId	string	yes	ID of the data type in OPC UA Server
value	string	yes	Value for the method argument
arrayDimension	string	no	Array dimension for the value to set if the value is an array

The result contains all output arguments with values set by the OPC UA server. Power of 5 is 25:

```
{
  "statusCode": 0,
  "result": [{
    "name": "Result",
    "description": "The result of 'operation(parameter)",
    "dataType": "Double",
    "dataTypeId": "i=11",
    "value": "25.0"
  }]
}
```

TESTING A DEVICE TYPE AGAINST A NODE ON AN OPC UA SERVER

This operation allows for testing a device type against a specific node on an OPC UA server. The operation result provides diagnostic information if the device type could be applied:


```
{
  "deviceId":"<server-device-id>",
  "c8y_ua_command_TestDeviceTypeMatching":{
    "deviceTypeId":"<device-type-id>",
    "rootNodeId":"<node-id>"
  },
  "description":"Test Device Type"
}
```

Available arguments for `c8y_ua_command_TestDeviceTypeMatching` :

Field	Type	Mandatory	Description
deviceTypeId	string	yes	ID of the managed object containing the device protocol
rootNodeId	string	yes	ID of the root node to execute the operation

If the device type can be applied to the given node, the operation result confirms this:

```
{
  "creationTime":"2020-08-20T12:28:57.973Z",
  "deviceId":"12789",
  "deviceName":"Test Server",
  "id":"15478",
  "status":"SUCCESSFUL",
  "c8y_ua_command_TestDeviceTypeMatching":{
    "deviceTypeId":"14989",
    "rootNodeId":"nsu=urn:cumulocity:opcua:test:server;s=HelloWorld/Dynamic2"
  },
  "c8y_Command":{
    "result":{"\n \matches\": true\n"}
  },
  "description":"Test Device Type"
}
```

Otherwise, the operation result provides an explanation why the device type could not be matched to the given root node:

```
{
  "creationTime":"2020-08-20T12:34:01.524Z",
  "deviceId":"12789",
  "deviceName":"Milo Reloaded",
  "id":"15688",
  "status":"SUCCESSFUL",
  "c8y_ua_command_TestDeviceTypeMatching":{
    "deviceTypeId":"14989",
    "rootNodeId":"nsu=urn:cumulocity:opcua:test:server;s=HelloWorld/Dynamic9"
  },
  "c8y_Command":{
    "result":{"\n \matches\": false,\n \reason\": "\nDoes not match browse path regex constraint, constraints: (*.Dynamic[1-3]), actual:
[[http://opcfoundation.org/UA:/Root, http://opcfoundation.org/UA:/Objects, urn:cumulocity:opcua:test:server:Dynamic Playground,
urn:cumulocity:opcua:test:server:Dynamic9]]\n"},
    "syntax":null,
    "text":null
  },
  "description":"Test Device Type"
}
```

ANALYZING THE SET OF NODES TO WHICH A DEVICE TYPE CAN BE APPLIED (DRY RUN)

As explained earlier, the Cumulocity OPC UA gateway performs an auto-discovery to determine the set of nodes that match a certain device protocol ("device type"). The following operation performs an auto-discovery for the given device protocol on the server, without

actually applying it to any node ("dry run"):

```
{
  "deviceId":"<server-device-id>",
  "c8y_ua_command_DryRunDeviceTypeMatching":{
    "deviceTypeId":"<device-type-id>"
  },
  "description":"Dry Run Device Type"
}
```

Available arguments for `c8y_ua_command_DryRunDeviceTypeMatching` :

Field	Type	Mandatory	Description
deviceTypeId	string	yes	ID of the managed object containing the device protocol

The result of the operation contains the set of nodes that match the device protocol. In addition to that, the fragment `matchednodes` is added to the operation. It contains a JSON representation of the matched nodes.

```

{
  "creationTime": "2020-08-20T12:22:07.947Z",
  "deviceId": "12789",
  "deviceName": "Test Server",
  "id": "15187",
  "status": "SUCCESSFUL",
  "c8y_Command": {
    "result": "Device protocol is currently disabled. Device protocol would be applied to the following nodes: [\n {\n  \"nodeId\": \"nsu=urn:cumulocity:opcua:test:server;s=HelloWorld/Dynamic2\", \n  \"deviceTypeId\": \"14989\", \n  \"mappedTargetNodes\": [\n {\n  \"browsePath\": [\n    \"urn:cumulocity:opcua:test:server:Double\", \n    \"targetNodeId\": \"nsu=urn:cumulocity:opcua:test:server;s=HelloWorld/Dynamic/Double2\", \n    \"attrs\": {\n  \"nodeId\": \"nsu=urn:cumulocity:opcua:test:server;s=HelloWorld/Dynamic1\", \n  \"deviceTypeId\": \"14989\", \n  \"mappedTargetNodes\": [\n {\n  \"browsePath\": [\n    \"urn:cumulocity:opcua:test:server:Double\", \n    \"targetNodeId\": \"nsu=urn:cumulocity:opcua:test:server;s=HelloWorld/Dynamic/Double1\", \n    \"attrs\": {\n  \"nodeId\": \"nsu=urn:cumulocity:opcua:test:server;s=HelloWorld/Dynamic3\", \n  \"deviceTypeId\": \"14989\", \n  \"mappedTargetNodes\": [\n {\n  \"browsePath\": [\n    \"urn:cumulocity:opcua:test:server:Double\", \n    \"targetNodeId\": \"nsu=urn:cumulocity:opcua:test:server;s=HelloWorld/Dynamic/Double3\", \n    \"attrs\": {\n  \"nodeId\": \"nsu=urn:cumulocity:opcua:test:server;s=HelloWorld/Dynamic2\"
    \"mappedTargetNodes\": [
      {
        \"targetNodeId\": \"nsu=urn:cumulocity:opcua:test:server;s=HelloWorld/Dynamic/Double2\",
        \"browsePath\": [
          \"urn:cumulocity:opcua:test:server:Double\"
        ]
      }
    ],
    \"deviceTypeId\": \"14989\",
    \"nodeId\": \"nsu=urn:cumulocity:opcua:test:server;s=HelloWorld/Dynamic2\"
  },
  {
    \"mappedTargetNodes\": [
      {
        \"targetNodeId\": \"nsu=urn:cumulocity:opcua:test:server;s=HelloWorld/Dynamic/Double1\",
        \"browsePath\": [
          \"urn:cumulocity:opcua:test:server:Double\"
        ]
      }
    ],
    \"deviceTypeId\": \"14989\",
    \"nodeId\": \"nsu=urn:cumulocity:opcua:test:server;s=HelloWorld/Dynamic1\"
  },
  {
    \"mappedTargetNodes\": [
      {
        \"targetNodeId\": \"nsu=urn:cumulocity:opcua:test:server;s=HelloWorld/Dynamic/Double3\",
        \"browsePath\": [
          \"urn:cumulocity:opcua:test:server:Double\"
        ]
      }
    ],
    \"deviceTypeId\": \"14989\",
    \"nodeId\": \"nsu=urn:cumulocity:opcua:test:server;s=HelloWorld/Dynamic3\"
  }
    ],
    \"syntax\": null,
    \"text\": null
  },
  \"description\": \"Dry Run Device Type\",
  \"c8y_ua_command_DryRunDeviceTypeMatching\": {
    \"deviceTypeId\": \"14989\"
  }
}

```

GET THE CURRENT APPLICATION STATE OF A DEVICE TYPE

In order to know what is the current state of a device type application, use the following operation:

```
{
  "description": "Get device type application state",
  "deviceId": "{server-device-Id}",
  "c8y_ua_command_GetDeviceTypeApplicationState": {
    "deviceTypeId": "{device protocol ID}",
    "matchingRootNodes": [{"root node ID #1"}, {"root node ID #2"}]
  }
}
```

Available arguments for `c8y_ua_command_GetDeviceTypeApplicationState` :

Field	Type	Mandatory	Description
deviceTypeId	string	yes	ID of the managed object containing the device protocol
matchingRootNodes	list<string>	no	List of ID of the root nodes to execute the operation. When it is not provided, the application state of all matching nodes will be returned.

The result will be populated into the operation result as a map of nodes telling whether the device type has been applied to that node or not.

Sample result when the device type has been applied to node #1 but not node #2:

```
{
  "{root node ID #1}": true,
  "{root node ID #2}": false
}
```

EXPIRING OPERATIONS

In certain cases it is desirable that the OPC UA gateway executes an operation only if it processes it before a given expiration time. Providing such an optional expiration time is supported for the following OPC UA operations:

- Reading the value of a node/nodes
- Reading all attributes of a node
- Reading an attribute
- Read complex
- Write value
- Write attribute
- Call method

For all the given operations this expiry mechanism can be activated by supplying an `expirationTime` fragment inside the operation body.

The following example shows how to mark a read operation as expiring:

```
{
  "deviceId": "<server-device-Id>",
  "c8y_ua_command_ReadValue": {
    "nodes": ["ns=3;s=FloatArray"],
    "expirationTime": "2021-02-08T15:00:00.000Z"
  },
  "description": "Expiring read value"
}
```

The operation above will only perform a read on the OPC UA server if processed by the gateway before the 8th of February, 2021 15:00. Otherwise, the operation will fail. In this case, `Operation expired` is returned as failure reason.

OPC UA EVENTS

MODEL CHANGE EVENTS

The model change events are created by the OPC UA server to notify about the changes in an address space node on runtime. The gateway subscribes to the events of type `BaseModelChangeEvent` on connection to the servers. The subscription to this event can be enabled or disabled for each server using the `subscribeModelChangeEvent` property during the server configuration. The changes for the events with type `GeneralModelChangeEvent` and `SemanticChangeEvent`, which are subtypes of `BaseModelChangeEvent`, are handled and address space information is persisted in the Cumulocity inventory as well as in the local address space file of the gateway.

TROUBLESHOOTING

PERMISSION DENIED ERROR WHEN RUNNING THE GATEWAY JAR FILE ON A LINUX OS

```
Caused by: java.io.FileNotFoundException: /root/.opcua/data/cumulocity-opcua-gateway.db (Permission denied)
    at java.io.RandomAccessFile.open0(Native Method)
    at java.io.RandomAccessFile.open(RandomAccessFile.java:316)
    at java.io.RandomAccessFile.<init>(RandomAccessFile.java:243)
    at org.mapdb.volume.RandomAccessFileVol.<init>(RandomAccessFileVol.java:51)
    ... 94 common frames omitted
```

If the following error appears, add a `baseDir` property to the YAML file. For example:

```
db:
  baseDir: ${user.home}/.opcua/profile/data
```

UNKNOWN HOST EXCEPTION WHEN RUNNING THE GATEWAY JAR

This error appears if the provided `baseUrl` property in the YAML file is incorrect.

FAILED TO LOAD PROPERTY SOURCE FROM LOCATION WHEN RUNNING THE GATEWAY JAR

The following error appears if the indentation of the properties in the YAML file is incorrect.

```
java.lang.IllegalStateException: Failed to load property source from location 'classpath:/application-profile.yaml'
    at org.springframework.boot.context.config.ConfigFileApplicationListenerLoader.loadIntoGroup(ConfigFileApplicationListenerLoader.java:476)
    at org.springframework.boot.context.config.ConfigFileApplicationListenerLoader.load(ConfigFileApplicationListenerLoader.java:450)
    at org.springframework.boot.context.config.ConfigFileApplicationListenerLoader.load(ConfigFileApplicationListenerLoader.java:386)
    at org.springframework.boot.context.config.ConfigFileApplicationListener.addPropertySources(ConfigFileApplicationListener.java:225)
    at org.springframework.boot.context.config.ConfigFileApplicationListener.postProcessEnvironment(ConfigFileApplicationListener.java:195)
    at org.springframework.boot.context.config.ConfigFileApplicationListener.onApplicationEnvironmentPreparedEvent(ConfigFileApplicationListener.java:182)
    at org.springframework.boot.context.config.ConfigFileApplicationListener.onApplicationEvent(ConfigFileApplicationListener.java:168)
    at org.springframework.context.event.SimpleApplicationEventMulticaster.doInvokeListener(SimpleApplicationEventMulticaster.java:172)
    at org.springframework.context.event.SimpleApplicationEventMulticaster.invokeListener(SimpleApplicationEventMulticaster.java:165)
    at org.springframework.context.event.SimpleApplicationEventMulticaster.multicastEvent(SimpleApplicationEventMulticaster.java:139)
    at org.springframework.context.event.SimpleApplicationEventMulticaster.multicastEvent(SimpleApplicationEventMulticaster.java:122)
    at org.springframework.boot.context.event.EventPublishingRunListener.environmentPrepared(EventPublishingRunListener.java:74)
    at org.springframework.boot.SpringApplicationRunListeners.environmentPrepared(SpringApplicationRunListeners.java:54)
    at org.springframework.boot.SpringApplication.prepareEnvironment(SpringApplication.java:325)
    at org.springframework.boot.SpringApplication.run(SpringApplication.java:296)
    at org.springframework.boot.builder.SpringApplicationBuilder.run(SpringApplicationBuilder.java:134)
    at com.cumulocity.opcua.client.gateway.GatewayApplication.main(GatewayApplication.java:20)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at org.springframework.boot.loader.MainMethodRunner.run(MainMethodRunner.java:48)
    at org.springframework.boot.loader.Launcher.launch(Launcher.java:87)
    at org.springframework.boot.loader.Launcher.launch(Launcher.java:50)
    at org.springframework.boot.loader.JarLauncher.main(JarLauncher.java:51)
```

JAVA.NET.BINDEXCEPTION: ADDRESS ALREADY IN USE

```

Caused by: java.net.BindException: Address already in use
    at sun.nio.ch.Net.bind0(Native Method)
    at sun.nio.ch.Net.bind(Net.java:433)
    at sun.nio.ch.Net.bind(Net.java:425)
    at sun.nio.ch.AsynchronousServerSocketChannelImpl.bind(AsynchronousServerSocketChannelImpl.java:162)
    at org.apache.sshd.common.io.nio2.Nio2Acceptor.bind(Nio2Acceptor.java:64)
    at org.apache.sshd.common.io.nio2.Nio2Acceptor.bind(Nio2Acceptor.java:88)
    at org.apache.sshd.server.SshServer.start(SshServer.java:310)
    at com.github.Foninus.ssh.shell.SshShellConfiguration.startServer(SshShellConfiguration.java:46)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at org.springframework.beans.factory.annotation.InitDestroyAnnotationBeanPostProcessor$LifecycleElement.invoke(InitDestroyAnnotationBeanPostProcessor.java:366)
    at org.springframework.beans.factory.annotation.InitDestroyAnnotationBeanPostProcessor$LifecycleMetadata.invokeInitMethods(InitDestroyAnnotationBeanPostProcessor.java:311)
    at org.springframework.beans.factory.annotation.InitDestroyAnnotationBeanPostProcessor.postProcessBeforeInitialization(InitDestroyAnnotationBeanPostProcessor.java:134)
    ... 24 common frames omitted

```

If this error appears, a Java process is running in the background. To fix this issue, the process must be stopped/killed.

CHANGING THE LOG LEVEL FOR TROUBLESHOOTING

For troubleshooting purposes, we recommend you to enable the DEBUG log level for subpackages and root if required, and send the log file to [product support](#).

For example:

```

<logger name="com.cumulocity.opcua.client.gateway" level="DEBUG"/>
<logger name="com.cumulocity" level="DEBUG"/>
<logger name="c8y" level="DEBUG"/>

<root level="DEBUG">
  <appender-ref ref="FILE" />
  <appender-ref ref="STDOUT"/>
</root>

```

If there is an unknown error during the address space scans, enable DEBUG or TRACE log level specifically for the scanners:

```

<logger name="com.cumulocity.opcua.client.BaseAddressSpaceScanner" level="DEBUG" />
<logger name="com.cumulocity.opcua.client.OpcuaAddressSpaceFullScanner" level="DEBUG" />
<logger name="com.cumulocity.opcua.client.OpcuaAddressSpaceLightScanner" level="DEBUG" />
<logger name="com.cumulocity.opcua.client.OpcuaAddressSpaceReverseFullScanner" level="DEBUG" />

<root level="INFO">
  <appender-ref ref="FILE" />
  <appender-ref ref="STDOUT"/>
</root>

```

For additional information about log levels, refer to the [Logback architecture documentation](#).

JAVA MANAGEMENT EXTENSIONS (JMX)

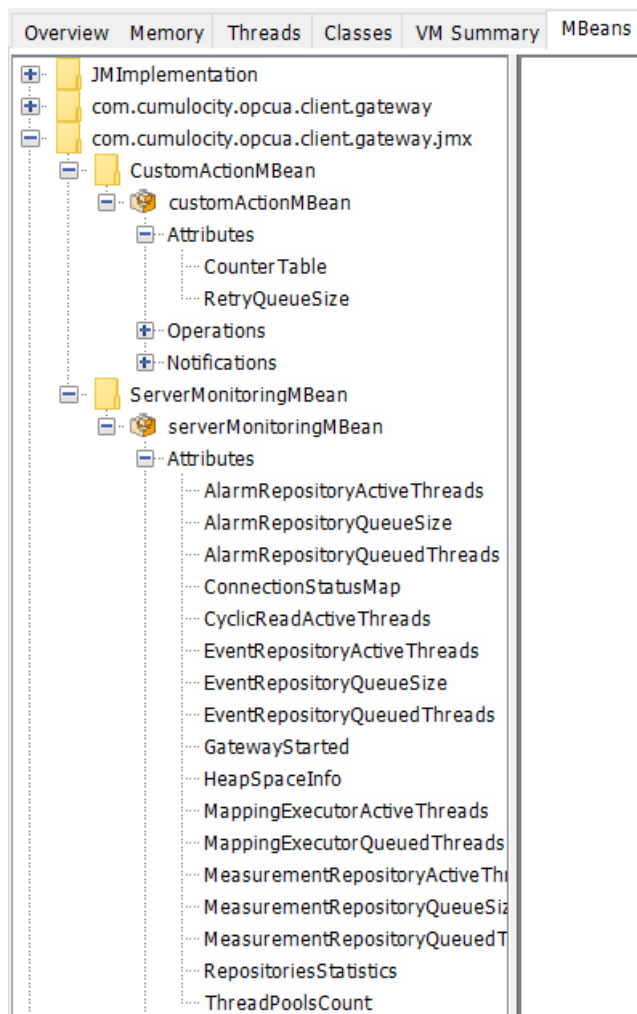
For additional monitoring, the gateway component provides MBeans. These MBeans get exposed if the following configuration is set in the *application.yaml* file:

```

spring:
  jmx:
    enabled: true

```

Via jconsole the MBeans can be selected and the following attributes can be accessed:



It can be useful to get some statistics for custom actions in particular. These attributes can be retrieved from the CustomActionMBean:

1. Table of all called URLs separated by HTTP return code and retry count.

The screenshot shows the JMX console with the 'MBeans' tab selected. The left pane displays a tree of MBeans, with 'Counter Table' selected under 'CustomActionMBean'. The right pane shows the 'Attribute value' for 'Counter Table'.

Attribute value

Name	Value
key	http://localhost:8080/api/test_200_0
value	20

Refresh

MBeanAttributeInfo

Name	Value
Attribute:	
Name	Counter Table
Description	counter Table
Readable	true
Writable	false
Is	false
Type	javax.management.openmbean.TabularData

Descriptor

Name	Value
Attribute:	
default	
descriptor Type	attribute
displayName	Counter Table
getMethod	getCounter Table
name	Counter Table

The key entry of the table consists of:

{URL}_{HTTP Response Code}_{Retry Count}

- If retry is enabled, the queue size of the retry queue can be monitored.

The screenshot shows the JMX console with the 'MBeans' tab selected. The left pane displays a tree of MBeans, with 'RetryQueueSize' selected under 'CustomActionMBean'. The right pane shows the 'Attribute value' for 'RetryQueueSize'.

Attribute value

Name	Value
RetryQueueSize	0

Refresh

MBeanAttributeInfo

Name	Value
Attribute:	
Name	RetryQueueSize
Description	retryQueueSize
Readable	true
Writable	false
Is	false
Type	java.lang.Long

Descriptor

Name	Value
Attribute:	
default	
descriptor Type	attribute
displayName	RetryQueueSize
getMethod	getRetryQueueSize
name	RetryQueueSize

OPC UA MAJOR VERSION UPGRADE NOTES

UPGRADING FROM 1021 TO 1022 GATEWAY VERSION

Major version 1022 of the OPC UA gateway introduces significant changes, including a new internal database and updated Java version. To ensure a smooth transition, follow these steps:

1. **Back up the existing data:** Before proceeding with the upgrade, back up your existing gateway data folder to prevent any potential data loss.
2. **Stop the existing gateway:** Ensure that the current gateway instance is stopped before starting the upgrade process.
3. **Download and run the data migration tool:** A data migration tool is provided to facilitate the transition of your existing data to the new format. Download the tool from <https://resources.cumulocity.com/examples/opc-ua/migration-tool-for-1022-upgrade/> and follow the instructions in the *README.md* file to execute the migration.
4. **Install Java 17:** The new gateway version requires Java 17. Ensure that your system has Java 17 installed and configured properly.
5. **Update the gateway JAR file:** Replace the existing gateway JAR file with the new version.
6. **Start the new gateway:** After completing the above steps, start the new gateway instance.

Rollback procedure

If at any point you encounter issues, stop the new gateway and restore the backup of your data folder. Then, restart the previous version of the gateway using Java 11.

CLOUD FIELDBUS

INTRODUCTION

Cloud Fieldbus enables you to connect any fieldbus device to Cumulocity. This connection can be done within minutes and at minimal cost and provides high levels of security and reliability. Connected devices can be completely managed from Cumulocity including data collection, visualization, fault management and remote control.

With Cumulocity Cloud Fieldbus you can collect data from fieldbus devices and remotely manage them. This section describes how to

- [Connect](#) fieldbus devices to Cumulocity.
- [Manage](#) the connected fieldbus devices.
- [Configure](#) the remote management capabilities of particular types of devices and [import and export](#) them.

Cloud Fieldbus is supported out of the box by several devices. For information on supported devices, refer to the [Cumulocity Partner Devices Ecosystem](#) which allows to filter for all devices which offer full functional support with Cloud Fieldbus.

INFO

To support Cloud Fieldbus with your terminal please contact [product support](#).

CONNECTING FIELDBUS DEVICES

For the following instructions, it is assumed that you have a Cloud Fieldbus terminal available and it is registered as a device in your Cumulocity tenant. To register a terminal with Cumulocity, follow the instructions provided with the terminal.

CONNECTING MODBUS/RTU DEVICES

To connect a Modbus/RTU device, follow these steps:

1. Physically connect the Modbus/RTU device through a RS-485 or RS-232 cable to the terminal.
2. Assign the device a unique Modbus address according to the instructions provided with the Modbus device (for example by setting a jumper on the device).
3. Check the serial communication settings of the device according to the instructions provided with the Modbus device (that is, baud rates and communication protocol). These must match with all devices on the bus.
4. In the Device Management application, click **All devices** in the **Devices** menu in the navigator. In the device list, select the terminal and switch to the **Modbus** tab.
5. Change the communication settings shown in the **Serial communication** section to match the settings on the bus, if needed.
6. Change the transmit rate and the polling rate according to your requirements. The transmit rate is the frequency where measurements are sent to Cumulocity. The polling rate is the frequency at which the Modbus devices are polled for changes.
7. Click **Save** to save your settings.

The screenshot shows the 'MQTT Device tedge_78f172b18d83' configuration page. The left sidebar contains a 'DEVICE MANAGEMENT' menu with options like Home, Devices, Registration, All devices, Map, Simulators, Availability, Overviews, Groups, EMEA, LWM2M Devices, Device types, and Management. The main content area is divided into two sections: 'Configuration' and 'RTU'. The 'Configuration' section has fields for 'Port' (9600), 'Data bits' (8), and 'Stop bits' (2), with a 'Save' button. The 'RTU' section has a table with columns 'Name', 'Device type', and 'Address'. Below the table, there are input fields for 'Name' (ABC-DX-ADCO-5432), 'Device type' (Modbus Protocol 1), and 'Address' (42), with 'Cancel' and 'Add' buttons. At the bottom, there is a green button labeled 'Add RTU device'.

To add child devices

1. To start the communication between the terminal and the Modbus/RTU device, click **Add RTU device**.
2. Enter a name for the device and select the device protocol from the dropdown field. See [Configuring fieldbus device protocols](#) for information on how to add a new device protocol. Set the Modbus address of the connected device.
3. Click **Add**. Cumulocity will now send a notification to the Modbus terminal that a new device is ready to be managed. This may take a few seconds.

After completion, a new child device has been added to the terminal and can now be managed. You can click on the name of the device in the list to navigate to the device. If you have not yet added Modbus devices to the terminal, you may have to reload your browser window to make the **Child Devices** tab visible.

CONNECTING MODBUS/TCP DEVICES

To connect a Modbus/TCP device, follow these steps:

1. Make sure that the Modbus/TCP device is connected to the terminal, that is, directly through an Ethernet cable or through a switch. If you are using a Modbus gateway, configure the gateway in a way it can communicate with the Modbus devices behind the gateway.
2. Check the network settings of the device using the instructions provided with the device.
3. In the Device Management application, click **All devices** in the **Devices** menu in the navigator. In the device list, select the terminal and switch to the **Network** tab. Verify that the LAN settings of the terminal match the settings of the device so that TCP communication can be established.
4. Switch to the **Modbus** tab.
5. Change the transmit rate and the polling rate according to your requirements. The transmit rate is the frequency at which measurements are sent to Cumulocity. The polling rate is the frequency at which the Modbus devices are polled for changes.
6. Click **Save** to save your settings.

To add child devices

1. To start the communication between the terminal and the Modbus/TCP device, click **Add TCP device**.
2. Enter a name for the device and select the device protocol from the dropdown field. See [Configuring fieldbus device types](#) for information on how to add a new device protocol. Set the Modbus address and the IP address of the connected device.
3. Click **Add**.

Cumulocity will now send a notification to the Modbus terminal that a new device is ready to be managed. This may take a few seconds.

INFO

It is assumed that all Modbus/TCP communication uses the standard Modbus/TCP port 502. On the NTC-6200 and NTC 220 series, the port to be used can be configured through the variable "service.cumulocity.modbus.port" via the device shell or the local web user interface of the device.

CONNECTING CAN DEVICES

To connect a CAN device, follow these steps:

1. Physically connect the CAN device to the terminal.
2. Check the serial communication baud rate of the device according to the instructions provided with the device. These must match all devices on the bus.
3. In the Device Management application, click **All devices** in the **Devices** menu in the navigator. In the device list, select the terminal and switch to the **CAN Bus** tab.
4. Change the baud rate setting shown in the section **CAN Bus communication** to match the settings on the bus, if needed.
5. Change the transmit rate according to your requirements. The transmit rate is the frequency where measurements are sent to Cumulocity.
6. Click **Save** to save your settings.

The screenshot displays the MQTT Device Management interface. On the left, a sidebar contains navigation links: Home, Devices, Registration, All devices, Map, Simulators, Availability, Overviews, Groups, EMEA, and a list of devices with their URNs. The main panel is titled 'MQTT Device tedge_78f172b18d83' and shows the 'CAN Bus' configuration. It includes fields for 'Transmit rate' (60 seconds) and 'Baud rate' (125000). Below these, there is a 'CAN' section with a 'Device type' dropdown menu currently set to 'Alta-feed-RSS-IQ-I'. At the bottom of this section are 'Cancel' and 'Add' buttons, and a green 'Add CAN device' button. A help icon is visible in the bottom right corner.

To add child devices

1. To start the communication between the terminal and the CAN device, click **Add CAN device**.
2. Enter a name for the device and select a device protocol from the dropdown field. See [Configuring fieldbus device types](#) for information on how to add a new device protocol.
3. Click **Add**.

Cumulocity will now send a notification to the fieldbus terminal that a new device is ready to be managed. This may take a few seconds.

After completion, a new child device has been added to the terminal and can now be managed. You can click on the name of the device in the list to navigate to the device. If you have not yet added fieldbus devices to the terminal, you may have to reload your browser window to make the **Child devices** tab visible.

CONNECTING PROFIBUS DEVICES

Connecting Profibus devices slightly differs from the regular plug & play approach of other Cloud Fieldbus protocols. The gateway acts as a device on the Profibus so it can easily be integrated into an existing infrastructure. This means that Profibus data must be actively sent to the gateway though. Typically, this is done by programming a PLC to actively send information to the gateway via its configured Profibus device address.

To connect a Profibus device, follow these steps:

1. Physically wire the Profibus device to the terminal.
2. In the Device Management application, click **All devices** in the **Devices** menu in the navigator. In the device list, select the terminal and switch to the **Profibus** tab.
3. The baud rate is automatically detected by the gateway and it is displayed here.
4. Change the transmit rate according to your requirements. The transmit rate is the interval at which measurements are sent to Cumulocity.
5. Set the device address of the terminal.
6. Configure your Profibus controller to communicate to that device address. To do so, refer to the gateway manual (see for example Smart box at the [Cumulocity Partner Devices Ecosystem](#)).
7. Click **Save** to update the gateway with the new settings.

To add child devices

1. To start the communication between the gateway and the Profibus device, click **Add Profibus device**.
2. Enter a name for the new device.
3. Select the device protocol from the dropdown field. See [Configuring fieldbus device types](#) for information on how to add a new device protocol.
4. Click **Add** to confirm and notify the gateway.

Now a child device will be created containing the data configured in the selected device protocol.

Cumulocity will notify the gateway to send data for the newly created child device.

MANAGING FIELDBUS DEVICES

Once connected, you can now manage your device. Switch to the **Child devices** tab of a device to list the connected fieldbus devices and navigate to a fieldbus device.

Depending on the capabilities of the device and its configuration in Cumulocity, you can:

- [Collect measurements](#)
- [Send alarms on coil or register changes](#)
- [Log coil and register changes as events](#)
- [Monitor the status of coils and registers](#)

COLLECTING MEASUREMENTS

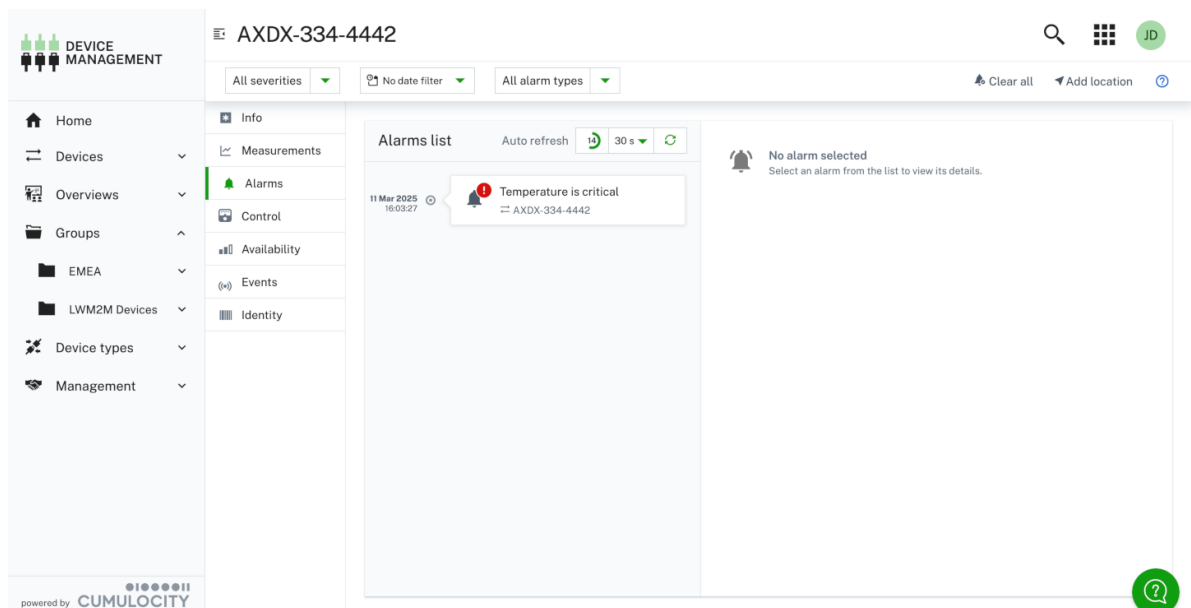
If the device protocol of the fieldbus device is configured to collect measurements, these will be visible in the **Measurements** tab. They will also be available for usage in the [Data explorer](#) and in [dashboards](#).

Data is collected according to the interval specified in the “transmit rate” property of the terminal as described above. To optimize the data traffic, data which is exactly the same as collected previously may not be sent again.



MONITORING ALARMS

If the device protocol of the fieldbus device is configured to send alarms, these will be visible in the **Alarms** tab and usable in widgets. To determine the alarm status, the fieldbus devices are monitored for changes according to the “polling rate” setting of the terminal. If a particular coil or register is non-zero, an alarm will be raised. If the value goes back to zero, the alarm will be cleared.



LOGGING EVENTS

Similar to alarms, changes in fieldbus devices can be monitored and logged as events. Each time, the value of the monitored coil or register changes, an event is created. You can see the events in the **Events** tab of the device or use them in widgets. You can inspect the new value of the monitored coil or register by clicking on the event and unfolding the event details.

DEVICE MANAGEMENT

AXDX-334-4442

Devices > All devices > AXDX-334-4442 > Events

Date from Date to Event type Apply

Realtime Reload Add location

Home Devices Registration All devices Map Simulators Availability Overviews Groups EMEA LWM2M Devices Device types Management

Info Measurements Alarms Control Availability Events Identity

Date	Event	Source
11 Mar 2025, 13:03:27	Switch open event	AXDX-334-4442
11 Mar 2025, 13:03:27	Switch open event	AXDX-334-4442
11 Mar 2025, 13:03:27	Switch open event	AXDX-334-4442
11 Mar 2025, 13:03:27	Switch open event	AXDX-334-4442
11 Mar 2025, 13:03:27	Switch open event	AXDX-334-4442
10 Mar 2025, 13:03:27	Switch open event	AXDX-334-4442

powered by CUMULOCITY

MONITORING THE DEVICE STATUS

The status of devices can be monitored in realtime using dashboard widgets in the Cockpit application. Navigate to the Cockpit application, create a dashboard or report, and add widgets as described in [Cockpit](#).

MONITORING THE DEVICE STATUS USING THE FIELDBUS DEVICE WIDGET

The “Fieldbus device” widget provides you with a tabular display of the status of a device. The status of the device can also be modified through the widget.

To use the “Fieldbus device” widget, follow these steps:

1. Select a dashboard and click **Add widget** in the top menu bar.
2. Select the “Fieldbus device” widget and edit the title of the widget.
3. Select the device that should be shown in the widget in the **Asset selection** section.
4. Select the coils and registers to be shown on the widget.

COCKPIT

Home Groups Alarms Data explorer Reports Configuration

Home

Add widget

Select widget Configuration Appearance

Fieldbus device widget
Allows for seeing status and operating Modbus device

Title: Fieldbus device widget

Asset selection
Search for groups or assets...
SELECTED: ABC-DE-KKL44
Groups > Unassigned devices > MQTT Dev...
MQTT Device tedge_78f172b1...
Filter this column...

ABC-DE-KKL44
AXDX-334-4442

Power Switch Power Switch Coil (discrete output)

sensors

Show	Label	Name	Type
<input type="checkbox"/>	Leak sensor	Leak sensor	Coil (discrete output)

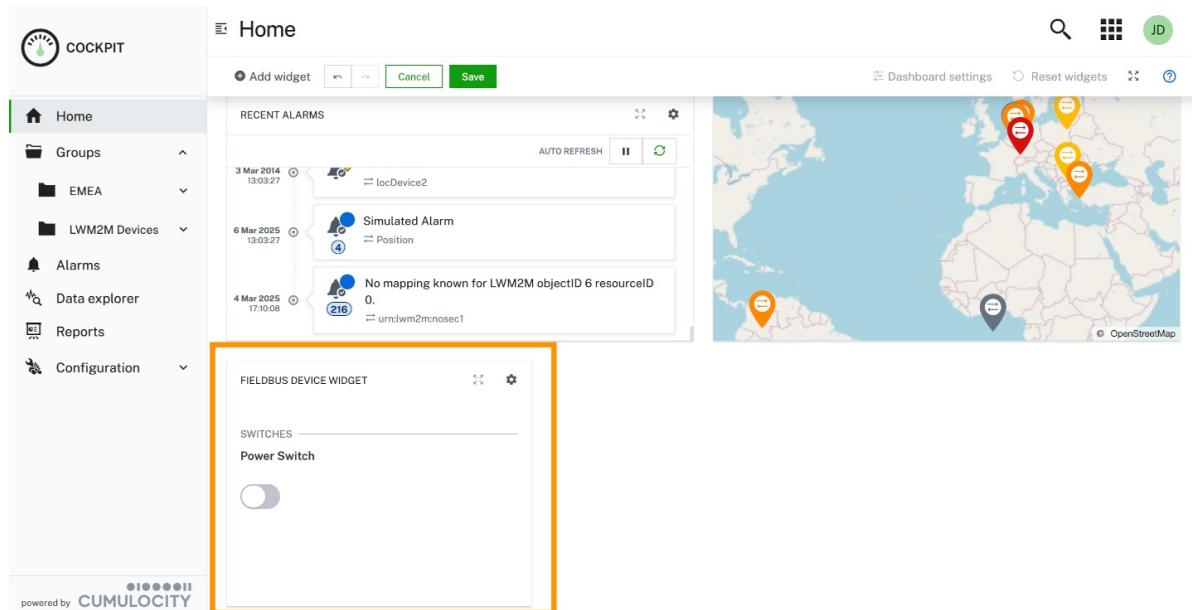
Cancel Save

Sigfox device provisioning has failed with the error

powered by CUMULOCITY

In the widget, the selected coils and registers are grouped into display categories as configured in the device protocol. The “Fieldbus

device" widget updates automatically as soon as there is new data available. You do not need to click **Reload**.



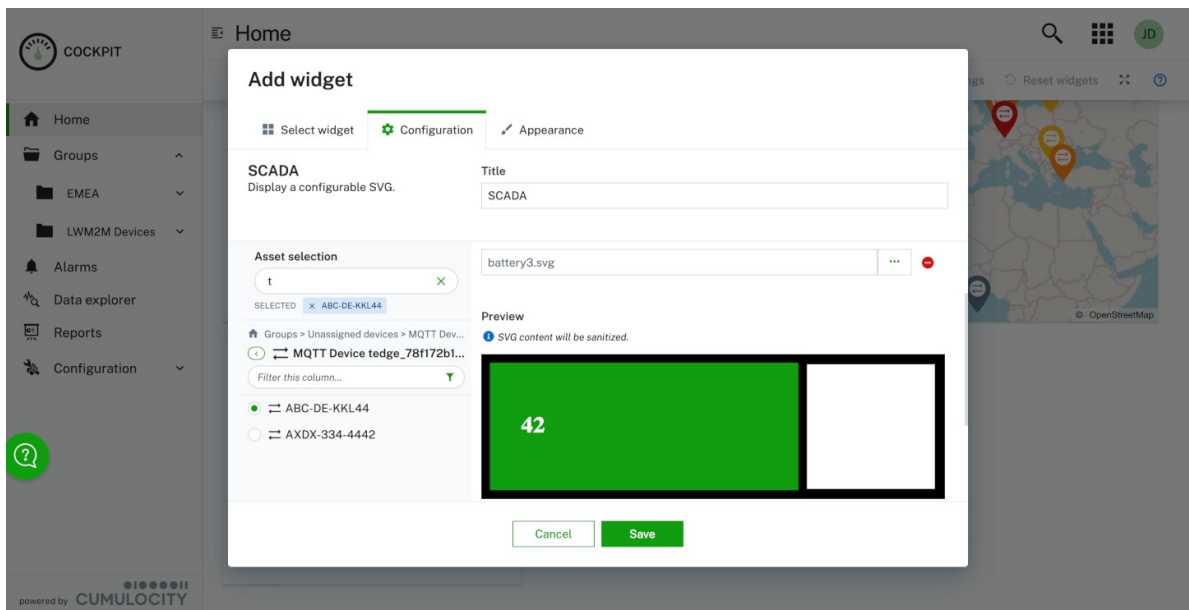
Registers and coils that can be changed are represented by active widgets. If you click a switch, an operation to change the corresponding coil or register is sent to the terminal. Similar, if you change a value and click **Set**, an operation is created. The terminal will then carry out the configuration change on the device, as requested through the operation. While the operation is being processed, a progress indicator is shown.

MONITORING THE DEVICE STATUS USING THE SCADA WIDGET

The "SCADA" widget provides you with a graphic representation of the status of a device.

To use the "SCADA" widget, follow these steps:

1. Select a dashboard and click **Add widget** in the top menu bar.
2. Select the "SCADA" widget and edit the title of the widget.
3. Select the device that should be shown in the widget in the **Asset selection** section.
4. Upload an SVG file with the graphic representation of the device. SVG files are vector graphics that must be specifically prepared with placeholders for the status information. See [Preparing SVG files for the SCADA widget](#) below.
5. Assign placeholders to devices. Note that multiple devices can be taken as source.
6. You now must assign each placeholder to a property of the device. Hover over each placeholder and select **Assign device property** or **Assign fieldbus property**. In the upcoming dialog box, basic device properties or fieldbus properties (that is, status coils and registers) can be selected. Select the desired property and click **Select**.
7. After assigning all placeholders, a preview of the widget with the current values of the properties is shown. Click **Save** to place the widget on the dashboard.



PREPARING SVG FILES FOR THE SCADA WIDGET

The SCADA widget accepts SVG files which use AngularJS directives, for example `ng-if`, `ng-show`, `ng-style`, `ng-repeat`, `ng-click`, for dynamic data presentation.

Moreover, JavaScript event attributes (like `onclick`, `onmouseover`) can be used in SVG files uploaded to SCADA widgets.

Data from devices (like latest measurements and other properties) are provided via placeholders. There are also predefined helper functions which can be used.

For creating SVG files, it is recommended to use <https://boxy-svg.com/>. It is an easy to use, quality Chrome extension.

Placeholders

For a placeholder to be recognized by the SCADA widget, it must occur at least once in double curly braces with no other expression, for example `{{placeholderName}}` (in a comment, attribute's value, or element's content - see example). Once annotated, the placeholder can be used within other expressions, for example `{{placeholderName * 3.1415}}`, `ng-class="{ active: placeholderName > 100 }"` or `ng-if="placeholderName === 'VALUE'"`.

Predefined functions

The following predefined functions are available for use in expressions:

- `goToGroupDetails(groupId)` – takes the group ID and redirects the user to the group details view, for example, `<... ng-click="goToGroupDetails(groupId)">`,
- `goToDeviceDetails(deviceId)` – takes the device ID and redirects the user to the device details view, for example, `<... ng-click="goToDeviceDetails(deviceId)">`,
- `getActiveAlarmsStatusClass(alarmsStatus)` – takes the alarm status object and returns a CSS class that can be used for styling: `none`, `warning`, `minor`, `major`, `critical`, for example, `<... ng-class="getActiveAlarmsStatusClass(alarmsStatus)">`.

Example

```

svg
<?xml version="1.0" encoding="utf-8"?>
<svg width="600px" height="600px" viewBox="0 0 600 600" xmlns="http://www.w3.org/2000/svg">
  <!-- Annotate placeholders in comments: -->
  <!-- {{batteryValue}} -->
  <!-- {{alarmsStatus}} -->

  <style>
    .critical {
      fill: red;
    }
  </style>

  <!-- or in an attribute: -->
  <text data-placeholder="{{batteryValue}}"
    class="text"
    x="50"
    y="200"
    width="200">
    <!-- pass placeholder's value to a predefined function to get alarms status CSS class: -->
    <span ng-class="getActiveAlarmsStatusClass(alarmsStatus)" style="font-size: 45pt;">
      <!-- or in an element's content: -->
      {{batteryValue}}

      <!-- a placeholder can be also a part of expression, for example: -->
      {{batteryValue * 100}} %
    </span>
  </text>
</svg>

```

CONFIGURING FIELDBUS DEVICE PROTOCOLS

New fieldbus device protocols can be created in the **Device protocols** page which is opened from the **Device types** menu in the navigator.

1. Click **Add device protocol** in the top menu bar.
2. Select the protocol of your device from the list.
3. Enter a name for your device and an optional description.
4. Click **Create** to create the protocol.

The device protocol will be added to the device protocol list.

<div> DEVICE MANAGEMENT </div> <ul style="list-style-type: none"> Home Devices Overviews Groups Device types SmartREST templates Device protocols LWM2M post-operati... Management <div> powered by CUMULOCITY </div>	<div> Device protocols Device types > Device protocols </div> <div> Filter... Add device protocol Import Reload ? </div>			
	Device protocol	Description	Type	Resources
	3 Device		LWM2M	23
	32920 PILZ Controller		LWM2M	10
	3303 Temperature		LWM2M	12
	5 Firmware Update		LWM2M	13
	7711 Cumulocity Validation		LWM2M	10
	7722 Cumulocity Binary Sample		LWM2M	1
	Modbus Protocol 1	A demo modbus protocol	Modbus	1
	Modbus Protocol X	The protocol for a demo	Modbus	4
	My Profibus Protocol		Profibus	0
	sigfox protocol		Sigfox	1

Next, you can configure the device protocol, following the descriptions for the respective protocol type below.

If you edit a device protocol that is currently in use, you may need to

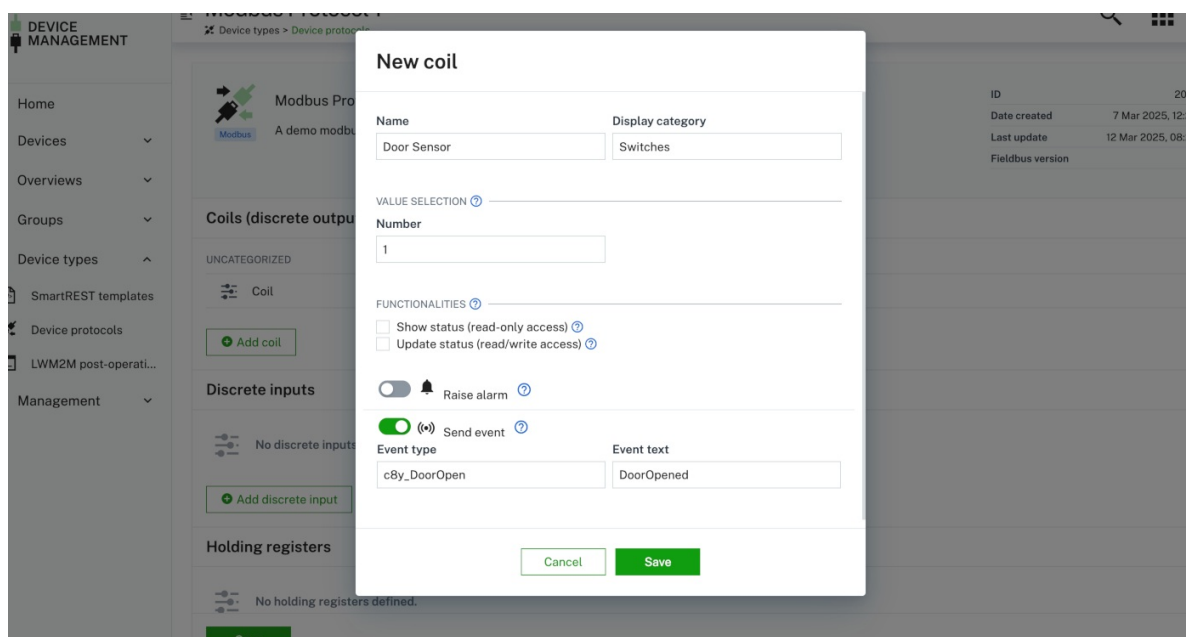
- restart the terminals that use the device protocol,
- reconfigure dashboards and widgets that use the device protocol.

CONFIGURING MODBUS DEVICE PROTOCOLS

To add a coil definition (discrete outputs)

Click **Add Coil** in the **Coils (discrete output)** section, to add a coil definition.

1. Enter the name of the coil as being displayed in the user interface.
2. Optionally, enter the display category to structure your data in widgets.
3. In the **Value selection** section, enter the number of the coil in the Modbus device.
4. In the **Functionalities** section, you may select the following actions:
 - **Show status** - To show the current value in the UI, for example, in the "Fieldbus device" widget. In this case, you can enter the text that the UI should show for unset and set coils.
 - **Update status** - To enable to update the current value from the UI, for example, in the "Fieldbus device" widget.
 - **Raise alarm** - To raise an alarm when the coil is set in the device. In this case, you can specify the type of the alarm that is raised, its text and its severity. Note that there can only be one alarm active of a particular type for a particular device.
 - **Send event** - To send an event each time the value changes. If selected, you may specify the type of event and the text in the event.
5. Click **Save** to save your configuration.



To add a discrete inputs definition

The same settings can be specified for discrete inputs. However, it is not possible to update the status of a discrete input.

To add a register definition

Click **Add holding register** under **Holding registers** or **Add input register** under **Input registers** to add a register definition.

1. In the **General** section, specify a name for the register and a display category to structure your data in widgets.
2. In the **Value selection** section, enter the number of the register in the Modbus device. You can indicate a subset of bits to be used from a register by providing a start bit and a number of bits. This allows you to split a physical Modbus register into a set of "logical registers". It is important to note that in Cumulocity registers are numbered as per the standard Modbus specification, that is they start from 1. This differs with some device manufacturers which count registers starting from 0.
3. In the **Normalization** section, specify how the raw value should be transformed before storing it in the platform. To scale the integer value read from the Modbus device, you can enter a **Multiplier**, a **Divisor** and a number of decimal places in the **Right Shift** field. The register value is first multiplied by the multiplier, then divided by the divisor and then shifted by the number of decimal places. Note, that the terminal may use integer arithmetic to calculate values sent to Cumulocity. For example, if you use a divisor of one and one decimal place, a value of 231 read from the terminal will be sent as 23.1 to Cumulocity. If you use a divisor of ten and no decimal places, the terminal may send 23 to Cumulocity (depending on its implementation). In the **Unit** field, indicate the unit of the data, for example, "C" for temperature values.
4. In the **Options** section, select the following options:
 - **Signed** - If the register value should be interpreted as signed number.
 - **Enumeration type** - If the register value should be interpreted as enumeration of discrete values. If **Enumeration type** is selected, you can click **Add value** to add mappings from a discrete value to a text to be shown for this value in the widget. Click **Remove value** to remove the mapping.
 - **Little endian** - If the register value should be interpreted in little-endian format based on 8-bit values.
5. In the **Functionalities** section, you may select the following actions:
 - **Show status** - To show the current value in the UI, for example, in the "Fieldbus device" widget.
 - **Update status** - To enable to update the current value from the UI, for example, in the "Fieldbus device" widget. If **Update status** is selected, two additional fields **Minimum** and **Maximum** appear. Using these fields, you can constrain numerical values entered in the widget.
 - **Send measurement** - To collect the values of the register regularly according to the transmit interval (see [above](#)). In this case, add a measurement type and a series to be used. For each measurement type, a chart is created in the **Measurements** tab. For each series, a graph is created in the chart. The unit is used for labelling the measurement in the chart and in the "Fieldbus device" widget.
 - **Raise alarm** - To raise an alarm when the register is not zero in the device measurement. In this case, you can specify the type of the alarm raised, its text and its severity. Note, that there can only be one alarm active of a particular type for a particular device.
 - **Send event** - To send an event each time the value of the register changes. If selected, you may specify the type of event and the text in the event.

6. Click **Save** to save the register.

In the **Options** section, select the checkbox **Use server time** to create the time stamps for data on the server instead of on the terminal. If you must support buffering of data on the terminal, leave this checkbox clear.

Finally, click **Save** to save the device protocol.

CONFIGURING CAN BUS DEVICE PROTOCOLS

CAN bus device protocols can be configured in a very similar way as Modbus device protocols. For more information, see [Configuring Modbus device protocols](#) above. The differences are:

- Holding registers are used to describe the different pieces of data inside CAN messages.
- Enter the CAN message ID of the specific message the data should be extracted from. Use a hexadecimal number for the message ID.
- Conversion of values is extended by an offset parameter (for example, any positive or negative number). This is added or subtracted from the register value, depending on its sign. The offset calculation is done after applying multiplier and divisor, and before performing decimal shifting.

CONFIGURING PROFIBUS DEVICE PROTOCOLS

Profibus device protocols can be configured in the following way:

1. In the **Registers** section, click **Add register** to add one or more register definitions as described exemplarily for Modbus devices in [To add a register definition](#) above.
2. In the **Options** section, select the checkbox **Use server time** to create the time stamps for data on the server instead of on the terminal. If you must support buffering of data on the terminal, leave this checkbox clear.
3. Finally, click **Save** to save your settings.

CONFIGURING CANOPEN DEVICE PROTOCOLS

CANopen device protocols can be configured in the following way:

In the **CANopen device type** field, specify the device type as a hex number.

In the **Variables** section, you determine the CANopen variables. Variables inside the “Object Dictionary”(OD) of the CANopen device can later be accessed by adding the variables to the device type definition.

Click **Add variable** to configure a new variable.

New variable

GENERAL

Name: Temperature Display category: Sensors

VALUE SELECTION

Index: 200 Sub-index: 3E Data type: Signed (8-bits)

Access type: Read-write

VALUE NORMALISATION

Unit: e.g. u

FUNCTIONALITIES

☒ Show status (read-only access)

Cancel Save

To configure a variable

1. In the **General** section, specify a name for the variable and a display category. Display categories are used to group variables into sections in the visualization.
2. In the **Value selection** section, specify from where the value should be extracted:
 - **Index** - Index of the variable in the OD of the device.
 - **Sub-index** - Sub-index of the variable in the OD of the device.
 - **Data type** - Type of the variable (for example boolean, unsigned).
 - **Access type** - Access type, for example, read-only, write-only.
3. Depending on the selected access type, the following functionalities may be specified:
 - **Show status** - To enable to show the current value in the UI, for example, in the "Fieldbus device" widget.
 - **Update status** - To enable to update the current value from the UI, for example, in the "Fieldbus device" widget. If selected, two additional fields **Minimum** and **Maximum** are displayed. Using these fields, you can constrain numerical values entered in the widget.
 - **Send measurement** - To create a measurement whenever the value is changed. If selected, you may specify a **Measurement type** and **Measurement series**.
 - **Raise alarm** - To raise an alarm if a given mask matches with the value of the variable ((value & mask) == mask). Additionally, you may specify the type of the alarm raised, its text and its severity.
 - **Send event** - To send an event each time the value of the register changes. If selected, you may specify the type of event and the text in the event.
4. In the **Normalization** section, specify a unit to define how the raw value should be transformed before storing it in the platform.
5. Click **Save** to save the variable.

The variable will be listed in the **Variables** section of the device protocol. All variables are grouped by the given display category, that means, variables with the same category are grouped together.

My CANOpen Protocol

Device types > Device protocols

My CANOpen Protocol

e.g. My protocol description

CANOpen device type e.g. 39FC23

ID	109246
Date created	12 Mar 2025, 15:47:51
Last update	12 Mar 2025, 15:50:47
Fieldbus version	4

Variables

SENSORS

Temperature	DATA TYPE: Signed (8-bits)	ACCESS TYPE: Read-write	INDEX: 200	SUB-INDEX: 3E
-------------	----------------------------	-------------------------	------------	---------------

[Add variable](#)

Options

☐ Use server time

[Save](#)

powered by CUMULOCITY

After completing your configuration, click **Save** to save the device protocol configuration.

Importing a CANOpen device protocol

See [Exporting and importing device protocols](#) for general information on how to import a device protocol.

After importing the EDS file, all variables defined in the file are listed in the **Variables** section of the CANOpen device protocol.

The user can then enrich the imported variable configurations manually, for example by adding the missing display category.

Configuring CANOpen device data

To configure CANOpen device data navigate to the desired device and switch to the **CANOpen** tab.

In the **CANOpen communication** section, the following parameters can be configured:

- **Baud rate:** This field must match with the used baud rate in the CANOpen network.
- **Polling rate:** The rate at which the agent sends requests to the CANOpen devices.
- **Transmit rate:** The transfer rate, that is, the rate at which the terminal sends regular measurements to Cumulocity.

In the **CANOpen** section, up to 127 CANOpen devices can be added to the gateway as child devices by providing the following parameters:


- **Name:** The name of the device shown in the UI.
- **Device type:** The device type of the CANOpen device. The user can select from a list of all CANOpen device types which are stored in the device database.
- **Node ID:** The CANOpen node ID of the device. It is used for addressing the device inside the CANOpen network.

INFO

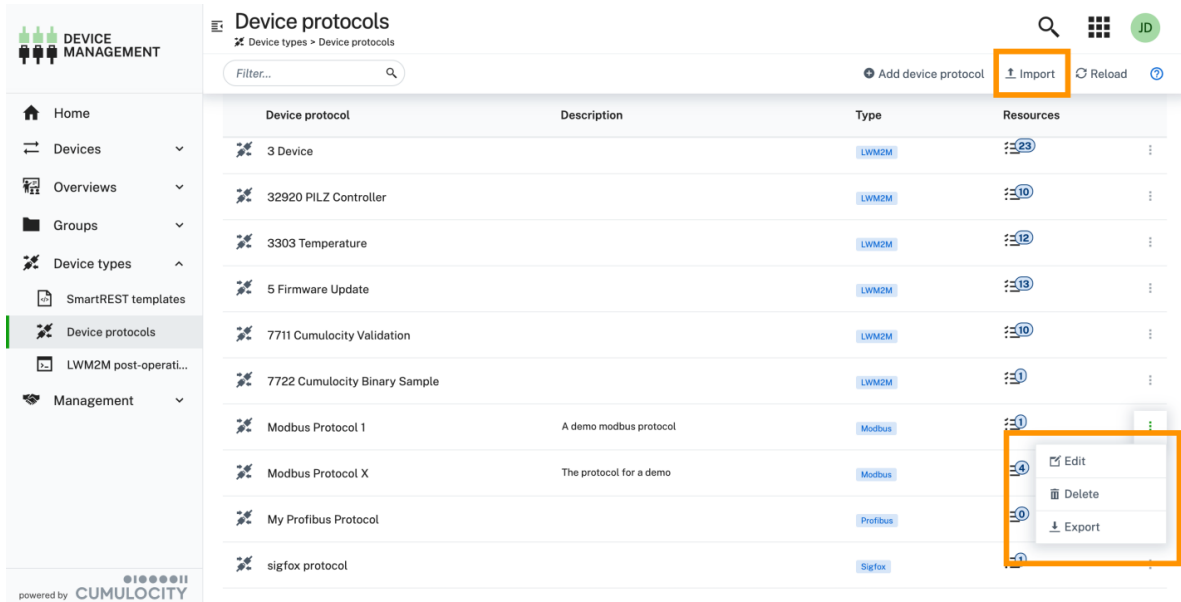
The device type and node ID must match with the real CANOpen device, otherwise setting up the communication is not possible or wrong values will be transmitted.

EXPORTING AND IMPORTING DEVICE PROTOCOLS

To manage device protocols more conveniently, you can export them to a file. The file can be re-imported to set up other Cumulocity accounts easily or to restore the protocols from a backup. The import functionality also supports importing ready-made device protocols provided by device manufacturers.

To export a device protocol, click the menu icon  at the right of the respective row and click **Export**.

A file with the device protocol definition will be downloaded, named "<device type>.json".



The screenshot shows the 'Device protocols' page in the Cumulocity Device Management interface. The left sidebar contains navigation links: Home, Devices, Overviews, Groups, Device types, SmartREST templates, Device protocols (selected), LWM2M post-operati..., and Management. The main content area displays a table of device protocols with columns: Device protocol, Description, Type, and Resources. The table lists several protocols, including '3 Device', '32920 PILZ Controller', '3303 Temperature', '5 Firmware Update', '7711 Cumulocity Validation', '7722 Cumulocity Binary Sample', 'Modbus Protocol 1', 'Modbus Protocol X', 'My Profibus Protocol', and 'sigfox protocol'. The 'Import' button in the top right corner is highlighted with an orange box. A context menu for the 'Modbus Protocol 1' row is also highlighted with an orange box, showing options: Edit, Delete, and Export.

Device protocol	Description	Type	Resources
3 Device		LWM2M	23
32920 PILZ Controller		LWM2M	10
3303 Temperature		LWM2M	12
5 Firmware Update		LWM2M	13
7711 Cumulocity Validation		LWM2M	10
7722 Cumulocity Binary Sample		LWM2M	1
Modbus Protocol 1	A demo modbus protocol	Modbus	1
Modbus Protocol X	The protocol for a demo	Modbus	4
My Profibus Protocol		Profibus	0
sigfox protocol		Sigfox	1

1. To import a device protocol, click **Import** in the top menu bar.
2. In the resulting dialog box, either select a pre-defined protocol or upload a file with a previously exported device protocol.
3. You may enter a new name for the device protocol.
4. Click **Import** to import the protocol.

LWM2M

INTRODUCTION

Lightweight M2M (LWM2M) is a traffic and resource-optimized protocol to remotely manage IoT devices. The protocol is standardized by the Open Mobile Alliance. For more information, see <https://www.openmobilealliance.org/lwm2m/>.

! IMPORTANT

Cumulocity currently supports LWM2M 1.1 over CoAP and UDP.

i INFO

Cumulocity LWM2M 1.1 introduces a composite operation and a client-side “send” operation. If a sensor or a number of sensors want to send measurements to the server, they can use the “send” operation to send single or composite measurement data. The LWM2M 1.1 “send” operation makes this simple on protocol level. Cumulocity also allows a timestamp-based operation. If a sensor reports a timestamp resource object in parallel with its data from the same object, the timestamp will be used.

Cumulocity LWM2M 1.1 is backward compatible. What has worked on Cumulocity LWM2M 1.0 will also work on Cumulocity LWM2M 1.1. However, Cumulocity LWM2M 1.1 introduces some new features which are not compatible with LWM2M 1.0 devices. If you try to run some of these features with LWM2M 1.0 devices, you may receive an error message response.

You can connect any device supporting LWM2M 1.1 or LWM2M 1.0 to Cumulocity without programming. Cumulocity expects the device and its capabilities (such as firmware update) to be compliant to the LWM2M specification. The device must support the UDP binding of the LWM2M standard.

✓ REQUIREMENTS

In order to use LWM2M, you must be subscribed to the LWM2M-agent application. If the LWM2M-agent is not available in your tenant please contact [product support](#).

i INFO

The LWM2M agent supports DTLS Connection ID. Contact your platform administrator if you use LWM2M devices that support connection IDs and if you are unsure if this feature is enabled.

Our LWM2M solution allows any LWM2M object to be easily interfaced with the platform. For the sake of convenience, we provide out-of-the-box integration for the following LWM2M objects:

- Device (/3)
- Connectivity monitoring (/4)
- Firmware update (/5)
- Location (/6)

Our LWM2M solution supports the following measurement types:

- Boolean - represented as "true" or "false" and mapped in Cumulocity LWM2M solution respectively as "0" or "1"
- Float - represented by any float numeric values and mapped in Cumulocity LWM2M solution as is
- Integer - represented by any integer numeric values and mapped in Cumulocity LWM2M solution as is
- Time - represented by any integer or a date format that can be converted to an integer and also mapped in the Cumulocity LWM2M solution as an integer
- String - represented by:
 - Any numeric values in a string format (for example integer and float)
 - Numeric values in scientific format in a positive or negative exponential notation (for example 1.23E10 or 3.57e+5 or 9.8e-4)
 - Any positive or negative numeric values starting with leading zero will be interpreted as a positive or negative octal value (for example -029 or 010) and will be stored in Cumulocity as its decimal representation
 - Any positive or negative value starting with 0x or 0X followed by any number or letter from A to F (case insensitive) will be interpreted as a positive or negative hexadecimal value (for example 0x23F3D5C1 or -0x42a3b3d1) and will be stored in Cumulocity as its decimal representation

! IMPORTANT

If a string is mapped into a measurement and the string value does not follow any of the notations above, it cannot be parsed. As a result, an alarm will be created.

To use these integrations, upload the corresponding DDF XML to your tenant. For arbitrary protocols, you can configure how LWM2M devices are mapped to Cumulocity using device protocols. See [Configuring fieldbus device protocols](#) for more information.

3304 Humidity

Device types > Device protocols

Humidity

This IPSO object should be used with a humidity sensor to report a humidity measurement. It also provides resources for minimum/maximum measured values and the minimum/maximum range that can be measured by the humidity

ID: 3304
Date created: 3/4/25, 1:19 PM
Last update: 3/4/25, 1:19 PM

Resources

Name	Functionalities	Ignore empty values	Validation rules
5518 Timestamp	Timestamp	<input checked="" type="checkbox"/>	
5601 Min Measured Value	--	<input type="checkbox"/>	
5602 Max Measured Value	--	<input type="checkbox"/>	
5603 Min Range Value	--	<input type="checkbox"/>	
5604 Max Range Value	--	<input type="checkbox"/>	
5605 Reset Min and Max Measured Values	--	<input type="checkbox"/>	
5700 Sensor Value	--	<input type="checkbox"/>	
5701 Sensor Units	--	<input type="checkbox"/>	
5750 Application Type	--	<input type="checkbox"/>	
6042 Measurement Quality Indicator	--	<input type="checkbox"/>	

powered by CUMULOCITY

REGISTERING LWM2M DEVICES

The data provided during registration must enable LWM2M communication and holds information for factory bootstrap and client-initiated bootstrap. In the factory bootstrap mode, the LWM2M client has been configured with the necessary bootstrap information prior to the deployment of the device. The client-initiated bootstrap mode requires a LWM2M bootstrap server account preloaded in the LWM2M client. Cumulocity supports registration for **unsecured** and **secured** LWM2M devices allowing connection with NO_SEC, PSK and X.509 security modes respectively.

You can register a LWM2M device in Cumulocity in two ways:

- [Single device registration](#)
- [Bulk device registration](#)

SINGLE DEVICE REGISTRATION

To register a LWM2M device in Cumulocity navigate to **Devices > Registration** in the Device Management application, click **Register device** at the top right and select **Single device registration > LWM2M** from the dropdown.

The LWM2M device registration wizard has four steps:

- **Device** - configuration of globally unique device identifier. Refer to [Device settings](#) section below for details about the fields.
- **Security** - configuration of LWM2M security modes, separate for bootstrap server ("Bootstrap Security Mode") and regular server ("Server Security Mode") connections. Refer to [Security settings](#) section below for details about the fields.
- **Bootstrap** settings for enabling the device to connect to the Cumulocity LWM2M bootstrap server. Refer to [Bootstrap settings](#) section below for details about the fields.
- **Advanced** setting for further optional configurations. Refer to [Advanced settings](#) section below for details about the fields.

After filling all applicable information for the device, click **Register**. The UI will display a confirmation message.

BULK DEVICE REGISTRATION

If there is a number of devices to be registered at the same time, it is more convenient to use bulk device registration.

To register the LWM2M devices in Cumulocity, navigate to **Devices > Registration** in the Device Management application, click **Register device** at the top right and select **Bulk device registration > LWM2M** from the dropdown. Upload a CSV file with the registration data in the resulting bulk registration dialog. The dialog also contains CSV template links. Refer to [Device settings](#), [Security settings](#), [Bootstrap settings](#) and [Advanced settings](#) below for details about the fields.

INFO

The maximum size allowed for the CSV file is 10 MB.

When you upload the CSV file, the dialog will display a confirmation message that tells you where to find the result. The bulk device registration operation will be displayed in the [LWM2M connector device](#) object created for the tenant.

Device settings

The fields below must be contained to be able to establish a connection:

Field	CSV field name	Type	Description	Mandatory
-------	----------------	------	-------------	-----------

Field	CSV field name	Type	Description	Mandatory
Endpoint client ID	endpoint id	String	Indicates the LWM2M client's endpoint ID in order to allow the LWM2M bootstrap to provision the bootstrap information for the LWM2M client. The endpoint ID has to be unique across all tenants and must have the same value as the ID. Registering a device using an endpoint ID already used will result in an error.	Mandatory

Security settings

We can distinguish 2 types of connectivity:

- **bootstrap** - connection established to bootstrap server in order to provision device with server configuration.
- **server** - connection established to regular server to exchange data.

Cumulocity supports different security modes for both connections so device can connect to bootstrap server and regular server using different security modes.

Currently Cumulocity supports 3 security modes:

- **NO_SEC** - Device connects to the platform without any security. It is highly recommended to always protect the LWM2M protocol. However, there are scenarios in which the LWM2M protocol is deployed in environments where the lower layer security mechanisms are provided.
- **PSK** - Device connects to the platform using DTLS with the given pre-shared ID and KEY (PSK).
 - **bootstrap connectivity** - PSK ID and KEY must be preconfigured on the device by manufacturer and the same credentials must be provided during device registration.
 - **server connectivity** - PSK ID and KEY must be provided during device registration. The device is expected to use these credentials during regular server connection. Bootstrap server will write these credentials to the device during a LWM2M bootstrap session. There are 2 ways to provide PSK ID and KEY:
 - **manual (PSK)** - Manually enter both ID and KEY values (useful when device is preconfigured, and it won't use bootstrap server).
 - **generated (PSK_GENERATED)** - Server will assume PSK ID is equal to the endpoint ID of the device, and it will generate random secure PSK KEY. This scenario is useful when you want to secure server connectivity without preconfiguring PSK keys on the device. Device can bootstrap with NO_SEC/X509, and it will be provisioned with auto-generated PSK credentials for server connectivity.
- **X509** - Device connects to the platform using X.509 certificate.
 - **bootstrap connectivity** - device must be preconfigured by manufacturer with X.509 certificate, private key and trust store/server certificate. There are no additional settings to provide during device registration.
 - **server connectivity** - here we can distinguish 2 scenarios:
 - **device has all X.509 credentials pre-configured by manufacturer** - no additional data is required during device registration.
 - **device expects bootstrap server to provision certificates** - in this case it's possible (but not mandatory, when empty it won't be written to device by bootstrap server) to provide:
 - X.509 Certificate and Private Key in PEM format. Private key will be stored as encrypted data on the platform.
 - Server certificate to use (more on this below).
 - Certificate usage (as defined in LWM2M specification)

More details on X.509 security

Client X.509 certificate must meet the requirements specified in [LWM2M specification](#). For testing purposes, certificate can be generated by a self-signed CA in the following way:

- Creation of self-signed CA:

```
openssl ecparam -name prime256v1 -genkey -noout -out myCA.key
openssl req -x509 -new -nodes -key myCA.key -days 36500 -out myCA.pem
```

- Creation of device certificate signed by our CA:

```
# create and sign certificate
openssl ecparam -name prime256v1 -genkey -noout -out myDevice.key
openssl req -new -key myDevice.key -out myDevice.csr
openssl x509 -req -in myDevice.csr -CA path/to/myCa.pem -CAkey path/to/myCa.key -CAcreateserial -out myDevice.crt -days 36500
# export to PKCS8 PEM format
openssl pkcs8 -topk8 -inform PEM -outform PEM -in myDevice.key -out myDevice.key.pem -nocrypt
# optionally export to DER format if your device needs it
openssl pkcs8 -topk8 -inform PEM -outform DER -in myDevice.key -out myDevice.der -nocrypt
```

Trusting CA in Cumulocity

Before devices are able to connect to the platform, CA that issued device certificates must be added to trusted certificates. See [Managing trusted certificates](#) on how to add and trust CA certificate.

All security field details

The fields below must be contained to configure security modes for bootstrap and regular server connection:

Field	CSV field name	Type	Description	Mandatory
Server security mode	securityMode	String	Determines the type of connection used by the LWM2M device when it connects to the Cumulocity LWM2M server. Possible values are: "NO_SEC", "PSK" and "X509".	Mandatory
Bootstrap security mode	bootstrapSecurityMode	String	Determines the type of connection used by the LWM2M device when it connects to the Cumulocity LWM2M bootstrap server. Possible values are: "NO_SEC", "PSK", "PSK_GENERATED" and "X509".	Mandatory
LWM2M PSK ID	lwm2m psk_id	String	The ID used by the device for server connections in PSK mode. The LWM2M PSK ID has to be unique across all tenants. Registering a device using an LWM2M PSK ID already used will result in an error.	Mandatory for PSK security mode.
LWM2M PSK key	lwm2m psk_key	String	The hex-encoded pre-shared key used by the device for server connections in PSK mode.	Mandatory for PSK security mode.
Bootstrap PSK ID	bootstrap psk_id	String	The ID used by the device for bootstrap connections in PSK mode. The bootstrap PSK ID has to be unique across all tenants. Registering a device using an bootstrap PSK ID already used will result in an error.	Mandatory for PSK security mode.
Bootstrap PSK key	bootstrap psk_key	String	The hex-encoded key used by the device for bootstrap connections in PSK mode.	Mandatory for PSK security mode.
X.509 certificate	x509ClientCertificate	String	X.509 device certificate (in PEM format) written to the device during bootstrap phase. An empty value means that it isn't written at all.	Optional

Field	CSV field name	Type	Description	Mandatory
X.509 private key	x509ClientPrivateKey	String	X.509 device private key (in PEM format) written to the device during bootstrap phase. An empty value means that it isn't written at all.	Optional
X.509 certificate usage	x509CertificateUsage	String	LWM2M Certificate usage written to the device during bootstrap phase. An empty value means that it isn't written at all. One of: <ul style="list-style-type: none"> • CA_CONSTRAINT • SERVICE_CERTIFICATE_CONSTRAINT • TRUST_ANCHOR_ASSERTION • DOMAIN_ISSUER_CERTIFICATE 	Optional
Server certificate	serverPublicKey	String	Name of the server certificate written to the device during bootstrap phase. Server certificates are preconfigured in the Management tenant. An empty value means that it isn't written at all.	Optional

Bootstrap settings

Registration of NO_SEC devices

Unsecured devices connect during bootstrap connection and server connection through unsecured ports:

- **5683**: unsecure bootstrap connection
- **5783**: unsecure direct server connection

Registration of secured devices

Secured devices connect during a bootstrap connection and a server connection through secured ports:

- **5684**: secured bootstrap connection
- **5784**: secured direct server connection

See the table below for the full set of bootstrap fields you can add:

Field	CSV field name	Type	Description	Mandatory
-------	----------------	------	-------------	-----------

Field	CSV field name	Type	Description	Mandatory
LWM2M server URI	lwm2m server uri	String	The URI the server uses for bootstrap. The LWM2M bootstrap server is used to provision the LWM2M client with the information required to contact the LWM2M servers. If you use the Cumulocity service, the hostname of the LWM2M server is "lwm2m.cumulocity.com". The bootstrap ports are "5683" for unsecure bootstrap connections and "5684" for secure bootstrap connections. The LWM2M server ports are "5783" for unsecure server connections and "5784" for secure server connections. Note that these values can be different for other services.	Mandatory for LWM2M bootstrap
LWM2M bootstrap short server ID	bootstrapShortServerId	Integer	The short server ID to be used for the bootstrap server. Default is "0".	Optional
LWM2M short server ID	lwm2mShortServerId	Integer	The short server ID to be used for LWM2M server. Default is "1".	Optional
Security instance offset	securityInstanceOffset	Integer	The first instance to be used during bootstrap to which entries are written. Default is "0". If set, for example, to "3", the first instance will be three.	Optional
Registration lifetime	registrationLifetime	Integer	The registration lifetime that is sent to the device during bootstrap. Overrides global agent configuration. The value must be specified in seconds.	Optional
Default minimum period	defaultMinimumPeriod	Integer	The default minimum period to configure during bootstrap. See the LWM2M specification for explanation.	Optional
Default maximum period	defaultMaximumPeriod	Integer	The default max period to configure during bootstrap. See LWM2M specification for explanation.	Optional

INFO

After creation, you can view and change device parameters in the **LWM2M configuration** tab in the **Device details** page, see [LWM2M configuration](#).

Advanced settings

See the table below for information on additional fields:

Field	CSV field name	Type	Description	Mandatory
LWM2M device type	type	String	Type to set for the device managed object on creation. Default is "c8y_lwm2m".	Optional
Binding mode	bindingMode	String	The LWM2M binding mode to be reported to the device. For LWM2M 1.0 devices the supported modes are "UQ" (default, queuing) and "U" (unqueued). Note that since Cumulocity LWM2M 1.1, the "Q" (queue) mode is not supported, and the default mode will be "U" (unqueued). Cumulocity will always queue operations, regardless of whether the device is connected or not. This means that the setting has no effect on the behavior of Cumulocity.	Optional
Awake time registration parameter	awakeTimeRegistrationParameter	Integer	Specifies a time interval in milliseconds for which a device is awake and accepting network traffic after sending a LWM2M registration or a registration update to Cumulocity. If set to 0, the device will be considered as always online. If the value is not set, the awake time is determined by the LWM2M client's registration awake time attribute "at" or, if this attribute is also not found, then by the global setting that is defined in the LWM2M microservice.	Optional
Notification storing when disabled or offline / notificationIfDisabled	notificationIfDisabled	Boolean	See the LWM2M specification . Allowed values are true or false. Default: Not configured.	Optional, defaults to Leshan default behavior
Disable timeout	disableTimeout	Integer	See the LWM2M specification . Allowed values are integer numbers. Default: Not configured.	Optional, defaults to Leshan default behavior

Field	CSV field name	Type	Description	Mandatory
LWM2M request timeout	lwm2mRequestTimeout	Integer	The timeout used for shell operation requests such as read, write, execute done by the LWM2M microservice to the LWM2M device. The value is in milliseconds and can be given to override the default value that is provided in the LWM2M microservice property file with the "C8Y.lwm2mRequestTimeout" property. The value must stay within the bounds of the minimum and maximum values defined in the LWM2M microservice properties: `C8Y.lwm2mMinRequestTimeout` and `C8Y.lwm2mMaxRequestTimeout`. If the provided value is lower than the minimum, the LWM2M agent will ignore it and set the allowed minimal value. Analogically, the LWM2M agent will correct a value higher than the allowed maximum.	Optional
Automatic setting of required interval	autoManageAvailabilityRequiredInterval	Boolean	This setting is configurable so that unavailability alarms will only be triggered when an LWM2M device is truly unavailable, reducing the number of false alarms. When true (default), the LWM2M service automatically sets the interval to registration lifetime plus 2 minutes. When false, the user can define a required interval using Availability required interval value "availabilityRequiredInterval" property during device registration or on the "Info" tab in the device details after device creation. If value is not provided ("null"), the default behavior of the LWM2M service will be used.	Optional, default = true
Availability required interval value	availabilityRequiredInterval	Integer	If not empty, this value will be used as the initial required interval in the created device.	Optional, default = empty
Binary delivery encoding	binaryDeliveryEncoding	String	Indicates the encoding format for writing binaries to a LWM2M device. The encoding format can be OPAQUE, TLV, JSON or TEXT. In case of empty or invalid entries, the default format is OPAQUE.	Optional

Field	CSV field name	Type	Description	Mandatory
Use source timestamp	enableResourceLevelTimestamp	Boolean	If this device property is enabled Cumulocity uses time stamps reported by the device for constructing measurements, events and alarms. LWM2M offers various methods for associating timestamp information with data points, including resources 5518 and 6050, SenML, or resource 5 for the location object (6). When activated, the LWM2M agent utilizes this timestamp data source to generate measurements, events, or alarms. If deactivated, the LWM2M agent resorts to using its local time. Default: false	Optional, defaults to false
Keep old values in the objects tab if an operation fails	c8y_GenericUI_retainOldValuesIfError	Boolean	Controls if stale values are kept in the Objects tab. If this flag is set to <code>true</code> (default) the agent never removes a value. If set to <code>false</code> the agent will remove values in two cases: <ol style="list-style-type: none"> 1. A failed read or write operation will lead to removal if the device answers with one of the following CoAP response codes: <code>4.01 (Unauthorized)</code>, <code>4.03 (Forbidden)</code>, <code>4.04 (Not found)</code>, <code>5.00 (Internal Server Error)</code> and <code>5.01 (not implemented)</code>. 2. Resources not contained in a discover response will be removed. This allows a discover operation to be used for purging resources that do not exist on the device any longer. 	Optional
Serialization format	serializationFormat	String	Indicates the preferred content format for Cumulocity to communicate with the device. The allowed content formats are: TLV, JSON, CBOR, TEXT, OPAQUE or SENML_JSON and SENML_CBOR. In case of an empty or invalid entry, Cumulocity automatically selects the serialization format which the device sends during device registration.	Optional
Disable the default behavior for object instances	disableInternalObjectInstanceActions	Boolean	Cumulocity implements default handlers for objects 3,4 and 6. For example, they update the device name upon the reception of the corresponding resource in object 3 or update the device location. This flag allows those behaviors to be turned off.	Optional

Field	CSV field name	Type	Description	Mandatory
Event log level	logLevel	String	Cumulocity can output detailed logs to the event stream. This field configures a log level. Allowed values are: NONE (nothing will be logged as events), LIFECYCLE (only registration, de-registration and registration update events), VERBOSE (LIFECYCLE + sent/received data + detailed firmware update information). Default is LIFECYCLE.	Optional

The following table explains several optional parameters related to firmware updates which help in tuning the Firmware Over The Air (FOTA) parameters on a device level.

Field	CSV field name	Type	Description	Mandatory
Firmware update delivery method	fwUpdateDeliveryMethod	String	Explains the firmware update delivery method. Allowed values are PUSH, PULL or BOTH.	Optional
Firmware update supported device protocol	fwUpdateSupportedDeviceProtocol	String	Indicates the device protocol to be used for the firmware update. Allowed values are COAP, COAPS, HTTP or HTTPS.	Optional
Firmware update reset mechanism	fwUpdateResetMechanism	String	Indicates the mechanism used to reset the firmware update state machine. Allowed values are PACKAGE or PACKAGE_URI. Depending on the value, the LWM2M agent either writes an empty string to package URI resource or sets the package resource to NULL (''). If this field is not used, the default reset state machine mechanism is used where a reset is done via package resource for PUSH and via package URI for PULL.	Optional
Initial State Machine Reset	fwResetStateMachineOnStart	Boolean	Controls if the LWM2M agent performs an initial state machine reset before it starts a firmware update. Default is TRUE.	Optional
Firmware update URL (DEPRECATED)	fwUpdateURL	String	DEPRECATED: Use the regular firmware repository to specify the firmware version that links to an external URL. This field will be removed in a future update. Indicates the firmware update URL from where the LWM2M device can download the firmware package.	Optional
Disable automated firmware update support	disableFirmwareStateMachine	Boolean	Indicates if the default firmware update state machine should be disabled. Default is false.	Optional

i INFO

Firmware updates are also supported for the registration of unsecured devices as well as secured devices. For more information, see [Managing firmware](#).

Registering LWM2M devices using the REST API

LWM2M internally uses our [Extensible device registration](#) feature. It provides an API based on JSON Schema and REST to extend Cumulocity with arbitrary wizards for device registration.

REST-based single LWM2M device registration

Before the actual registration of a LWM2M device, it first is important to understand the set of available device properties. This set can be obtained using the `metadata` endpoint of LWM2M:

```
GET /service/lwm2m-agent/deviceRegistration/metadata
```

The registration of a new device then can be accomplished by posting a set of these values to the corresponding registration endpoint:

```
POST /service/lwm2m-agent/deviceRegistration/
```

Example request payload:

```
{
  "bootstrapSecurity": {
    "bootstrapSecurityMode": "PSK",
    "bootstrapId": "98ABCD32",
    "bootstrapKey": "AABB3104D212"
  },
  "serverSecurity": {
    "securityMode": "X.509"
  },
  "bootstrapShortServerId": 0,
  "lwm2mShortServerId": 1,
  "securityInstanceOffset": 0,
  "bindingMode": "UQ",
  "enableResourceLevelTimestamp": false,
  "genericUIRetainOldValuesIfError": true,
  "binaryDeliveryEncoding": "OPAQUE",
  "disableObjectInstanceActions": false,
  "disableFirmwareStateMachine": false,
  "stateMachineResetBeforeFirmwareUpdate": true,
  "endpointId": "urn:my:example:device",
  "lwm2mServerUri": "coaps://lwm2m.cumulocity.com:5784",
  "registrationLifetime": 12000
}
```

REST-based bulk Registration for LWM2M Devices

Alternatively, LWM2M devices can be registered in bulk using the API by posting a CSV file to the LWM2M service. The API endpoint and request format are as follows:

```
POST /service/lwm2m-agent/deviceRegistration/bulk`
Content-Type: multipart/form-data; boundary=boundary

--boundary
Content-Disposition: form-data; name="file"; filename="<input csv file>"
Content-Type: text/csv

--boundary--
```

For more details on the CSV format being used, please refer to the section on [bulk device registration](#).

DUPLICATE LWM2M DEVICES

If a LWM2M device has been registered with the same endpoint ID before, the device registration will not register the device, neither for single nor for bulk device registrations. For single device registrations, the duplication error message will be displayed after clicking register. For bulk device registrations, the information about duplicate LWM2M devices will be displayed under the [LWM2M connector device](#)'s bulk upload operation result.

DEVICE DELETION

To remove a LWM2M device, delete it through the [All devices](#) list in the Device Management application or via the **Device status** card on the **Info tab** in the device details.

Alternatively, you can delete a LWM2M device using a REST call. With the managed object ID (device ID) of the device to be deleted, this can be accomplished using the following DELETE request.

Rest-based single LWM2M device deletion

```
DELETE /service/lwm2m-agent/deviceRegistration/{device ID}
```

! IMPORTANT

It is not recommended to use the inventory API for directly deleting LWM2M devices. This action may result in issues when attempting to register a device with the same endpoint name at a later time.

Rest-based bulk LWM2M device deletion

Multiple LWM2M devices can be deleted in bulk by posting a CSV file to the LWM2M REST API.

```
DELETE /service/lwm2m-agent/deviceRegistration/bulk`
Content-Type: multipart/form-data; boundary=boundary

--boundary
Content-Disposition: form-data; name="file"; filename="<input csv file>"
Content-Type: text/csv

--boundary--
```

This endpoint uses the same CSV format which is also used to [register](#) LWM2M devices in bulk.

LWM2M CONNECTOR DEVICE

The LWM2M connector device is an automatically generated device for the tenants which have a subscription to the LWM2M application. You can use this device to manage tenant-wide LWM2M devices. The [help](#) shell command shows the available operations and how to use them.

Additionally, the bulk device registration status and result are shown under this device.

The screenshot displays the 'LWM2M cucumber-java-tenant-sejbo1cny7 connector' interface. The left sidebar shows a navigation menu with options like Home, Devices, Overviews, Groups, Device types, and Management. The 'Management' section is expanded, showing 'Shell' as the active tab. The main area is divided into 'Command' and 'Operations' sections. The 'Command' section has a 'Predefined commands' button and a text area for commands. The 'Operations' section shows a list of operations, with the most recent one being a 'SUCCESSFUL' bulk device upload operation. The details of this operation are shown, including the description, status, operation ID, and the command used. The response is a JSON object indicating successful registration of devices.

⚠ CAUTION

We recommend you to never delete the connector device.

MIGRATION OF THE LWM2M DEVICES

Starting from version 10.15.0, the new device registration for LWM2M is introduced. LWM2M devices created earlier than version 10.15.0 and new LWM2M devices created via bulk registration must be migrated to the new structure using the `migrateLwm2mDevices` command for the tenant. It migrates all devices and their existing client registration objects to the new format in the database. It is backwards compatible since it retains old fragments.

The argument `-d` or `--devices` followed by a list of managed object IDs can be used to migrate specific device managed objects. To skip the migration of their corresponding client registration objects, use the `-sr` or `--skipRegistrations` argument.

Example usages:

- `migrateLwm2mDevices` : to migrate all devices and client registration objects of the tenant
- `migrateLwm2mDevices --skipRegistrations` : to migrate all devices without their client registration objects
- `migrateLwm2mDevices --devices 1111 2222` : to migrate specific devices with their client registration objects
- `migrateLwm2mDevices -sr -d 1111 2222` : to migrate specific devices without their client registration objects

INVALIDATE REGISTRATIONS

The LWM2M connector device may be used to invalidate LWM2M registrations. This is sometimes helpful to force a LWM2M device to re-register.

INVALIDATE REGISTRATIONS BY ENDPOINT

This command removes the LWM2M registrations using an endpoint ID.

Syntax: `invalidateRegistrationsForEndpoint <endpoint_ID>`

Example usage: `invalidateRegistrationsForEndpoint urn:imei:012345678901234`

This command invalidates all known LWM2M registrations for the endpoint `urn:imei:012345678901234`.

INVALIDATE REGISTRATIONS BY LWM2M REGISTRATION ID

Alternatively an LWM2M registration may be invalidated using its ID, using the following command:

Syntax: `invalidateRegistrationById <registration_ID>`

Example usage: `invalidateRegistrationById F7Dqjmw3Yy`

This command invalidates the LWM2M registration with ID `F7Dqjmw3Yy`.

LWM2M DEVICE PROTOCOLS

To process data from LWM2M devices, Cumulocity uses device protocols. Device protocols are accessible through the **Devices Types** menu in the Device Management application. For details on the general usage, see [Managing device types](#).

CREATING LWM2M DEVICE PROTOCOLS

Once you have registered a device, you can manage LWM2M device protocols. Each piece of information available by the LWM2M client is a resource. The resources are further logically organized into objects. The LWM2M client can have any number of resources, each of which belongs to an object. In the device protocols you can observe your resources. Furthermore, you can choose whether to create measurements, events or alarms out of those resources.

To add a new LWM2M device protocol follow these steps:

1. In the Device Management application, move to the **Device protocol** page.
2. Click **Add device protocol** in the top menu bar.
3. In the upcoming dialog select **LWM2M** as device protocol type.
4. Next, upload an appropriate DDF or XML file. DDF or XML files describe the data provided by your device. They are typically provided by the manufacturer or by standards bodies such as IPSO. There are also 3 "special" device protocols (DDF files) for standard OMA objects: 6 (location tracking), 5 (firmware update) and 3 (device information). If these files are not uploaded, then neither location tracking nor firmware updates will work. By default, the LWM2M agent adds mappings to these objects and knows how to "handle" their information without any additional configuration. The XML schema used by LWM2M can be found at <http://www.openmobilealliance.org/tech/profiles/LWM2M.xsd>.
If the DDF files for the default mappings are uploaded in the Management tenant, all subscribed user tenants will inherit this behavior. In the next dialog, you can see the name and description of the protocol. Click **Complete** to create the new device protocol.
5. The device protocol opens in a new page that contains the object ID and description and the list of resources that belongs to this object. In this page additional functionalities to a resource can be added.

DEVICE MANAGEMENT

[Home](#)
[Devices](#)
[Overviews](#)
[Groups](#)
[Device types](#)
[SmartREST templates](#)
[Device protocols](#)
[LWM2M post-operati...](#)
[Management](#)

powered by **CUMULOCITY**

3303 Temperature

Device types > Device protocols

Temperature

ID: 3303

Date created: 3/4/25, 1:19 PM

Last update: 3/4/25, 1:19 PM

Description: This IPSO object should be used with a temperature sensor to report a temperature measurement. It also provides resources for minimum/maximum measured values and the minimum/maximum

[LWM2M](#)

Resources

Name	Functionalities	<input type="checkbox"/> Ignore empty values	Validation rules
5601 Min Measured Value	--	<input type="checkbox"/>	
5602 Max Measured Value	--	<input type="checkbox"/>	
5603 Min Range Value	--	<input type="checkbox"/>	
5604 Max Range Value	--	<input type="checkbox"/>	
5605 Reset Min and Max Measured Values	--	<input type="checkbox"/>	
5700 Sensor Value	--	<input type="checkbox"/>	
5701 Sensor Units	--	<input type="checkbox"/>	

INFO

LWM2M protocol resources cannot be edited.

ADDING ADDITIONAL FUNCTIONALITIES TO A RESOURCE

To access resource functionalities proceed to the device protocol page and click one of the resource instances. See below for the functionalities that you may enable:

Send measurement

Turn on **Send measurement** to specify a measurement.

- Enter the type of the measurement. For example, "c8y_AccelerationMeasurement".
- Series are any fragments in measurements that contain a "value" property. For example, in the series field you can enter: "c8y_AccelerationMeasurement.acceleration".
- The "Unit" field specifies the unit of the given measurement. For example, "m/s" for velocity.

Create alarm

Turn on **Create alarm** if you want to create an alarm out of the resource. Specify the following parameters (all mandatory):

- Severity: CRITICAL, MAJOR, MINOR, WARNING
- Type
- Status: ACTIVE, ACKNOWLEDGED, CLEARED
- Text

Send Event

Turn on **Send event** to send an event each time you receive a resource value. Specify the following parameters:

- Enter the type of the event. For example, "com_cumulocity_model_DoorSensorEvent".

- Enter the text which will be sent. For example, "Door sensor was triggered".

Custom Actions

Turn on **Custom Actions** to map LWM2M data into Cumulocity using custom data processing actions. For specialized integration use cases, it is required to perform customized data processing on LWM2M data. One example are LWM2M resources of the OPAQUE data type that contain proprietary, binary data, CBOR, XML or alike.

We distinguish here between predefined custom actions and decoder microservices.

Predefined custom actions

There are several predefined custom actions which can be selected to apply actions to the relevant resources.

Actions that are relevant for a device object (object ID /3):

- device:updateManufacturer
 - Adds manufacturer information to the name of the device in the following format "LWM2M <manufacturer> <registration endpoint>"
- device:updateModelNumber
 - Stores to the device managed object with the `c8y_Hardware` fragment "model" property.
- device:updateSerialNumber
 - Stores to the device managed object with the `c8y_Hardware` fragment "serialNumber" property.
- device:updateFirmwareVersion
 - Stores to the device managed object with the `c8y_Hardware` fragment "revision" property.

Actions that are relevant for connectivity monitoring (object ID /4):

- connectivity:updateCellId
 - Stores to the device managed object with the `c8y_Mobile` fragment "cellId" property.
- connectivity:updateSmnc
 - Stores to the device managed object with the `c8y_Mobile` fragment "mnc" property.
- connectivity:updateSmcc
 - Stores to the device managed object with the `c8y_Mobile` fragment "mcc" property.
- connectivity:updateRssi
 - Stores the value as device measurement with the `c8y_SignalStrength` type and fragment and "rssi" property.
 - In the same measurement, stores resource path information in "resourcePath" fragment and also in "objectResourcePath_<resource path>" fragment name.
 - In the same measurement, stores device name information in "device!Name" fragment.
 - In the same measurement, stores device mobile information in "device!c8y_Mobile" fragment.

Below is an example where the "connectivity:updateRssi" custom action is selected for the Connectivity monitoring (/4) radio signal strength in order to create the signal measurement for the device.

Decoder microservices

Cumulocity LWM2M allows the set of custom actions to be extended using decoder microservices. A decoder microservice is an ordinary Cumulocity microservice that implements a simple decoder interface. The LWM2M agent calls this microservice for decoding data in a customer-specific way. We provide an according example how to write such a decoder microservice in our public [GitHub repository](#).

The LWM2M agent serializes different LWM2M data types to binary data depending on the type of the corresponding resource:

LWM2M resource type	Length (bytes)	Byte Order	Comment
OPAQUE	dynamic	unchanged	OPAQUE data is used to transmit raw binary blobs between a LWM2M device and a LWM2M server. The LWM2M agent simply forwards this data to the decoder microservice. The length and the order of the bytes remain unchanged.
INTEGER	8	Big Endian	
FLOAT	8	Big Endian	
STRING	dynamic		The string is serialized into a series of bytes, starting from the first character in the string. The first byte corresponds to its first character. Likewise, the last byte corresponds to the last character of the string.
CORE LINK	dynamic		The core links are first converted into a string representation of multiple links, separated by comma. Example: <code></43/0>,</33/1></code> . This string is then serialized to a series of bytes (see above).

TIME	8	Big Endian	Time data is represented as a 8-byte integer value (see above). The value here corresponds to the number of milliseconds since January 1, 1970, 00:00:00 GMT.
BOOLEAN	1		Boolean data is represented by a single byte (0 = false, 1 = true).
OBJLNK	dynamic		The OBJLNK is converted to a string first. The binary payload corresponds to the bytes of the string, starting with the first character.

Ignore empty values

If the **Ignore empty values** option is selected and a device sends empty data, the agent does not perform any action that is configured. If this option is not selected and a device sends empty data, the agent performs the mapped action (default behavior).

Auto observe

If **Auto-Observe** is turned on for a resource, the LWM2M server observes a specific resource for changes.

The screenshot shows the configuration page for a resource named '3304 Humidity'. On the left is a sidebar with navigation options like Home, Devices, Overviews, Groups, Device types, SmartREST templates, Device protocols, LWM2M post-operati..., and Management. The main area displays the resource details, including its ID (3304), creation date (3/4/25, 1:19 PM), and last update (3/4/25, 1:19 PM). Below this, there's a section for 'Resources' with a table showing the resource name, functionalities, and validation rules. The '5518 Timestamp' resource is listed. To the right of the resource name, there are toggle switches for 'Ignore empty values' and 'Auto Observe'. The 'Auto Observe' toggle is turned on and highlighted with an orange box. Below the toggles, there are fields for 'Type' (c8y_time) and 'Text' (time has changed). At the bottom, there are fields for 'Minimum period' (5 seconds) and 'Maximum period' (e.g. 12 seconds), along with 'Less than' and 'Greater than' options.

INFO

At least one functionality must be set to enable “Auto observe”.

ALARMS ON DEVICE PROTOCOL MAPPING FAILURES

There are two types of alarms raised related to device protocol mapping failures:

- Alarm for no mapping known: This alarm is raised when value is read or observed but no mapping for this resource is found. This can be resolved by importing device protocol for this resource.
- Alarm due to non-numeric/non-boolean value received for measurement mapping: This alarm is raised when the resource has a measurement mapping set up but measurement cannot be created because received value is a non-numeric/non-Boolean value.

LWM2M DEVICE DETAILS

INFO

In the Device Management application, you can view all details of a device. The following details are specific to LWM2M devices. For information on general details refer to [Viewing device details](#).

OBJECTS

In the **Objects** tab of a LWM2M device, you can view all objects, resources and instances of the device. Additionally, you can create new operations, see all currently pending operations and view the history of all previous operations.

DEVICE MANAGEMENT

Home

Devices

Overviews

Groups

Device types

Management

powered by CUMULOCITY

urn:imei:1-qMrcPrODXw

Devices > All devices > urn:imei:1-qMrcPrODXw > Objects

Audit configuration

Info

Measurements

Alarms

Control

Firmware

Software

Availability

Events

Location

LWM2M Config...

Objects

Shell

Tracking

Identity

Objects

Device /3/0

Firmware Update /5/0

Location /6/0

Temperature /3303/0

Location /6/0

Latitude

51.19999694824219

Longitude


6.75

Timestamp

1741090785000

INFO

In order to see resources in the **Objects** tab, the resources first must be added in the **Device Protocols** page.

The following operations may be available in each instance after clicking the menu icon  at the end of each object row:

- Read Object: Reads all instances for the selected object and lists all available resources for each instance.
- Read Instance: Reads the current instance of the given object and lists all available resources.
- Create Instance: Creates a new instance for the selected object.
- Delete Instance: Deletes the selected instance.

INFO

Some instances do not have all of the listed operations.

Some object cards show additional operations which can be performed. These operations become available after reading the object/instance. The possible options are **Write**, **Execute** and **Execute with parameters**. For example, after reading device **Firmware update** in order to perform the operation **Execute** without parameters, find the **Update** section on the object card and click **Execute**. To perform an operation with parameters click **Execute with parameters** and enter a value.

More information can be acquired for each resource by hovering over the help icon (?) present on the right of the field name.

Additional information on recent operations can be viewed by clicking the operations button located at the right side of an instance card. The button is only visible if any operation has been performed. The number of unread operations can be seen on the top right of the button. In the example below there are two.

To view the history of all operations, click **View history**. Note, that you will be redirected to the **Control** tab.

Audit Configuration

IMPORTANT

As announced in the release notes for [release 10.18](#), the LWM2M device audit configuration feature is deprecated. This feature will be disabled by default in a future version.

In the **Audit configuration** page you can audit the current device by comparing it to a selected reference device. It is also possible to sync properties to the values of the referenced device.

Click **Audit configuration** in the right of the top menu bar to navigate to the **Audit configuration** page.

To sync properties, select the desired reference device from the dropdown list. Check the properties that you wish to sync and click **Sync selected properties**.

INFO

The numbers in the green circles represent the number of properties in the instance which have the same value in both devices. Respectively, the numbers in the red circles represent the number of properties which have different values compared to the values of the referenced device. If an instance is expanded, you can select only specific properties which can be synced.

LWM2M CONFIGURATION

The **LWM2M configuration** tab displays all LWM2M settings related to the device. These settings are grouped logically by the function they affect.

- **Device settings** - General device parameters:
 - Endpoint ID
 - Awake time
 - Request timeout
 - Serialization format
 - Binary format
 - Timestamp resources
 - Objects tab behavior (keep old values)
 - Disable the default behavior for object instances of objects 3 (device), 4 (connectivity monitoring) and 6 (location)
 - Automatic setting of required interval
 - When enabled, the LWM2M service automatically sets the interval to registration lifetime plus 2 minutes. If disabled, the user can define a required interval for the device on the **Info** tab in the device details. If set to default, the default behavior of the LWM2M service will be used.
 - Logging event level - the logging levels can be adjusted based on types or completely be disabled for the device. When enabled, log events are visible in the device's events. Levels:
 - NONE - nothing will be logged as events
 - LIFECYCLE - only registration, de-registration and registration update events
 - VERBOSE - LIFECYCLE events, sent/received data and detailed firmware update information
- **Bootstrap settings** - related to LWM2M bootstrap
 - Bootstrap server ID
 - Security instance offset
- **Servers to write during bootstrap** - list of servers that will be written to the device when it bootstraps. Each server has:
 - Server URI
 - Server ID
 - Registration lifetime
 - Default min period
 - Default max period
 - Binding mode
 - Disable timeout
 - Security mode
 - depending on the selected mode there are additional settings for PSK and X.509
 - Notifications flag
- **Connectivity settings** - security configuration for bootstrap and regular LWM2M connection.
- **Firmware update settings**
 - Disable automated firmware update support
 - Firmware URL
 - Delivery method
 - Delivery protocol
 - Reset state machine settings
 - Fail device firmware update on unexpected result (terminate the device firmware update process if an unexpected situation is detected)

For a detailed description of the parameters above, see [Registering LWM2M devices](#).

LWM2M CLIENT AWAKE TIME

LWM2M client awake time specifies how long a device can be expected to be listening for incoming traffic before it goes back to sleep. The LWM2M server uses the client awake time to determine if the operations are passed down to a device. The operations are sent during the awake time after the registration or after the registration update request is received by the LWM2M server. After the awake time has passed, the operations are being queued and will be sent to the device on the next registration or registration update. This applies to all operations that can be applied to the device.

LWM2M client awake time is determined based on the following priority:

1. (If provided) Device managed object "awakeTimeRegistrationParameter" fragment.
2. (If provided) Registration awake time attribute "at" in the registration request by the LWM2M client.
3. Global setting of the LWM2M microservice.

Device managed object "awakeTimeRegistrationParameter" fragment can be provided during the device registration as explained in [Registering LWM2M devices](#) or set with the managed object update request as in the example:

```
PUT /inventory/managedObjects/<device-managed-object-id>
{
  "awakeTimeRegistrationParameter": 180000
}
```

The value is in milliseconds. If set to 0, the device will be considered as always online.

HANDLING LWM2M SHELL COMMANDS

In the **Shell** tab of a device, LWM2M shell commands can be performed. Each command has a different functionality. Find all available placeholders (for example "objectID", "instanceID") and commands with their respective descriptions below:

Placeholder	Description
objectID	The ID of the object.
instanceID	The ID of the instance. Some objects can have multiple instances. For example, "3300" is a temperature sensor object. Each device can have up to 10 sensors. In this case the instance ID would be 3300/1...10 depending on the sensor that you would like to focus.
resourceID	The ID of the desired resource. The resources describe the characteristics of each object. All instances of a given object have the same resources, but the value of the resources may be different.
value	The value to be written to the resource. Must be given using the type of the resource.
executeParameters	The execute parameters must conform to <i>arglist</i> ANBF syntax as described in the OMA Lightweight M2M specification (Section 6.3.5) .
Firmware version	The current version of the firmware.
Firmware url	The URL from which the new version of the firmware will be downloaded.
SER	The supported data formats are <code>TLV</code> , <code>JSON</code> , <code>CBOR</code> , <code>TEXT</code> , <code>OPAQUE</code> , <code>SENML_JSON</code> and <code>SENML_CBOR</code> . <code>DEFAULT</code> restores the standard behavior.

Placeholder	Description
requestJson	The raw CoAP request can be specified using the following JSON syntax.
<pre>REQUEST_JSON = { "method": \${METHOD}, "uri": \${URI}, "contentFormat" : \${CONTENTFORMAT}, "accept": \${ACCEPT}, "payloadHex": \${PAYLOADHEX} } METHOD = "get" "post" "put" "delete" "fetch" "ipatch" "patch" URI = "/" "[A-Fa-f0-9]" uri null CONTENTFORMAT = null "IANA Content Type" ACCEPT = null "IANA Content Type" PAYLOADHEX = null "[A-Fa-f0-9]+\$"</pre>	

In the next table you will see all available commands and a brief description of their functionality.

Command	Supported version	Description
read /<objectID>/<instanceID>/<resourceID>	1.0, 1.1	Reads a resource path.
cread /<objectID>/<instanceID>/<resourceID> [/<objectID>/<instanceID>/<resourceID>]	1.1	Composite reads of one or more resource paths. The resource data from all listed resource paths will be read in a single client response.
observe /<objectID>/<instanceID>/<resourceID>	1.0, 1.1	Enables the observe functionality.
cobserve /<objectID>/<instanceID>/<resourceID> [/<objectID>/<instanceID>/<resourceID>]	1.1	Enables composite observe functionality for one or more resource paths. The resource data from all listed resource paths will be sent to the Cumulocity's LWM2M agent in a single client request.
execute /<objectID>/<instanceID>/<resourceID> [<executeParameters>]	1.0, 1.1	Executes a resource on the device with optional parameters.
executelegacy /<objectID>/<instanceID>/<resourceID> [<STRING>]	1.0, 1.1	Executes a resource on the device and sends the parameters as TEXT/PLAIN string. This was the behavior of the execute command in Cumulocity until version 10.15. In contrast to the regular <code>execute</code> command, <code>executelegacy</code> allows execute parameters not in line with the Lightweight M2M standard to be sent to the device.
write /<objectID>/<instanceID>/<resourceID> <value>	1.0, 1.1	Writes value to a resource on the device.
cwrite /<objectID>/<instanceID>/<resourceID> <value> [/<objectID>/<instanceID>/<resourceID> <value>]	1.1	Composite writes facilitate the transmission of one or more values to LWM2M resources or resource instances on the device. The data will be written to the specified resource paths within a single request, adhering to the specified order. Writing multiple resource instance values directly to the corresponding resource ID is not permitted; each resource instance ID must be defined with its respective value. When writing multiple resource instance values, the resource instances will be transmitted exactly as defined in the operation, without aggregation or sorting by the resource instance ID.
writep /<objectID>/<instanceID>/<resourceID> <value>	1.1	Writes value to a resource on the device using COAP POST method.

Command	Supported version	Description
<p>writeb /<objectID>/<instanceID>/<resourceID> <hexadecimal-string> OR writeb /<objectID>/<instanceID>/<resourceID> binary:<binary-file-id></p>	1.0,1.1	<p>Writes binary data represented as a hex string to a resource on the device. The representation must be an even number of characters. For example: writeb /3442/0/150 010A0B020F.</p> <p>Writes binary data to a resource on the device from a file uploaded to the Cumulocity platform. The 'binary-file-id' is the object ID that has already been uploaded to the Cumulocity platform. Adding the prefix 'binary:' lets the agent read the file's data and write it to the device. For example: writeb /3442/0/150 binary:12345.</p>
<p>cancelobservation /<objectID>/<instanceID>/<resourceID> [/<objectID>/<instanceID>/<resourceID>]</p>	1.0, 1.1	Cancels either a single or a composite observation of the desired resources using a "reset" message. To cancel a composite observation the same list of paths must be mentioned as the composite observation was created. The list of paths is not required to be in the same order as the composite observation that was created.
<p>cancelobserve /<objectID> [/<instanceID>/<resourceID>]</p>	1.1	Cancels a single observation of the desired path using "GET with observe option" method.
<p>cancelCompositeObserve /<objectID> [/<instanceID>/<resourceID>] [/<objectID>/<instanceID>/<resourceID>]</p>	1.1	Cancels a composite observation of the desired paths using "GET with observe option" method.
<p>delete /<objectID>/<instanceID> [/<resourceID>]</p>	1.0, 1.1	Deletes a given object/instance/resource.
<p>discover /<objectID>/<instanceID>/<resourceID></p>	1.0, 1.1	Shows all resources of the given object.
<p>create /<objectID> [JSON]</p>	1.0, 1.1	Creates a new object. The JSON argument is optional.
<p>writeattr /<objectID>/<instanceID>/<resourceID> pmin= <sec>&pmax=<sec>&gt=<num>&lt;= <num>&st=<num>&cancel</p>	1.0, 1.1	Writes additional attributes to the object. Typically used for conditional observes.
<p>fwupdate <Firmware name>;<firmware version>;<firmware_url></p>	1.0, 1.1	Updates the firmware of the agent.
<p>serialization <SER></p>	1.0, 1.1	Sets the data format.

Command	Supported version	Description
coap <requestJson>	1.0, 1.1	<p>Allows a raw CoAP request to be sent to a LWM2M device. The command takes a request JSON string as a single argument.</p> <p>Example:</p> <pre>coap { "method" : "get", "uri" : "/3/0", "accept" : "application/vnd.oma.lwm2m+json"}</pre> <p>The CoAP response data is populated into the operation response. Note that Cumulocity does not further process CoAP responses. We also recommend you to use raw CoAP requests for device interactions only in exceptional cases. Any interaction with an LWM2M device should be carried out using standard LWM2M operations.</p>

INFO

A shell command can also be used to send multiple operations to a LWM2M device at once. To do that, simply enter all instructions with a line break between them. Make sure that the shell command does not carry any leading or trailing white spaces. The LWM2M agent then uses the line break to split a multi-line operation into separate LWM2M shell operations.

SHELL COMMAND LIFECYCLE

The handling of LWM2M shell commands follows the following lifecycle:

1. When a shell command is created the corresponding operation status is set to PENDING. This means that the corresponding CoAP request has not yet been sent to the device.
2. The Cumulocity platform processes the shell command. It sends a corresponding CoAP request to the device and updates the operation status to EXECUTING.
3. The next status update depends on the response of the device.
 - *Successful request:* In case the device signals a successful operation using a 2.XX CoAP response code, the operation result is updated and the operation status is turned to SUCCESSFUL.
 - *Failed requests:* If the CoAP request fails with a 4.XX or 5.XX error on the device, the operation is marked as FAILED. The operation result contains a possible response of the device.
 - *Not-responding:* When sending a LWM2M command to a device the Cumulocity platform is not precisely aware if the device can be reached using a UDP datagram. If the request times out Cumulocity assumes that there is no connectivity. It puts the operation back to PENDING. A redelivery of the operation is triggered as soon as the device sends a registration update or a new LWM2M registration request.
4. To view the history of all operations, click the **Control** tab.

INFO

If enabled, the agent will periodically look for starved operations of a tenant and fail them automatically. Starved operations are device operations which have had a status of EXECUTING and have not been updated for a long time. Platform administrators can configure how long such operations stay alive. This is described in the *LWM2M agent installation & operations guide*, see also [Additional resources > Installation and operations documentation](#). Contact your Operations team for further details.

ADDING VALIDATION RULES TO RESOURCES

Validation rules are used to verify that the data a user enters in a resource meets the constraints you specify before the user can save the resource.

Validation rules can only be added to resources which have “write” permissions. Resources which can have validation rules are marked by the following icon:

The screenshot shows the '7711 Cumulocity Validation' page in the Cumulocity Device Management interface. The page title is '7711 Cumulocity Validation' with a subtitle 'Object designed for manual validation testing'. The page is divided into a left sidebar with navigation options (Home, Devices, Overviews, Groups, Device types, SmartREST templates, Device protocols, LWM2M post-operati..., Management) and a main content area. The main content area shows a table of resources with the following columns: Name, Functionalities, Ignore empty values, and Validation rules. The resource '2 Sample integer' is highlighted with an orange box, indicating it is eligible for validation rules. The table also includes a 'Resources' section with a list of resources and their corresponding validation rules.

Name	Functionalities	Ignore empty values	Validation rules
0 Sample text		<input checked="" type="checkbox"/>	
1 Sample text array	--	<input type="checkbox"/>	
2 Sample integer	--	<input type="checkbox"/>	
3 Sample integer array	--	<input type="checkbox"/>	
4 Sample time	--	<input type="checkbox"/>	
5 Sample time array	--	<input type="checkbox"/>	
6 Sample boolean	--	<input type="checkbox"/>	
7 Sample boolean array	--	<input type="checkbox"/>	
8 Sample float	--	<input type="checkbox"/>	
9 Sample float array	--	<input type="checkbox"/>	

Add a new validation rule by clicking on the desired resource and then click **Add validation rule**.

Validation rules can have the following types:


- **Date:** Simply enter a date and select your desired rule.
- **Number:** Only values of “Integer” or “Float” type are allowed depending on the resource.
- **ObjectLink:** Reference to another object using the format “/Object/Instance/Resource”.
- **Regex:** Add a string which describes the validation pattern. For example, “.*dd” means that the string must end with “dd”.
- **String:** Enter a string value which can be either “True” or “False”.

After selecting a type, the following rules can be selected:

- Greater than
- Lower than
- Equals
- Equals not
- Greater or equals than
- Lower or equals than

INFO

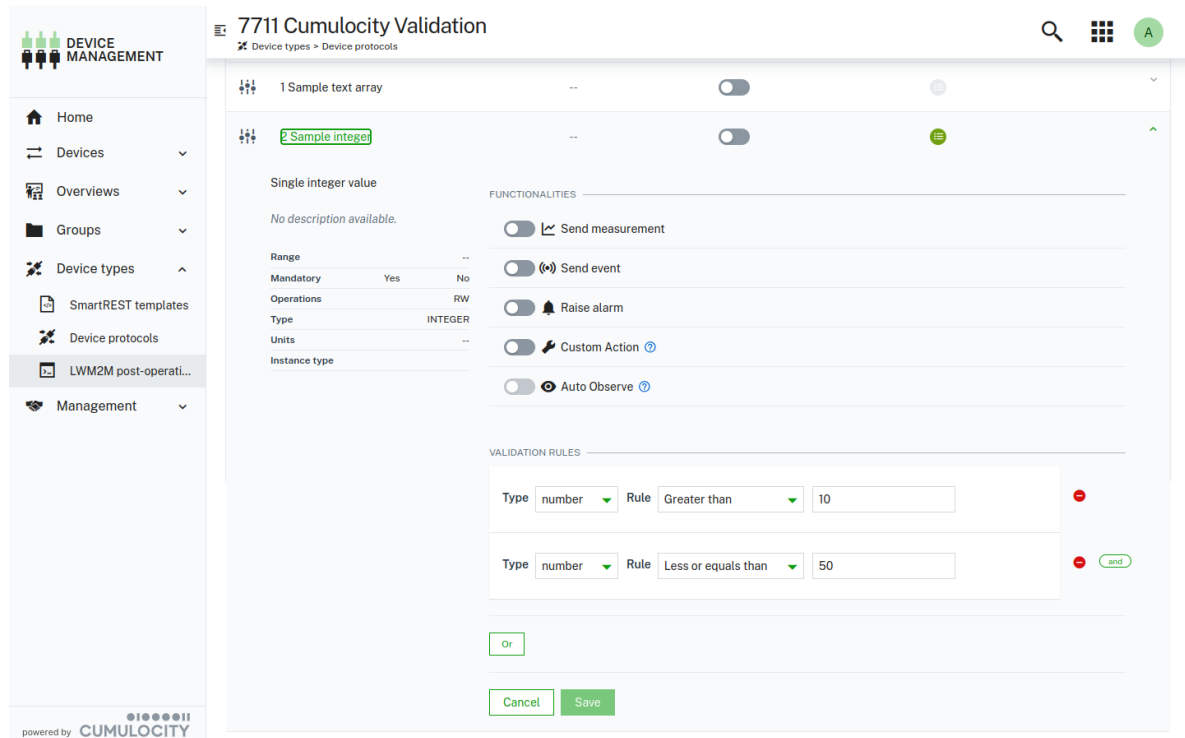
Not all rules are available to each type.

To delete a rule, click on the red remove icon  on the right side of that rule. Click **Save** to save your settings.

COMPLEX RULESETS

In order to enable more complex conditions, multiple validation rules can be defined for a resource:

- Multiple rules can be defined in a validation rule group. A user input is only valid if each of the rules in the validation rule group is satisfied (logical AND).
- It is possible to declare multiple validation rule groups. If multiple validation rule groups are declared, user input is valid if any of the validation rule groups is satisfied (logical OR).



The screenshot above provides an example for the use of validation rule groups: User input is valid if the given string does not match “test” (equals not). It is also valid if it ends with “asd” and it matches the contents of the LWM2M resource /3/0/15.

Complex rulesets are based on Boolean Disjunctive Normal Form, which allows arbitrary complex rules to be defined.

DEVICE LIFECYCLE EVENTS

The LWM2M agent creates events of device lifecycle in Cumulocity. Following are the specific event types for device bootstrap and registration process. The LWM2M agent creates the events with the specific event type during the device bootstrap and registration process.

- Bootstrap event types:
 - c8y_LWM2MDeviceBootstrapStart.
 - c8y_LWM2MDeviceBootstrapEnd.
 - c8y_LWM2MDeviceBootstrapFailure.
- Registration event types:
 - c8y_LWM2MDeviceRegistration.
 - c8y_LWM2MDeviceDeRegistration.
 - c8y_LWM2MDeviceRegistrationUpdate.

HANDLING LWM2M POST REGISTRATION ACTIONS

The LWM2M shell commands can be performed in the **Shell** tab of each device. It is also possible to execute some common operations when a device sends a full registration request. This can be done in the **LWM2M post-operations** page accessible from the **Device types** menu in the navigator. A set of shell commands can be saved in the Commands section, which will be performed on each device on registration.

The screenshot shows the 'LWM2M post-operations' page. At the top, there's a breadcrumb 'Device types > LWM2M post-operations'. Below the header, there's a 'Commands' section with a text area containing the following commands:

```
read /3303
discover /3
```

Below the text area is a green 'Save' button. To the right of the text area, there's a blue information icon and a text block explaining the command syntax:

You can enter a set of LWM2M shell commands. Each command will be sent to every newly registered LWM2M device. Each command must be entered in a new line. Use the following syntax for specifying post-registration actions:

```
# Discover all resources of Object 3300, the generic sensor.
discover <objectId>

# Read maximum sensor value
read <objectId>/<objectInstanceId>/<resourceId>

# Write application type "CO2" to device
write <objectId>/<objectInstanceId>/<resourceId> <applicationType>

# Reset min and max values
execute <objectId>/<objectInstanceId>/<resourceId>

# Observe sensor value. Causes device to send every new sensor value
observe <objectId>/<objectInstanceId>/<resourceId>

# Only send observations if sensor value is higher than 100
writeattr <objectId>/<objectInstanceId>/<resourceId> greater=<value>
```

The above image shows the **LWM2M post-operations** page with a set of sample shell commands. More information on shell commands can be found in [Handling LWM2M shell commands](#).

DEVICE OPERATIONS HANDLING

If the operations are created while the device is offline, all the operations will be executed when the device comes online as those operations will be delivered through the real-time channel. A configurable property can limit the number of operations to be executed as part of the post-registration process, for example, the operations which were already delivered to the device once via the real-time channel, but they still have a status of PENDING.

INFO

The default maximum limit of the pending operations is 10, which is a configurable value for the agent. In case this limit is not sufficient for your use case please contact [product support](#).

LWM2M DEVICE FIRMWARE UPDATE (FOTA)

The Cumulocity LWM2M agent supports FOTA (Firmware update Over The Air) using a firmware binary that is uploaded to the Cumulocity platform or hosted externally. To upload a firmware or specify an external firmware location, go to **Device Management > Firmware repository > Add firmware**

Select the firmware binary to upload from your local computer or enter an external location URL. The device type filter must be left empty or filled with a value of `c8y_lwm2m`.

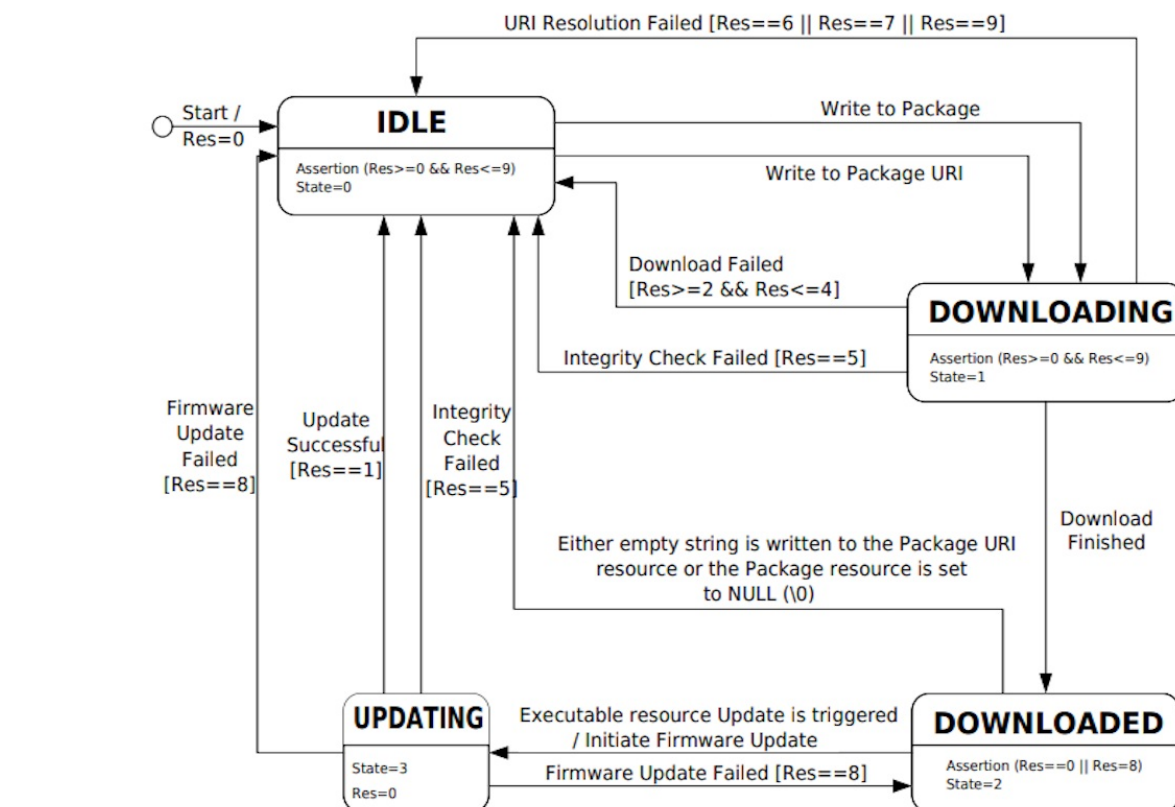
Assuming that you have already registered and connected your device, go to the device page to trigger a firmware update. Once the firmware update has been triggered, the LWM2M agent creates and queues up a PENDING firmware update operation for execution.

i INFO

This document is not supposed to cover every detail of firmware update process because they are already specified in the LWM2M specification. This instead summaries the process, highlights the key points and possible customizations of the firmware update process in Cumulocity LWM2M.

FIRMWARE UPDATE STATE MACHINE

The firmware update procedure is well standardized within the LWM2M specification, and a standard Firmware Update Object (/5) is used to perform the process. Let's have a quick glance at the firmware update state machine as defined by the LWM2M specification:



(Source: openmobilealliance.org)

Basically the whole update process contains different phases of interactions between the LWM2M server and the device. The above diagram consists of the possible states and transitions that could be introduced during the firmware update process.

If the device goes offline or is considered offline by the LWM2M agent, the firmware update operation is left IN_PROGRESS and the agent will try to resume the firmware update process if possible when the device connects again via a registration or registration update.

RESETTING THE STATE MACHINE

When the firmware operation is being executed, the LWM2M agent first of all tries to reset the firmware state machine to the original state to avoid any leftover downloaded firmware that has not been installed or failures of the previous firmware update attempts on the device. Cumulocity LWM2M agent supports the following mechanisms of resetting the firmware update state machine:

- If only PUSH delivery method is supported by the device, the state machine is reset by writing a byte array of a single element (value is 0) to the package resource: **write /5/0 10**
- If both PUSH and PULL or only PULL delivery method is supported, the state machine is reset by writing a NULL string to the

package URI resource: **write /5/0/1 10**

- This mechanism can also be specified in the device managed object by using fragment: **fwUpdateResetMechanism**. When this is set, the delivery method is disregarded. Possible values: **** PACKAGE**: This works the same as when only PUSH delivery method is supported, writing a byte array of a single element (value is 0) to the package resource: **write /5/0/0 10 ****
PACKAGE_URI: The state machine is reset by writing an empty string ("") to the package URI resource: **write /5/0/1 <empty string>**

If resetting the state machine has failed because the device is not reachable, the firmware update operation stays in PENDING status and will be executed when the device connects. If it's failed by any other reason, the firmware update operation set to FAILED. If the state machine is reset successfully, the firmware update operation is marked as IN_PROGRESS and the process continues to the next steps.

QUERYING THE DEVICE CONFIGURATION

In order to determine what is the best way to deliver the firmware to the device, the LWM2M agent tries to read the device configuration by executing a read request on the firmware update object on the device: read /5/0. In this step, the agent will learn:

- What the supported delivery methods are on the device specified by the value on resource **/5/0/9**, for example: 0 (PULL), 1 (PUSH) or 2 (BOTH). If both delivery methods are supported, PULL will be taken.
- What the supported delivery protocols are on the device, specified by the value on resource **/5/0/8**, for example: 0 (CoAP) or 1 (CoAPs). If this value is not specified by the device, 0 (CoAP) will be taken.
- What the current state of the firmware update is on the device. This value must be 0 (IDLE), otherwise the firmware update process is aborted immediately.

Supported firmware delivery methods and delivery protocols can also be specified in the device managed object by setting these fragments:

- **fwUpdateDeliveryMethod**. Possible values: PUSH, PULL, BOTH
- **fwUpdateSupportedDeviceProtocol**. Possible values: COAP, COAPS, HTTP, HTTPS

If they are specified in the device managed object, the values sent by the device are ignored.

FIRMWARE DELIVERY

As the first step of the delivery, the agent tries to establish the observations on two resources to monitor the firmware delivery transitions on the device:

- **observe /5/0/3**: Observe the firmware update state
- **observe /5/0/5**: Observe the firmware update result.

Depending on the supported delivery protocols and methods by the device, the agent now delivers the firmware to the device.

When PULL is selected as the delivery method, the agent will try to write the firmware URI to the device firmware package URI: write **/5/0/1** . The agent constructs the firmware URI according to the selected delivery protocol.

When PUSH is selected as the deliver method, the agent will try to write the firmware binary to the device firmware package: write **/5/0/0** .

In both cases, if the firmware binary cannot be delivered as one single message, the agent delivers the firmware using so-called block-wise transfer. The preferred size of each block can be specified by the device in the negotiation phase with the LWM2M agent. If the device does not specify it, the agent uses its default block size of 512 bytes.

When the delivery is completed on the device (no matter if it's successful or failed, for example, because the device runs out of storage, or due to network issues) the device must inform the agent by updating the value of the firmware update state (/5/0/3) and/or firmware update result (/5/0/5). Practically, the device can keep sending the value periodically for the firmware update state resource even if the firmware is still being transferred, with the value 1 (Downloading) or 2 (Downloaded).

TRIGGERING THE FIRMWARE UPDATE ON THE DEVICE

When the firmware delivery is completed successfully and the agent is informed, it will trigger the firmware update on the device by sending an execute request to the update resource: execute /5/0/2. Note that the observations on the update state and update result are still being maintained. When the update process is completed on the device, it must communicate to the agent by updating the value of firmware update result (and firmware update state).

COMPLETING OF THE FIRMWARE UPDATE PROCESS

When the firmware update is completed (no matter if it's successful or failed) on the device and the agent is informed, the agent

completes the firmware update process.

- If the firmware update is successful on the device, the agent sets the firmware information to the device managed object and marks the firmware update operation as completed successfully.
- If the firmware update has failed on the device, the agent marks the firmware update operation as failed.

CANCELING THE FIRMWARE UPDATE PROCESS

In practice, the communications between the device and the agent are not always smooth, for example in the case of network failures or the device is not able to report to the agent about its status, or in case of other failures, you might want to cancel the firmware update process entirely and start a new one. To do that, send an HTTP request as the following: `PUT .../service/lwm2m-agent/shell/{tenantId}/{deviceId}/cancelFirmwareUpdate` in which **tenantId** is the ID of your tenant, **deviceId** is your device managed object ID. The ongoing firmware update process will be canceled by the agent. Alternatively, the firmware update process is also canceled if you delete the firmware update operation.

LPWAN

INTRODUCTION

This section details the mechanisms for integrating non-IP devices. Cumulocity provides native connectors for three major LPWAN connectivity providers including [Loriot](#), [Actility](#), and [Sigfox](#). These connectors abstract the vendor-specific APIs required for device provisioning and data retrieval.

This section documents the configuration and operation of the LPWAN agent, which performs the following core functions:

- **Ingestion** - Receives uplink messages from Sigfox, Loriot, or Actility (ThingPark) backends.
- **Normalization** - Decouples the connectivity provider from the device logic, allowing devices to be managed using standard Cumulocity interfaces.
- **Translation** - Routes binary payloads through user-defined device protocols to extract measurements, events, and alarms.
- **Downlink management** - Queues and forwards commands to the Network Server APIs for transmission to the device.

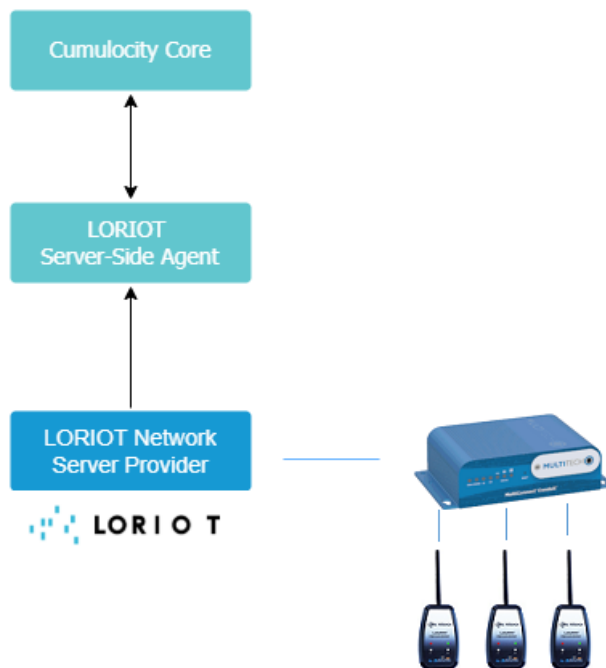
LORIoT LORA

INTRODUCTION

Cumulocity can interface with the Loriot Network Server through the Loriot agent microservice. You can:

- Register the device in two ways:
 - Create a Loriot LNS connection and register the device using Cumulocity.
 - Configure the Loriot agent endpoint via Loriot Network Server and register the device via uplink message. In order to be able to send downlink messages, the devices created using this method must be re-registered via Cumulocity to be associated with a connection and device type.
- Assign a device protocol for the LoRa device for payload processing.
- Decode upstream payload packets using a web-based user interface.
- Post-process raw device data through Cumulocity events.
- Make use of existing Cumulocity features with LoRa devices, for example: connectivity monitoring, device management, data visualization with dashboards, real-time analytics and more.

The following illustration gives an overview of the Cumulocity Loriot LoRa integration.



INFO

Your subscription must include this feature to be able to use it. If you do not see the functionality described in this document please contact [product support](#).

DEVICE REGISTRATION VIA UPLINK MESSAGE

Before using LoRa devices with Cumulocity, you must configure the Cumulocity Lorient agent endpoint details in Lorient Network Server.

Configuring the Lorient endpoint using basic authentication

In Lorient Network Server you can create multiple applications. Each application allows you to configure LoRa devices.

To specify the Lorient agent endpoint with user credentials, navigate to one of the applications in your Lorient Network Server account and select **Output** in the **Application** menu in the navigator.

Lorient Network Server forwards the LoRa device messages to the external applications using different connectors which are available in the **Output** section.

Application Output / BE010106

Choose output type

Where should we feed your IoT data? See the Application API catalog for details

Cumulocity
Mechanism Data delivery through 3rd party Cumulocity cloud service

Setup guide

1. Sign in to your Cumulocity IoT Tenant or create a [Free Trial Tenant](#)
2. Request the activation of LORIoT service by filling in [this form](#) or through your point of contact at Software AG
3. Connect to your tenant on Cumulocity IoT and from the Administration application, select Accounts > Users in the navigator. Click Add user and create the user with no roles
4. Enter your Tenant URL and credentials in the form below
5. Devices will be automatically created in your Tenant as data arrives. To configure LoRaWAN payload decoding please follow these [instructions](#)

Setup parameters

Tenant URL:

Tenant ID:

Username:

Password:

[Add Output](#) [Cancel](#)

Use Cumulocity data forwarder for configuring the Lorient endpoint using basic authentication.

Cumulocity
Mechanism Data delivery through 3rd party Cumulocity cloud service

Setup guide

1. Sign in to your Cumulocity IoT Tenant or create a [Free Trial Tenant](#)
2. Request the activation of LORIoT service by filling in [this form](#) or through your point of contact at Software AG
3. Connect to your tenant on Cumulocity IoT and from the Administration application, select Accounts > Users in the navigator. Click Add user and create the user with no roles
4. Enter your Tenant URL and credentials in the form below
5. Devices will be automatically created in your Tenant as data arrives. To configure LoRaWAN payload decoding please follow these [instructions](#)

Setup parameters

Tenant URL

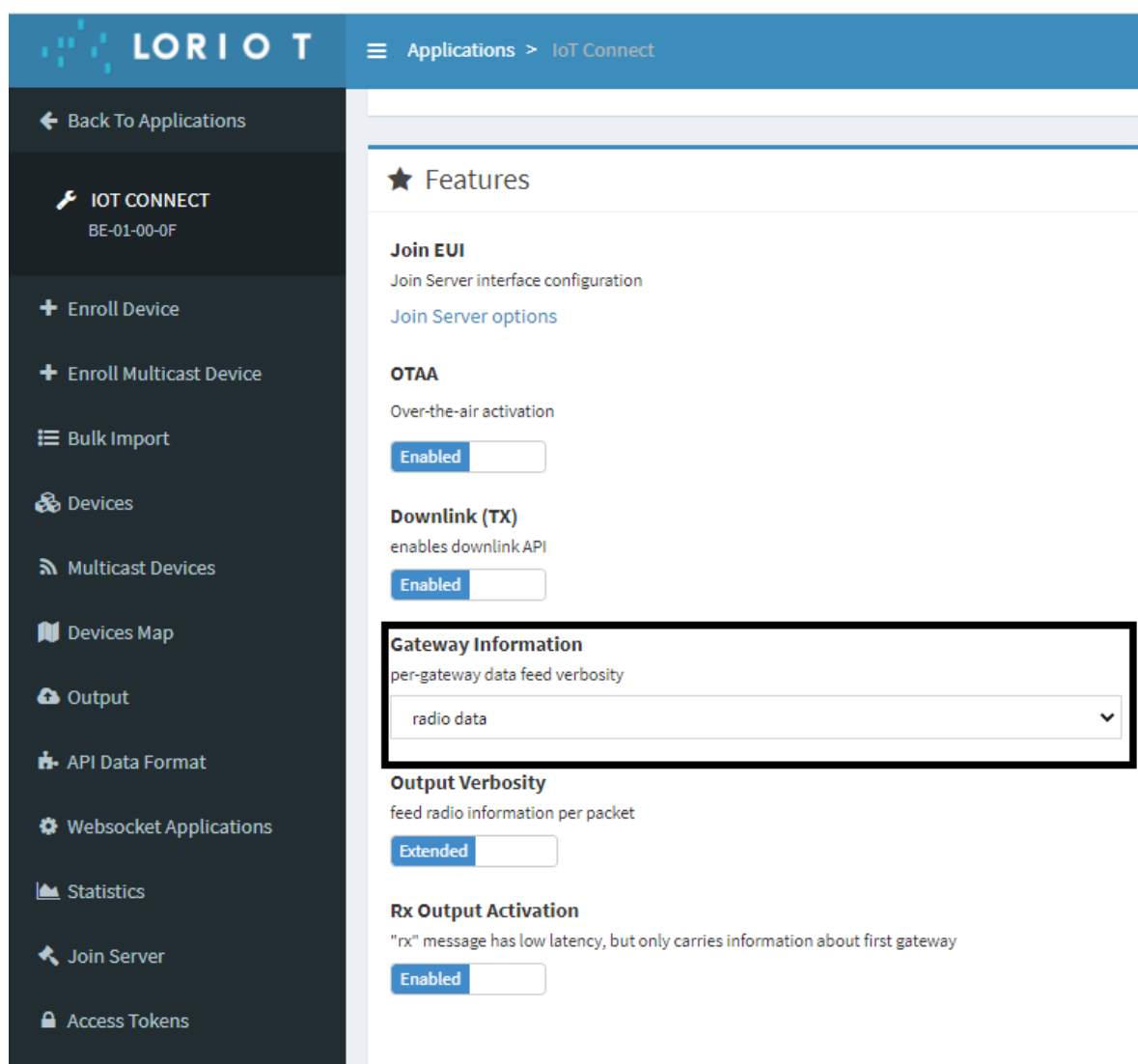
Tenant ID

Username

Password

[Add Output](#) [Cancel](#)

Always keep the **Gateway Information** option enabled because the Lorient agent only processes "gw" (gateway information) messages.



The Lorient devices can now be registered in Cumulocity when uplink messages are received.

Device creation via Lorient uplink message

While processing the Lorient LoRa device request, the Lorient agent automatically creates the device in the Cumulocity platform, if it does not yet exist. This means that you do not have to register the Lorient LoRa device explicitly.

The Lorient Network Server forwards two types of messages to the Lorient agent: “rx” (uplink message) and “gw” (gateway information).

The Lorient LoRa agent only processes “gw” messages to avoid duplicate measurements or events in Cumulocity, because most of the information matches with “gw” message whereas “gw” message also carries all gateway information.

INFO

You must enable the “gw” message option on Lorient Network Server while connecting to the Lorient LoRa agent, see [Device registration via Cumulocity](#).

In the Lorient LoRa device message below, **gws** represents a list of gateways involved in the network:

```
{
  "cmd" : "gw",
  "EUI" : "0102030405060708",
  "ts" : 1470850675433,
  "ack" : false,
  "fcnt" : 1,
  "port" : 1,
  "data" : "0102AABB",
  "freq" : 868500000,
  "dr" : "SF12 BW125 4/5",
  "gws" : [
    {
      "rssi" : -130,
      "snr" : 1.2,
      "ts" : 43424140,
      "gweui" : "1122334455667788.0",
      "lat" : 47.284687,
      "lon" : 8.565746
    }
  ]
}
```

The Lorient LoRa agent picks `gw` with the oldest timestamp for processing. The Lorient LoRa agent maps the `rssi` value to the standard Cumulocity `SignalStrength` object and updates the device managed object with the `lat` and `lon` values.

In order to be able to send downlink operations, the devices registered via uplink message must be re-registered using Cumulocity (see [Device registration via Cumulocity](#)), to be associated with a connection and a device type.

DEVICE REGISTRATION VIA THE CUMULOCITY PLATFORM

Creating a Lorient LNS connection in Cumulocity

Before using LoRa devices with Cumulocity, you must configure the Cumulocity Lorient agent endpoint details in the Administration application. Click the **Connectivity** tab in the **Settings** menu to create, edit, delete or update multiple Lorient connections.

To add a new connection

When you select **Connectivity** for the first time, you are asked to create a connection. Click **Add Connection**.

Enter the following information:

- **Name** - the name of the Lorient connection being created
- **Base URL** - the URL associated with the Lorient provider account
- **Username** - your Lorient account username
- **Password** - your Lorient account password

Click **Save**. If the information you have entered is correct, the message "Connection created" appears.

To add another connection, click **Add Connection** and follow the steps above.

INFO

Always keep the **Gateway Information** option enabled because the Lorient agent only processes "gw" (gateway information) messages.

The screenshot displays the LOR I O T IoT Connect configuration interface. The left sidebar contains navigation links: Back To Applications, IOT CONNECT (BE-01-00-0F), Enroll Device, Enroll Multicast Device, Bulk Import, Devices, Multicast Devices, Devices Map, Output, API Data Format, Websocket Applications, Statistics, Join Server, and Access Tokens. The main content area is titled 'Applications > IoT Connect' and features a 'Features' section. This section includes configuration options for Join EUI, OTAA, Downlink (TX), Gateway Information, Output Verbosity, and Rx Output Activation. The 'Gateway Information' section is highlighted with a black box, showing a dropdown menu set to 'radio data' for 'per-gateway data feed verbosity'.

To update a connection

Select the connection to be updated, make your edits, and save the connection.

If there are devices associated with the connection, an error message appears, stating "Can not update the LNS Connection with `<name of LNS Connection>` as it's associated with `<number of devices>`". Click the link to download the file with the details of the associated devices: `/service/<agent-context-path>/lms-connection/<lms-connection-name>/device`".

ADMINISTRATION

- Home
- Accounts
- Tenants
- Ecosystem
- Business rules
- Management
- Settings
- Authentication
- Application
- Properties library
- Enterprise tenant
- License management
- Branding
- Configuration
- Connectivity
- Localization
- Data broker

Connectivity
Settings > Connectivity > LORIoT

LORIoT connections

loriot connection

Name
loriot connection

Description
e.g. This connection has a built-in functionality to...

URL
https://eu1.loriot.io

Username
username@domain.com

Change password

Add connection Cancel Delete Save

powered by CUMULOCITY

To delete a connection

Select the connection to be deleted and click **Delete**.

If there are devices associated with the connection, an error message appears, stating “Can not delete the LNS Connection with **<name of LNS Connection>** as it's associated with **<number of devices>**”. Click the link to download the file with the details of the associated devices: **/service/<agent-context-path>/lms-connection/<lms-connection-name>/device**”.

ADMINISTRATION

- Home
- Accounts
- Tenants
- Ecosystem
- Business rules
- Management
- Settings
- Authentication
- Application
- Properties library
- Enterprise tenant
- License management
- Branding
- Configuration
- Connectivity
- Localization
- Data broker

Connectivity
Settings > Connectivity > LORIoT

LORIoT connections

test3

Name
test3

Description
e.g. This connection has a built-in functionality to...

URL
https://eu1.loriot.io

Username
grigor.lekarov@cumulocity.com

Change password

Add connection Cancel Delete Save

powered by CUMULOCITY

Failed to delete the connection

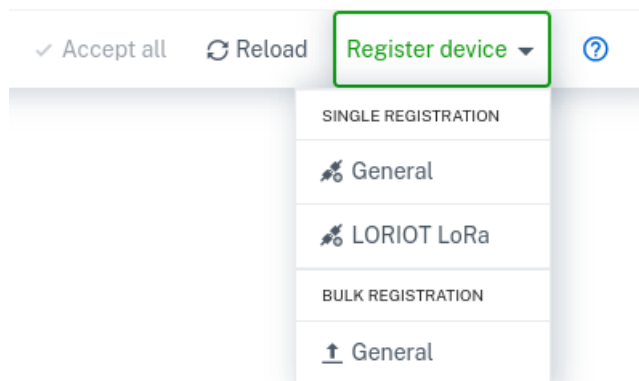
Can not delete the LNS connection with name 'test3' as it's associated with '1' device(s).

Click the link to download the file with the affected devices.

Cancel

Loriot device registration

To register a Loriot device in Cumulocity navigate to **Devices > Registration** in the Device Management application, click **Register device** at the top right and select **Single device registration > Loriot LoRa** from the dropdown.




INFO

If Loriot is not one of the available options, your tenant is not subscribed to the relevant applications, see information at the top.

In the next window, fill in the required information:

- **Title** - title of the device to be registered.
- **Device EUI** - this is the unique identifier for the device. It is a 16 character (8 byte) long hexadecimal number. You can find it on the device itself.
- **Application EUI** - this is a global application ID in the IEEE EUI64 address space that uniquely identifies the application provider of the device. It is a 16 character (8 byte) long hexadecimal number.
- **Application key** - this is an AES-128 application key specific for the device that is assigned to the device by the application owner and is responsible to encrypt. The application key is a 32 character (16 byte) long hexadecimal number.
- **Connection** - lists all configured Loriot connections in the tenant. The **Application name** option (see below) is populated based on the selected Loriot connection.
- **Application name** - select the appropriate application name under which the device must be registered in the Loriot provider.
- **Device protocol** - select the appropriate device protocol from the dropdown list. For more information on how to create a device protocol refer to [Creating device protocols](#).


LORIoT REGISTRATION

Register a single LORIoT device

Title

Sample Lorient Device

Device EUI

DABB121243435678

Application EUI

DABC121243435678

Application key

DABB121243435678DABC121243435678

Connection

test3

Application name

TestApp

Device protocol

test

Cancel

Register

Click **Register** to submit the device registration request and create the device.

You can verify that the device is connected by incoming events. Click on a device and open its **Events** tab. All events related to this device are listed.

For more information on viewing and managing your connected devices, also refer to the [Device Management application](#).

In order to migrate the device from one LNS connection to another, the device must be re-registered:

1. Navigate to the **LPWAN** tab of the Device.
2. Click the **Provider connection** dropdown.
3. A prompt will appear stating that in order to migrate the device from one LNS connection to another, you must re-register the device. Click the **Re-Register** button.
4. You are directed to the device registration page where you can perform the re-registration following the steps above and selecting the desired LNS connection.

ASSIGNING THE LORIoT ADMIN ROLE PERMISSION

In the Cumulocity platform, assign the Lorient admin role permission to the user configured in the Lorient Network Server.

In the Administration application, click **Roles** in the navigator and select the **ADMIN** checkbox for "Lorient".

The screenshot shows the 'Admin User' configuration page in the Cumulocity Administration application. The left sidebar contains the 'ADMINISTRATION' menu with options like Home, Accounts, Users, Roles, Audit logs, Tenants, Ecosystem, Business rules, Management, Settings, and Data broker. The main content area is titled 'Admin User' and shows the 'Global role' configuration. The 'Name' field is 'Lorient Admin Role' and the 'Description' field is 'Can access Lorient Endpoint'. The 'Permissions' section shows a table with columns for TYPE, READ, ADMIN, CREATE, and UPDATE. The 'Lorient' role is listed with the ADMIN checkbox checked. The 'Application access' section shows a list of subscribed applications with checkboxes for each.

}

CREATING DEVICE PROTOCOLS

To process data from LoRa devices, Cumulocity needs to understand the payload format of the devices. Mapping payload data to Cumulocity data can be done by creating a LoRa device protocol.

During the [device registration](#), you can associate this device protocol. The received uplink message for this device with a hexadecimal payload will then be mapped to the ones you have configured in your device protocol.


INFO

Device protocol mapping only supports decoding for fixed byte positions based on the message type. The length for the device payload parts, which is set in the **Number of bits** field, can be maximum 32 bits (4 bytes).

In order to create a device protocol, navigate to the Device Management application and select **Device protocols** in the **Device types** menu in the navigator. You can either import an existing device protocol or create a new one.

Importing a predefined device protocol

1. In the **Device protocols** page, click **Import**.
2. Select the predefined device type, for example "LoRaWAN Demonstrator" or upload from a file.
3. Click **Import**.



IMPORT DEVICE PROTOCOL

1 SELECT DEVICE PROTOCOL

Select from predefined

Adeunis LoRaWan Demonstrator

Or load it from a file

Select file to upload...

2 SAVE WITH THE FOLLOWING NAME

Name

Adeunis LoRaWan Demonstrator


Cancel

Import

Alternatively, you may also load the device protocol from a file and import it.

Creating a new device protocol


In the **Device protocols** page, click **New device protocol** in the top menu bar. The following window will open:



ADD DEVICE PROTOCOL

Select one of the available options

Find your protocol in the [user documentation](#) to get more information.

 LoRa

Cancel

Select **LoRa** as the device protocol type, provide a name for it and click **Create**.

Under **Message types**, specify the message types. LoRa devices can send messages of different types with different encodings per type.

Select the way the message type is encoded in the **Source** dropdown box:

- **FPort** - if the message type can be determined by looking at the FPort parameter of a message.
- **Payload** - if the message type can be determined by looking at the subset of the message payload itself.

In the following example payload structure, the first byte indicates the message type source (as highlighted).

Message type source

0	1	2	3	4	5	6	7	8	9	10	11
Code	Status	PAYLOAD									
0x01	Cf Status	Channel 1 Type	Measure sensor 1 (LSB First)			Channel 2 Type	Measure sensor 2 (LSB first)			X	X
0x03	Cf Status	Device Type	Transmit period	Channel On/Off	Channel 1 Type	Channel 2 Type	NA (0x00)	Switch Value	X	X	

In the user interface you can enter this type of message type source information as follows: In the **Start bit** field, indicate where the message type information starts in the payload and in the **Number of bits** field, indicate how long this information is, for example start bit = "0" and number of bits = "8".

LoRa Test Protocol

Device types > Device protocols

Search

Grid

A

LoRa

LoRa Test Protocol

e.g. My protocol description

ID

3438224

Date created

Mar 6, 2025, 12:37:40 AM

Last update

Mar 6, 2025, 12:37:40 AM

Fieldbus version

4

Message types

Source

Payload

Start bit ?

0

Number of bits ?


8

Values

No values defined.

LoRa devices can send messages of different types with different encodings per type. Depending on the device, the type can be determined by looking either at the FPort parameter of a message (**Source: FPort**) or at the subset of the message payload itself (**Source: Payload**). Indicate where the type information starts in the payload (**Start bit**) and how long this information is (**Number of bits**).

Click **Add value** to create the value configuration.



LoRa Test Protocol

e.g. My protocol description

ID 3438224

Date created Mar 6, 2025, 12:37:40 AM


Last update Mar 6, 2025, 12:37:40 AM

Fieldbus version 4


Message types

Source

Payload

Start bit 


0


Number of bits 

8

LoRa devices can send messages of different types with different encodings per type. Depending on the device, the type can be determined by looking either at the FPort parameter of a message (**Source: FPort**) or at the subset of the message payload itself (**Source: Payload**). Indicate where the type information starts in the payload (**Start bit**) and how long this information is (**Number of bits**).

Values

 No values defined.

 Add value

Save

In the upcoming window, configure the relevant values as shown in this example.

New value window part 1

New value

MESSAGE TYPE ?

Message ID ?

GENERAL

Name

Display category

VALUE SELECTION ?

Start bit

Number of bits

VALUE NORMALISATION ?

Multiplier

Offset

Unit

New value window part 2

OPTIONS

☐ Signed ?
☐ Packed decimal ?
☐ Little-endian ?

FUNCTIONALITIES

☐ Send measurement ?
☐ Raise alarm ?
☒ Send event ?

Event type

Event text

Event fragment

Event property ?

☐ Update managed object ?

The value configuration maps the value in the payload of a message type to the Cumulocity data.

Under **Message type**, configure the **Message ID** according to your device message specification and map it to the Cumulocity data. The message ID is the numeric value identifying the message type. It will be matched with the message ID found in the source specified on the device protocol main page (that is, Payload or FPort). The message ID must be entered in decimal numbers (not hex).

In this example payload structure the message ID is "1".

Message ID

0	1	2	3	4	5	6	7	8	9	10	11
Code	Status	PAYLOAD									
0x01	Cf Status	Channel 1 Type	Measure sensor 1 (LSB First)		Channel 2 Type	Measure sensor 2 (LSB first)			X	X	
0x03	Cf Status	Device Type	Transmit period	Channel On/Off	Channel 1 Type	Channel 2 Type	NA (0x00)	Switch Value	X	X	

New value

MESSAGE TYPE ?

Message ID ?

1

GENERAL

Name
Channel type1

Display category
Sensor measure data

VALUE SELECTION ?

Start bit
16

Number of bits
8

VALUE NORMALISATION ?

Multiplier
1

Offset
0

Unit
e.g. u

OPTIONS

☐ Signed ?
☐ Packed decimal ?
☐ Little-endian ?

Cancel

Save

Under **General**, specify a name for the value and the category under which it will be displayed in the values list. The associated name for this value will be displayed under the **Display category** given.

Under **Value selection**, define from where the value should be extracted. In order to do so, indicate where the value information starts in the **Start bit** field and how long this information is in the **Number of bits** field. The maximum value for the number of bits is 32 bits (4 bytes).

In this example the "Channel 1 Type" information starts in byte 2 (that means, start bit = "16") and is 1 byte long (that means, number of bits = "8").

Value selection:
start bit and number of bits

0	1	2	3	4	5	6	7	8	9	10	11
Code	Status		PAYLOAD								
0x01	Cf Status	Channel 1 Type	Measure sensor 1 (LSB First)			Channel 2 Type	Measure sensor 2 (LSB first)			X	X
0x03	Cf Status	Device Type	Transmit period		Channel On/Off	Channel 1 Type	Channel 2 Type	NA (0x00)	Switch Value	X	X

New value

MESSAGE TYPE [?](#)

Message ID [?](#)

GENERAL

Name

Display category

VALUE SELECTION [?](#)

Start bit

Number of bits

VALUE NORMALISATION [?](#)

Multiplier

Offset

Unit

OPTIONS

☐ Signed [?](#)

☐ Packed decimal [?](#)

☐ Little-endian [?](#)

The hexadecimal value is converted to a decimal number and afterwards a “value normalisation” is applied.

Under **Value normalisation** define how the raw value should be transformed before being stored in the platform and enter the appropriate values for:

- **Multiplier** - this value is multiplied with the value extracted from the **Value selection**. It can be decimal, negative and positive. By default it is set to 1.
- **Offset** - this value defines the offset that is added or subtracted. It can be decimal, negative and positive. By default it is set to 0.
- **Unit** (optional) - a unit can be defined which is saved together with the value (for example temperature unit “C” for degree Celsius).

For detailed information on how to decode the payload, refer to the documentation of the device.

i INFO

“Little endian” support to decode the payload has been added.

Under Options, select one of the following options, if required:

- **Signed** - if the value is a signed number.
- **Packed decimal** - if the value is BCD encoded.

Under **Functionalities**, specify how this device protocol should behave:

- **Send measurement** - creates a measurement with the decoded value.
- **Raise alarm** - creates an alarm if the value is not equal to zero.
- **Send event** - creates an event with the decoded value.
- **Update managed object** - updates a fragment in a managed object with the decoded value.

You can also have a nested structure with several values within a measurement, event or managed object fragment. In case of a measurement all the properties of the same type will be merged to create a nested structure. In case of an event or a managed object all the properties with the same fragment are merged to create a nested structure. Also refer to the [example](#) of a nested structure for a "Position" device protocol below.

Click **OK** to add the values to your device protocol.

LoRa Test Protocol

Device types > Device protocols

A

LoRa

LoRa Test Protocol

e.g. My protocol description

ID

3438224

Date created

Mar 6, 2025, 12:37:40 AM

Last update

Mar 6, 2025, 12:37:40 AM

Fieldbus version

4

Message types

Source

Payload

Start bit

0

Number of bits

8

LoRa devices can send messages of different types with different encodings per type. Depending on the device, the type can be determined by looking either at the FPort parameter of a message (**Source: FPort**) or at the subset of the message payload itself (**Source: Payload**). Indicate where the type information starts in the payload (**Start bit**) and how long this information is (**Number of bits**).

Values

SENSOR MEASURE DATA

Channel type1

MESSAGE TYPE ID 1 START BIT 16 NUMBER OF BITS 8

Add value

Save

After clicking **Save**, your device protocol is created with the values you defined.

[Example with single property](#)

The following image shows an example for a message which sends a measurement when the battery level changes.

New value window part 1

New value

MESSAGE TYPE ?

Message ID ?

GENERAL

Name

Display category

VALUE SELECTION ?

Start bit

Number of bits

VALUE NORMALISATION ?

Multiplier

Offset


Unit

OPTIONS

☐ Signed ?☐ Packed decimal ?☐ Little-endian ?



New value window part 2

FUNCTIONALITIES ?

 Send measurement ?

Measurement type

Measurement series

 Raise alarm ? Send event ?

Example with nested structure


The following image shows an example of a nested structure for a device protocol reporting the current position of a GPS device. The display category is named "Position" and contains values for longitude and latitude.

The message ID should be the same for all the values. Enter the rest of the parameters according to the instructions above. Enter "c8y_Position" in the **Managed object fragment** field and create a new value for each: longitude and latitude.


New value window, Longitude

FUNCTIONALITIES ?


☐

 Send measurement ?


☐

 Raise alarm ?

☐

 Send event ?

☒

 Update managed object ?

Managed object fragment

c8y_Position

Managed object property ?

lng


Cancel

Save


New value window, Latitude

FUNCTIONALITIES ?


☐

 Send measurement ?


☐

 Raise alarm ?

☐

 Send event ?

☒

 Update managed object ?

Managed object fragment

c8y_Position

Managed object property ?

lat

Cancel

Save

This will be the result:

GPS Protocol

Device types > Device protocols

A

LoRa

GPS Protocol

e.g. My protocol description

ID

6738225

Date created

Mar 6, 2025, 1:08:20 AM

Last update

Mar 6, 2025, 1:08:20 AM

Fieldbus version

4

Message types

Source

Payload

Start bit

0

Number of bits

8

LoRa devices can send messages of different types with different encodings per type. Depending on the device, the type can be determined by looking either at the FPort parameter of a message (**Source: FPort**) or at the subset of the message payload itself (**Source: Payload**). Indicate where the type information starts in the payload (**Start bit**) and how long this information is (**Number of bits**).

Values

UNCATEGORIZED

Latitude

MESSAGE TYPE ID 1 START BIT 8 NUMBER OF BITS 32

Longitude

MESSAGE TYPE ID 1 START BIT 40 NUMBER OF BITS 32

+ Add value

Save

Using custom decoding

The Lorient agent also supports the decoding functionality by plugging in the custom microservice. Refer to [LPWAN custom protocols](#) for further details.

ASSIGNING THE LORIENT LORA DEVICE PROTOCOL

Once the Lorient LoRa device is available in the Cumulocity platform, you must assign a device protocol from the **LPWAN** tab.

LPWAN

Page 266 of 320

The screenshot shows the 'Loriot Device DABB121243435678' configuration page. The left sidebar contains a 'DEVICE MANAGEMENT' menu with options like Home, Devices, Registration, All devices, Map, Availability, Overviews, Groups, Device types, SmartREST templates, Device protocols, and Management. The main content area has a sub-menu on the left with 'Info', 'Measurements', 'Alarms', 'Control', 'Availability', 'Events', 'LPWAN' (selected), 'Shell', and 'Identity'. The 'LPWAN configuration' panel displays 'Current device protocol' as 'test' and 'Current connection' as 'test3'. Below these are dropdown menus for 'Select new device protocol' and 'Select new connection', and a green 'Save' button.

Select the respective protocol from the dropdown list and click **Apply**. If successfully applied, the message "Device protocol set" will show up.

SENDING OPERATIONS

If the device supports sending hexadecimal commands, you can send them using shell operations. Note that these commands are not serial monitor commands.

In order to send an operation, navigate to the device you want to send an operation to in the Device Management application under **All devices** and switch to the **Shell** tab.

In the following screenshot you can find some examples of a device protocol's predefined commands and their format:

Select a predefined command

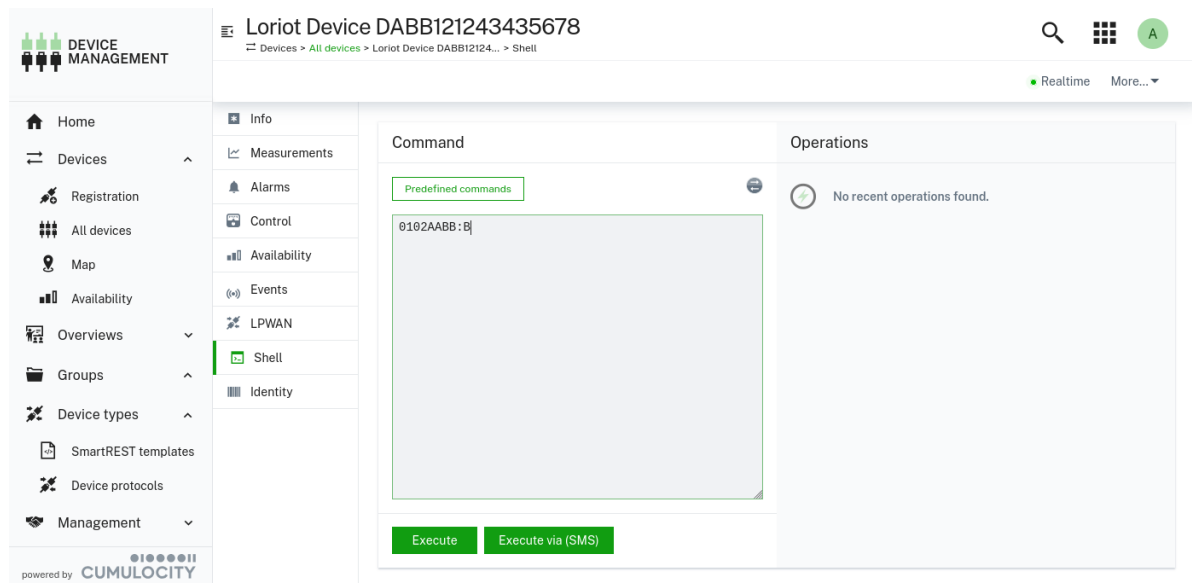
The dialog box titled 'Select a predefined command' features a search bar with the placeholder text 'Search command by name or text' and a magnifying glass icon. To the right of the search bar is a green button labeled '+ New template'. Below the search bar is a list of predefined commands, each with a terminal icon, a name, a description, and a 'Default' button. The commands listed are:

- set config**: { "breakpoint": true, "selfadapt": true, "openoff": true, "alreport": true, "pos": 0, "hb": 0 } (Default button: set config)
- position request**: position request (Default button: position re...)
- register request**: register request (Default button: register re...)
- device request**: device request (Default button: device req...)

 At the bottom of the dialog are two large buttons: 'Cancel' and 'Use'.

Enter the shell command or view/edit the predefined command in the **Command** field.

If you enter the command without defining a port, it will be sent to the default target port (that is, port 1) of the device. If you enter the command and define a port (format "command:port"), it will be sent to the specified target port instead of the default port.



Click **Execute**. The operation is sent to the device. The timing depends on the Lorient platform.

The status of the operation is set to **SUCCESSFUL** when the operation has successfully been sent to the Lorient platform. The status of the operation is set to **FAILED** when a problem occurred with the validation of the command or after the operation has been sent to the Lorient platform.

UPLINK MESSAGE PROCESSING

On receiving an uplink message, the Cumulocity platform creates the following measurements and events, and updates the corresponding device managed object.

- **Unprocessed data** - An event of type `c8y_LorientUplinkRequest` is created with the unprocessed data.
- **Position** - The `c8y_Position` fragment of the device managed object is updated to capture the latitude, longitude, altitude and accuracy information of the device. Also, an event is created with the position information.
- **Spreading factor** - The `c8y_SpreadingFactor` fragment of the device managed object is updated to capture the spreading factor of the device.
- **Signal strength** - A measurement is created with RSSI and SNR values of the device signal strength.

TROUBLESHOOTING

Device registration

No LoRa device registered in Cumulocity after configuring the Lorient agent endpoint in the Lorient Network Server account

The Lorient agent verifies if the user has appropriate permissions. Check whether the user configured in the Lorient Network Server has assigned the Lorient admin role.

Make sure that the **Gateway Information** is enabled in the Lorient Network Server since the Lorient agent only processes "gw" messages.

Device type error warning

The warning message "Device type error" shows up in the log if no device protocol has been assigned to the device. To assign a device protocol refer to the section [Assign the Lorient LoRa device protocol](#).

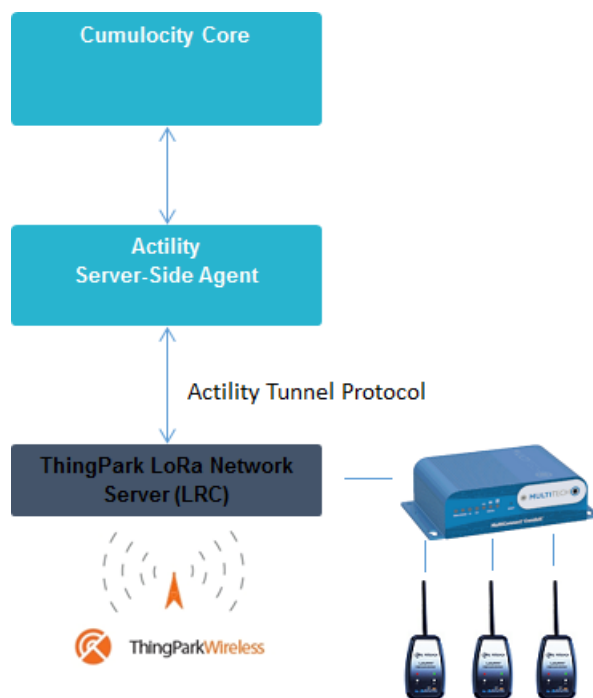
ACTIVITY LORA

INTRODUCTION

Cumulocity can interface with LoRa devices through Activity's ThingPark Wireless, Enterprise or Community edition. You can:

- Provision and deprovision LoRa devices easily using the Cumulocity Device Management application. No interaction in the ThingPark user interface is required.
- Decode upstream payload packets using a web-based user interface.
- Debug and postprocess raw device data through Cumulocity events.
- Send downstream data to the device using Cumulocity operations.
- Make use of existing Cumulocity features with LoRa devices, for example, connectivity monitoring, device management, data visualization with dashboards, real-time analytics and more.

The following illustration gives an overview of the Cumulocity Activity LoRa integration.



i INFO

Your subscription must include this feature. If you do not see the functionality described in this document please contact [product support](#).

CONFIGURING MULTIPLE THINGPARK ACCOUNT CONNECTIONS

Before using LoRa devices with Cumulocity, you must configure your ThingPark account details in the Administration application.

To add a new connection

Click the **Connectivity** tab in the **Settings** menu to create, edit, delete or update multiple Activity connections.

If you select **Connectivity** for the first time, you are asked to create a connection. Click **Add Connection**.

Enter the following information:

Setting name	Notes
Name	The name of the Activity connection being created.
Description	The description of the Activity connection being created.
Activity ThingPark URL	Base URL of the corresponding Activity DX API being used.
Profile ID	Your ThingPark account profile identifier.
Application server ID	Application server ID for TLS security between ThingPark platform and agent. Optional field. Leave empty to disable security. If enabled, the agent generates a token for all uplink and down-link messages.
Application server key	Application server private key for TLS security between the ThingPark platform and agent for uplink and downlink communications. Value should be in hex and 16 bytes. Optional field. Leave empty to disable security. If enabled, the agent generates a token for all uplink and down-link messages.
Admin API version	Version that the ThingPark admin API uses. By-default set to "latest".
Core API version	Version that the ThingPark core API uses. By-default set to "latest".
Username	Your ThingPark account username.
Password	Your ThingPark account password.
Connection type	The ThingPark account type that is being used (Enterprise or Wireless).

Environment-specific information

The following settings vary depending on your ThingPark environment:

ThingPark community:

- URL: <https://community.thingpark.io/thingpark/dx/>
- Profile ID: community-api
- Application server ID and key: Leave empty (HTTPS used internally)
- Connection type: Choose Enterprise

Other environments (Enterprise/Wireless):

- URL: Contact ThingPark support
- Profile ID: Contact ThingPark support
- Application server ID and key: Look up in your application server profile in ThingPark or contact ThingPark support
- Connection type: Choose based on your environment (Enterprise or Wireless)

! IMPORTANT

Do not use the same ThingPark login (username and password) for other tenants. The profile ID, username and password are used to retrieve an access token to send further requests to the ThingPark platform. It is possible to renew the access token by replacing the account credentials for a particular connection.

Click **Save**. If you have entered the correct information, you see the message “Connection created”.

To add another connection, click **Add Connection** and follow the steps above.

To update a connection

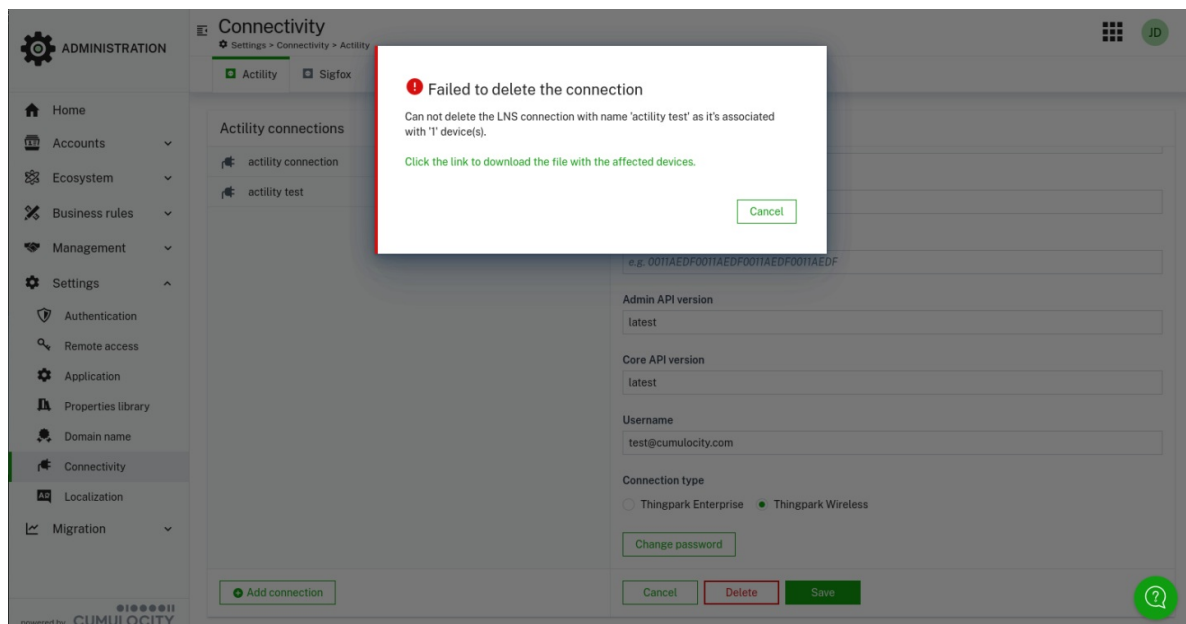
Select the connection to be updated, make your edits, and save the connection.

If there are devices associated with the connection, an error message will appear, stating “Can not update the LNS Connection with `<name of LNS Connection>` as it's associated with `<number of devices>`”. Click the link to download the file with the details of the associated devices: `/service/<agent-context-path>/lms-connection/<lms-connection-name>/device`”.

To delete a connection

Select the connection to be deleted and click **Delete**.

If there are devices associated with the connection, an error message will appear, stating "Can not delete the LNS Connection with **<name of LNS Connection>** as it's associated with **<number of devices>**". Click the link to download the file with the details of the associated devices: `/service/<agent-context-path>/lms-connection/<lms-connection-name>/device` ".



CREATING DEVICE PROTOCOLS

To process data from LoRa devices, Cumulocity needs to understand the payload format of the devices. Mapping a payload data to

Cumulocity data can be done by creating a LoRa device protocol.

During the [device registration](#), you can associate this device protocol. The received uplink callbacks for this device with a hexadecimal payload will then be mapped to the ones you have configured in your device protocol.

The device protocol assigned during device registration can be changed from the **LPWAN** tab in the device details page.

Activity_ABCDEF0123498765

Devices > [All devices](#) > Activity_ABCDEF012349... > LPWAN

<div><div>Info</div><div>Measurements</div><div>Alarms</div><div>Control</div><div>Availability</div><div>Events</div><div>LPWAN</div><div>Shell</div><div>Identity</div></div>	<div><h3>LPWAN configuration</h3><table><tr><td>Current device protocol test4</td><td>Current connection actility test</td></tr><tr><td><div>Select new device protocol</div></td><td><div>Select new connection</div></td></tr><tr><td colspan="2"><div>Save</div></td></tr></table></div>	Current device protocol test4	Current connection actility test	<div>Select new device protocol</div>	<div>Select new connection</div>	<div>Save</div>	
Current device protocol test4	Current connection actility test						
<div>Select new device protocol</div>	<div>Select new connection</div>						
<div>Save</div>							

INFO


Device protocol mapping only supports decoding for fixed byte positions based on the message type. The length for the device payload parts, which is set in the **Number of bits** field, can be maximum 32 bits (4 bytes).

In order to create a device protocol, navigate to the Device Management application and select **Device protocols** in the **Device types** menu in the navigator. You can either import an existing device protocol or create a new one or use the device protocols created by an LPWAN custom codec microservice.

Importing a predefined device protocol

In the **Device protocols** page, click **Import**.

Select the predefined device type, for example "LoRaWAN Demonstrator" or upload from a file. Click **Import**.



IMPORT DEVICE PROTOCOL

1 SELECT DEVICE PROTOCOL

Select from predefined

Adeunis LoRaWan Demonstrator

Or load it from a file

Select file to upload...

2 SAVE WITH THE FOLLOWING NAME

Name

Adeunis LoRaWan Demonstrator


Cancel

Import

Alternatively, you may also load the device protocol from a file and import it.

Creating a new device protocol


In the **Device protocols** page, click **New device protocol** in the top menu bar. The following window will open:



ADD DEVICE PROTOCOL

Select one of the available options

Find your protocol in the [user documentation](#) to get more information.

 LoRa

Cancel

Select **LoRa** as the device protocol type, provide a name for it and click **Create**.

Under **Message types**, specify the message types. LoRa devices can send messages of different types with different encodings per type.

Select the way the message type is encoded in the **Source** dropdown box:

- **FPort**: if the message type can be determined by looking at the FPort parameter of a message.
- **Payload**: if the message type can be determined by looking at the subset of the message payload itself.

In the following example payload structure, the first byte indicates the message type source (as highlighted).

Message type source

0	1	2	3	4	5	6	7	8	9	10	11
Code	Status	PAYLOAD									
0x01	Cf Status	Channel 1 Type	Measure sensor 1 (LSB First)			Channel 2 Type	Measure sensor 2 (LSB first)			X	X
0x03	Cf Status	Device Type	Transmit period	Channel On/Off	Channel 1 Type	Channel 2 Type	NA (0x00)	Switch Value	X	X	

In the user interface you can enter this type of message type source information as follows: In the **Start bit** field, indicate where the message type information starts in the payload and in the **Number of bits** field, indicate how long this information is, for example start bit = "0" and number of bits = "8".

LoRa Test Protocol

Device types > Device protocols

Search

Grid

A

LoRa Test Protocol

e.g. My protocol description

LoRa

ID

3438224

Date created

Mar 6, 2025, 12:37:40 AM

Last update

Mar 6, 2025, 12:37:40 AM

Fieldbus version

4

Message types

Source

Payload

Start bit ?

0

Number of bits ?

8

LoRa devices can send messages of different types with different encodings per type. Depending on the device, the type can be determined by looking either at the FPort parameter of a message (**Source: FPort**) or at the subset of the message payload itself (**Source: Payload**). Indicate where the type information starts in the payload (**Start bit**) and how long this information is (**Number of bits**).

Values

No values defined.

Click **Add value** to create the value configuration.



LoRa Test Protocol

e.g. My protocol description

ID	3438224
Date created	Mar 6, 2025, 12:37:40 AM
Last update	Mar 6, 2025, 12:37:40 AM
Fieldbus version	4

Message types

Source

Payload

Start bit

0

Number of bits

8

LoRa devices can send messages of different types with different encodings per type. Depending on the device, the type can be determined by looking either at the FPort parameter of a message (**Source: FPort**) or at the subset of the message payload itself (**Source: Payload**). Indicate where the type information starts in the payload (**Start bit**) and how long this information is (**Number of bits**).

Values



No values defined.

Add value

Save

In the upcoming window, configure the relevant values as shown in this example.

New value window part 1

MESSAGE TYPE ?

Message ID ?

1

GENERAL

Name

Channel type1

Display category

Sensor measure data

VALUE SELECTION ?

Start bit

16

Number of bits

8

VALUE NORMALISATION ?

Multiplier

1

Offset

0

Unit

e.g. u

OPTIONS

☐ Signed
☐ Packed decimal
☐ Little-endian

FUNCTIONALITIES

☐ Send measurement

☐ Raise alarm

☒ Send event

Event type

c8y_MeasureSensor

Event text

Measurement sensor 1 event

Event fragment

c8y-MeasureSensor

Event property

sensor1 Type

☐ Update managed object

Cancel

Save

The value configuration maps the value in the payload of a message type to the Cumulocity data.


Under **Message type**, configure the **Message ID** according to your device message specification and map it to the Cumulocity data. The message ID is the numeric value identifying the message type. It will be matched with the message ID found in the source specified on the device protocol main page (that is, Payload or FPort). The message ID must be entered in decimal numbers (not hex).


In this example payload structure the message ID is "1".

Message ID

0	1	2	3	4	5	6	7	8	9	10	11
Code	Status	PAYLOAD									
0x01	Cf Status	Channel 1 Type	Measure sensor 1 (LSB First)			Channel 2 Type	Measure sensor 2 (LSB first)			X	X
0x03	Cf Status	Device Type	Transmit period		Channel On/Off	Channel 1 Type	Channel 2 Type	NA (0x00)	Switch Value	X	X

New value

MESSAGE TYPE 

Message ID 

1


GENERAL

Name

Channel type1

Display category

Sensor measure data


VALUE SELECTION 

Start bit

16

Number of bits

8

VALUE NORMALISATION 

Multiplier

1


Offset


0


Unit

e.g. u

OPTIONS

☐ Signed 

☐ Packed decimal 

☐ Little-endian 

Cancel

Save

Under **General**, specify a name for the value and the category under which it will be displayed in the values list. The associated name for this value will be displayed under the **Display category** given.

Under **Value selection**, define from where the value should be extracted. In order to do so, indicate where the value information starts in the **Start bit** field and how long this information is in the **Number of bits** field. The maximum value for the number of bits is 32 bits (4 bytes).

In this example the "Channel 1 Type" information starts in byte 2 (that means, start bit = "16") and is 1 byte long (that means, number of bits = "8").

Value selection:
start bit and number of bits

0	1	2	3	4	5	6	7	8	9	10	11
Code	Status		PAYLOAD								
0x01	Cf Status	Channel 1 Type	Measure sensor 1 (LSB First)			Channel 2 Type	Measure sensor 2 (LSB first)			X	X
0x03	Cf Status	Device Type	Transmit period		Channel On/Off	Channel 1 Type	Channel 2 Type	NA (0x00)	Switch Value	X	X

New value

MESSAGE TYPE ?

Message ID ?

1

GENERAL

Name
Display category

Channel type1

Sensor measure data

VALUE SELECTION ?

Start bit

16

Number of bits

8

VALUE NORMALISATION ?

Multiplier

1

Offset

0

Unit

e.g. u

OPTIONS

☐ Signed ?
☐ Packed decimal ?
☐ Little-endian ?

Cancel

Save

The hexadecimal value is converted to a decimal number and afterwards a “value normalisation” is applied.

Under **Value normalisation** define how the raw value should be transformed before being stored in the platform and enter the appropriate values for:

- **Multiplier:** This value is multiplied with the value extracted from the **Value selection**. It can be decimal, negative and positive. By default it is set to 1.
- **Offset:** This value defines the offset that is added or subtracted. It can be decimal, negative and positive. By default it is set to 0.
- **Unit (optional):** A unit can be defined which is saved together with the value (for example temperature unit “C” for degree Celsius).

For detailed information on how to decode the payload, refer to the documentation of the device.

i INFO

“Little endian” support to decode the payload has been added.

Under **Options**, select one of the following options, if required:

- **Signed** - if the value is a signed number.
- **Packed decimal** - if the value is BCD encoded.

Under **Functionalities**, specify how this device protocol should behave:

- **Send measurement**: Creates a measurement with the decoded value.
- **Raise alarm**: Creates an alarm if the value is not equal to zero.
- **Send event**: Creates an event with the decoded value.
- **Update managed object**: Updates a fragment in a managed object with the decoded value.

You can also have a nested structure with several values within a measurement, event or managed object fragment. In case of a measurement all the properties of the same type will be merged to create a nested structure. In case of an event or a managed object all the properties with the same fragment are merged to create a nested structure. Also refer to the [example](#) of a nested structure for a "Position" device protocol below.

Click **OK** to add the values to your device protocol.

LoRa Test Protocol

Device types > Device protocols

A

LoRa

LoRa Test Protocol

e.g. My protocol description

ID

3438224

Date created

Mar 6, 2025, 12:37:40 AM

Last update

Mar 6, 2025, 12:37:40 AM

Fieldbus version

4

Message types

Source

Payload

Start bit

0

Number of bits

8

LoRa devices can send messages of different types with different encodings per type. Depending on the device, the type can be determined by looking either at the FPort parameter of a message (**Source: FPort**) or at the subset of the message payload itself (**Source: Payload**). Indicate where the type information starts in the payload (**Start bit**) and how long this information is (**Number of bits**).

Values

SENSOR MEASURE DATA

Channel type1

MESSAGE TYPE ID 1 START BIT 16 NUMBER OF BITS 8

+ Add value

Save

After clicking **Save**, your device protocol is created with the values you defined.

[Example with single property](#)

The following image shows an example for a message which sends a measurement when the battery level changes.

New value window part 1

LPWAN

Page 281 of 320

New value

MESSAGE TYPE ?

Message ID ?

GENERAL

Name

Display category

VALUE SELECTION ?

Start bit

Number of bits

VALUE NORMALISATION ?

Multiplier

Offset


Unit

OPTIONS

- ☐ Signed ?
- ☐ Packed decimal ?
- ☐ Little-endian ?


New value window part 2

FUNCTIONALITIES ?

☒  Send measurement ?

Measurement type

Measurement series

☐  Raise alarm ?

☐  Send event ?

Example with nested structure


The following image shows an example of a nested structure for a device protocol reporting the current position of a GPS device. The display category is named "Position" and contains values for longitude and latitude.

The message ID should be the same for all the values. Enter the rest of the parameters according to the instructions above. Enter "c8y_Position" in the **Managed object fragment** field and create a new value for each: longitude and latitude.


New value window, Longitude

FUNCTIONALITIES ?


☐

 Send measurement ?


☐

 Raise alarm ?

☐

 Send event ?

☒

 Update managed object ?

Managed object fragment

c8y_Position

Managed object property ?

lng


Cancel

Save


New value window, Latitude

FUNCTIONALITIES ?


☐

 Send measurement ?


☐

 Raise alarm ?

☐

 Send event ?

☒

 Update managed object ?

Managed object fragment

c8y_Position

Managed object property ?

lat

Cancel

Save

This will be the result:

GPS Protocol

Device types > Device protocols

Q

A

LoRa

GPS Protocol

e.g. My protocol description

ID

6738225

Date created

Mar 6, 2025, 1:08:20 AM

Last update

Mar 6, 2025, 1:08:20 AM

Fieldbus version

4

Message types

Source

Payload

Start bit

0

Number of bits

8

LoRa devices can send messages of different types with different encodings per type. Depending on the device, the type can be determined by looking either at the FPort parameter of a message (Source: FPort) or at the subset of the message payload itself (Source: Payload). Indicate where the type information starts in the payload (Start bit) and how long this information is (Number of bits).

Values

UNCATEGORIZED

Latitude

MESSAGE TYPE ID 1 START BIT 8 NUMBER OF BITS 32

Longitude

MESSAGE TYPE ID 1 START BIT 40 NUMBER OF BITS 32

+ Add value

Save

Using custom decoding/encoding

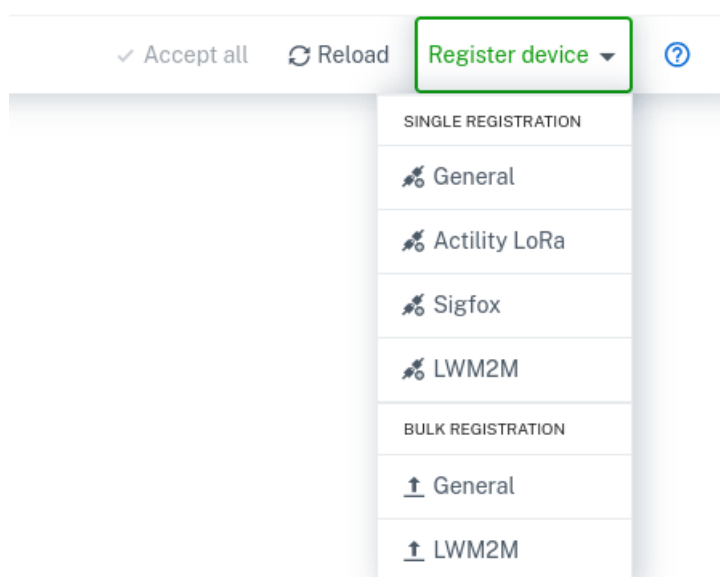
The Activity agent also supports the decoding/encoding functionality by plugging in the custom microservice. Refer to [LPWAN custom protocols](#) for further details.

REGISTERING ACTILITY LORA DEVICES

To register a LoRa device in Cumulocity navigate to **Devices > Registration** in the Device Management application. Click **Register device** at the top right, and select **Single device registration > Actility LoRa** from the dropdown.

LPWAN

Page 284 of 320



Cumulocity fully supports the LoRa device provisioning with Over-the-Air Activation (OTAA) which is the preferred and most secure way to connect with the LoRa network. If Activation by Personalization (ABP) is required to be used, refer to [LoRa device registration with ABP](#).

In the next window fill in the required information:

- **Connection:** Lists all configured Activity connections in the tenant. The following device profile and connectivity plan option is populated based on the selected Activity connection.
- **Device profile:** Select the Activity Thingpark device profile from the dropdown list that matches the device that you are registering.

The Activity ThingPark device profile allows to manage multi-RF profiles, ensures different LoRaWAN class compatibility (A, B or C) and allows application payload decoding for easy third-party application integration.

- **Device protocol:** Select the appropriate device protocol from the dropdown list. For more information on how to create a device protocol refer to [Creating device protocols](#).
- **Device EUI:** This is the unique identifier for the device. It is a 16 character (8 byte) long hexadecimal number. You can find it on the device itself.
- **Application EUI:** This is a global application ID in the IEEE EUI64 address space that uniquely identifies the application provider of the device. It is a 16 character (8 byte) long hexadecimal number. There can be only one application EUI for a tenant but multiple tenants can have the same application EUI.
- **Application key:** This is an AES-128 application key specific for the device that is assigned to the device by the application owner and is responsible to encrypt. The application key is a 32 character (16 byte) long hexadecimal number. JOIN communication. You can find this key on the device itself.
- **Connectivity plan:** Select the appropriate connectivity plan from the dropdown list.



Register a single Actility device

Connection

actility connection

Device profile

Micro Tracker

Device protocol

LANSITEC : Asset Tracker

Device EUI

12345BCD45123456

Application EUI

703456F345234523

Application key

703456F345234523703456F345234523

Connectivity plan

dev-ope Testing CP

Cancel

Register

Click **Register** to submit the device registration request and create the device.

You can verify that the device is really connected by checking that events are actually coming in. You can do so by clicking on a device and opening the **Events** tab. All events related to this device are listed here.

The provision status is shown under **Device data** in the **Info** tab of the device.

DEVICE DATA

ID	19641
NAME	Actility_0018AA0405060708
TYPE	Adeunis LoRaWan Demonstrator
OWNER	ozge
LAST UPDATED	2018-10-09T12:31:07.572Z
LPWAN DEVICE	
PROVISIONED	true
CREATION TIME	2018-10-09T12:31:07.572Z

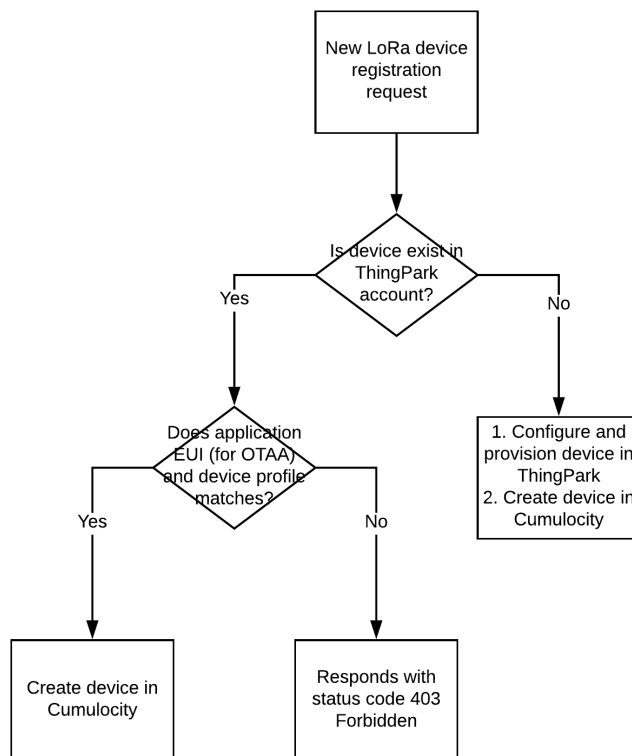
For more information on viewing and managing your connected devices, also refer to the [Device Management application](#).

In order to migrate the device from one LNS Connection to another, the device must be re-registered.

1. Navigate to the **LPWAN** tab of the device.
2. Click the **Provider connection** dropdown. A prompt will appear stating that in order to migrate the device from one LNS connection to another, you must re-register the device.
3. Click **Re-Register**.

You are directed to the device registration page where you can perform the re-registration by following the steps above and selecting the desired LNS connection.

LoRa device registration process



A device is created based on the above workflow.

First it is checked, if the device already exists. If no device exists with the same device EUI in the ThingPark account, the device is first provisioned on the ThingPark platform and then created on the Cumulocity platform with a link to the device in the ThingPark platform. If the device exists in the ThingPark account, a validation will be applied to compare these devices based on application EUI (for OTAA activation) and device profile. If the validation is successful, the device is created only in Cumulocity with a link to the device in the ThingPark platform. If the validation fails, a failure message is shown and the device is not created in Cumulocity.

LoRa device registration with Activation by Personalization (ABP)

Activating the device by personalization is not recommended and not fully supported in Cumulocity LoRa device registration.

However, if you would like to create a device with this activation type in Cumulocity and use the LoRa features - such as sending operations to a device, deprovisioning a device and setting LoRa device protocol type with custom device protocol configuration - you must first provision the device in the ThingPark platform. Moreover you must create "AS Routing Profile" for Cumulocity using the destination <http://actility-server.cumulocity.com> as a "Third Party AS (HTTP)" and assign it to your devices manually. Afterwards, you can register this device using LoRa device registration. In this case, the **Application key** field in the LoRa device registration is invalid.

Limitations for LoRa devices created with general device registration

The general device registration for LoRa devices is no longer supported.

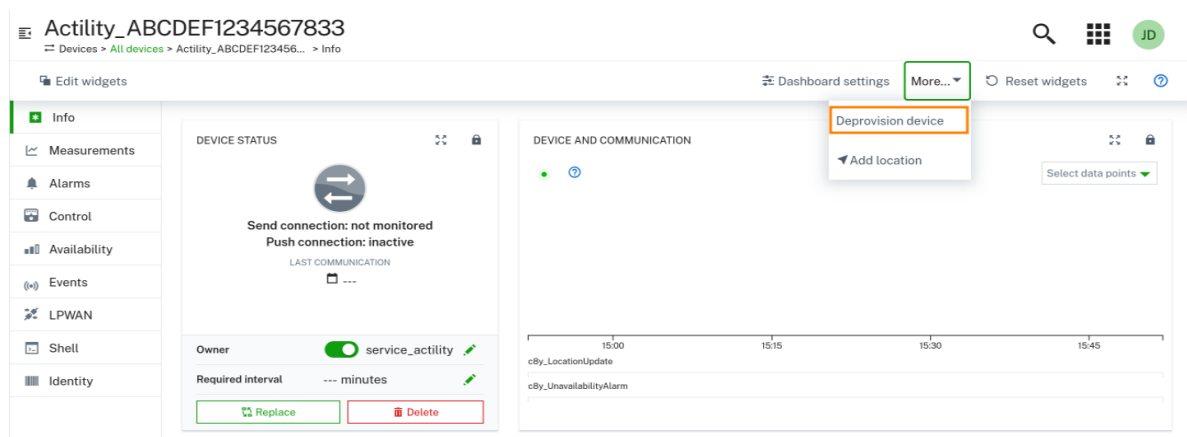
Existing LoRa devices that have been created in Cumulocity with the general device registration process have limitations. For those devices, it is not possible to send operations to the device, deprovision the device and set the LoRa device protocol type with custom device protocol configuration.

It is recommended to delete and re-register these devices using LoRa device registration to fully use the LoRa feature.

DEPROVISIONING LORA DEVICES

You can deprovision a LoRa device in the ThingPark platform. This means that the device is no longer connected to the network. Its history data is still available in Cumulocity, but the device is deleted in ThingPark.

To deprovision a device, navigate to the respective device in the Device Management application under **All devices**. Click **More** in the top right and select **Deprovision device**.



After confirming the deprovisioning, the device will be deprovisioned in ThingPark.

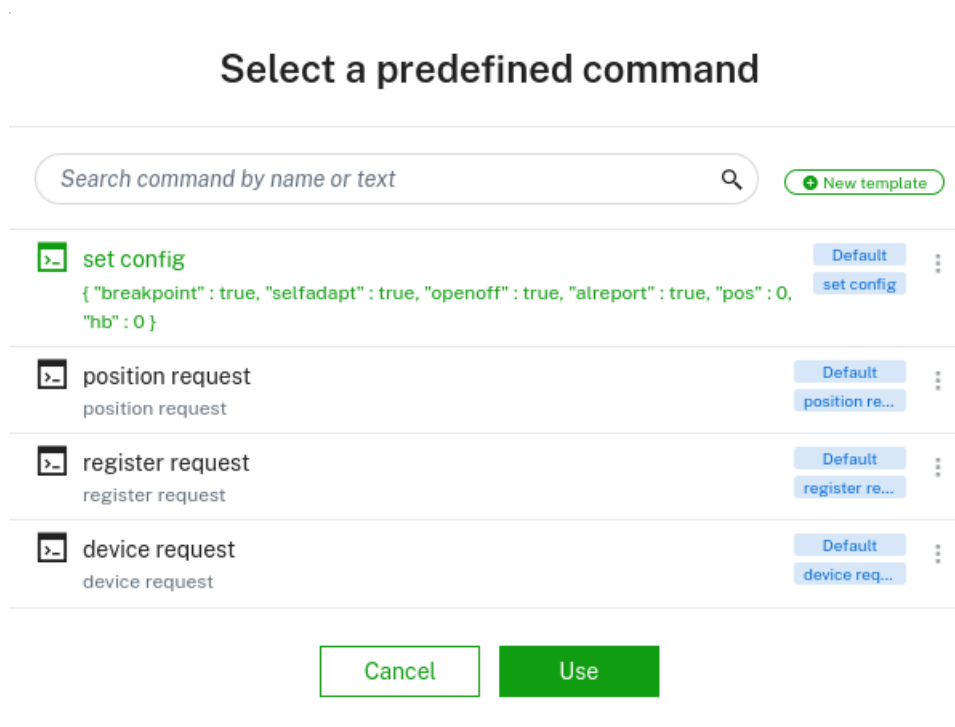
To provision the device again, the device must be deleted and re-registered using LoRa device registration.

SENDING OPERATIONS

If a LoRa device supports receiving hexadecimal commands, you can send them using shell operations. Note that these commands are not serial monitor commands.

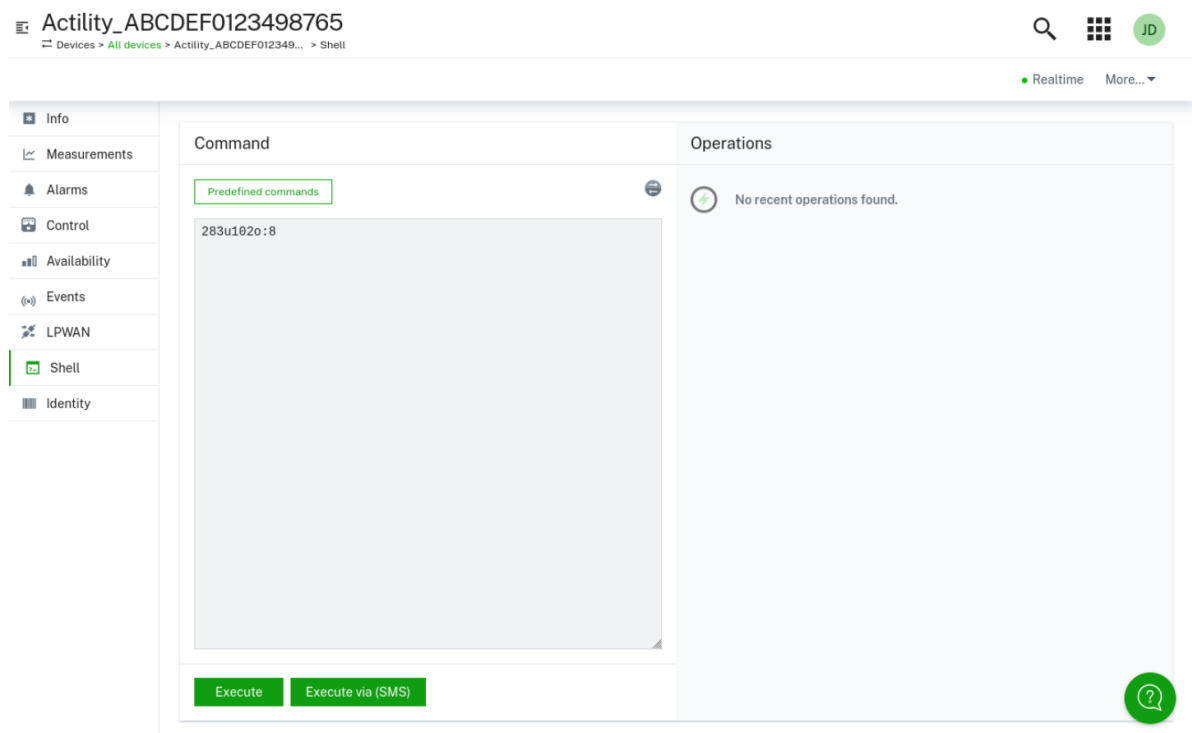
In order to send an operation, navigate to the device you want to send an operation to in the Device Management application under **All devices** and switch to the **Shell** tab.

In the following screenshot you can find some examples of a device protocol's predefined commands and their format.



Enter the shell command or view/edit the predefined command in the **Command** field.

If you enter the command without defining a port, it will be sent to the default target port (that is, port 1) of the device. If you enter the command and define a port (format "command:port"), it will be sent to the specified target port instead of the default port.



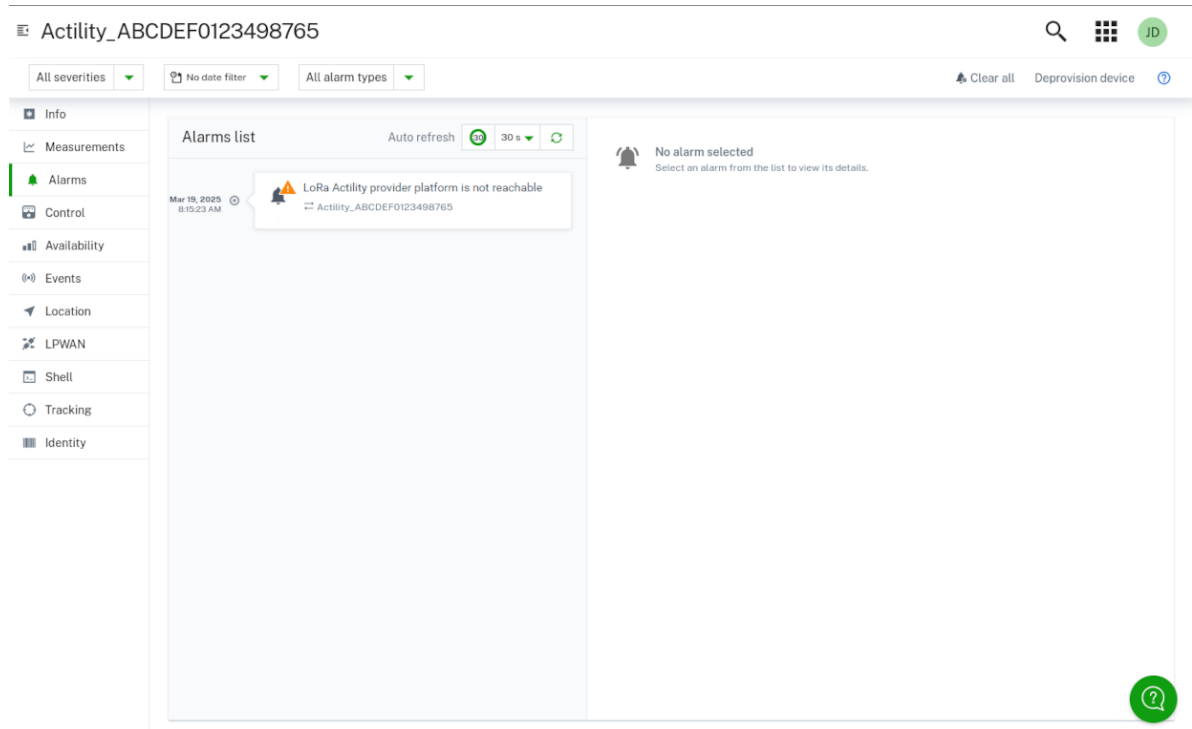
Click **Execute**. The operation is sent to the device. The timing depends on Activity ThingPark.

The status of the operation is set to **SUCCESSFUL** when the operation has successfully been sent to the ThingPark platform. The status of the operation is set to **FAILED** when a problem occurred with the validation of the command or after the operation has been sent to the

Thingpark platform.

THINGPARK API AVAILABILITY MONITORING

The ThingPark API is monitored, and if it is not reachable, an alarm is created to notify all subscribed tenants using this feature. The alarm is cleared right after the ThingPark API is reachable again.



UPLINK MESSAGE PROCESSING

On receiving an uplink message, the Cumulocity platform creates the following measurements and events, and updates the corresponding device managed object.

- **Unprocessed data** - An event of type `c8y_ActivityUplinkRequest` (this type is based on the `event.uplink.type` set in the configuration file) is created with the unprocessed data.
- **Position** - The `c8y_Position` fragment of the device managed object is updated to capture the latitude, longitude, altitude and accuracy information of the device. Also, an event is created with the position information.
- **Spreading factor** - The `c8y_SpreadingFactor` fragment of the device managed object is updated to capture the spreading factor of the device.
- **Signal strength** - A measurement is created with RSSI and SNR values of the device signal strength.

TROUBLESHOOTING

Device registration

Access to device denied

This warning message shows up when there already exists a provisioned device in ThingPark with the same device EUI used for device registration and the validation comparing those devices based on application EUI (for OTAA activation) and device profile has failed.

To resolve this, provide the correct application EUI from the [Connectivity](#) application and device profile and try again.

No LoRa provider settings found

This warning message shows up when there are no credentials set up for the ThingPark account. To resolve this click **Settings** to navigate to the Administration application where the connections are configured.



! Could not get connectivity plans from the LoRa platform. Verify the ThingPark credentials in the Administration app under [Settings](#).

Close

To resolve this, refer to [Configuring multiple ThingPark account connections](#).

Getting device profiles from provider failed

This warning message shows up when the tenant's access token to Thingpark becomes invalid. Invalidation of the token might happen when the same ThingPark credentials are used for another tenant.

This issue can be solved by reconfiguring the Activity ThingPark credentials to renew the access token. Refer to [Configuring multiple ThingPark account connections](#) for reconfiguration of the credentials.

No device protocols configured

This warning message shows up when no LoRa device protocol exists to be used for device registration. To resolve this, click **Device protocols** to navigate to the **Device protocols** page where the protocols are configured.



! No device protocols configured. Create a LoRa device protocol in [Device protocols](#).

Close

To resolve this, configure at least one device protocol in the [Device database](#).

No connectivity plans with free slots available

This warning message shows up when the connectivity plan in Activity ThingPark has reached the limit for the device count.

To resolve this, either contact ThingPark on the device quota limits for your connectivity plans or remove unused devices from ThingPark and retry registering the device in Cumulocity.

Connectivity

Authorization to the LoRa platform failed

This warning message shows up if a provided profile ID, username or password is invalid.

To resolve this, provide correct credentials and try again.

Authentication to the Activity platform failed. Check if the base URL is correct

To resolve this, provide a correct URL and try again.

Authentication to the Activity platform failed with status code '400'. Check if the profile ID is correct

To resolve this, provide a correct profile ID and try again.

Authentication to the Activity platform failed with status code '401'. Check if the credentials are correct

To resolve this, provide a correct username and password and try again.

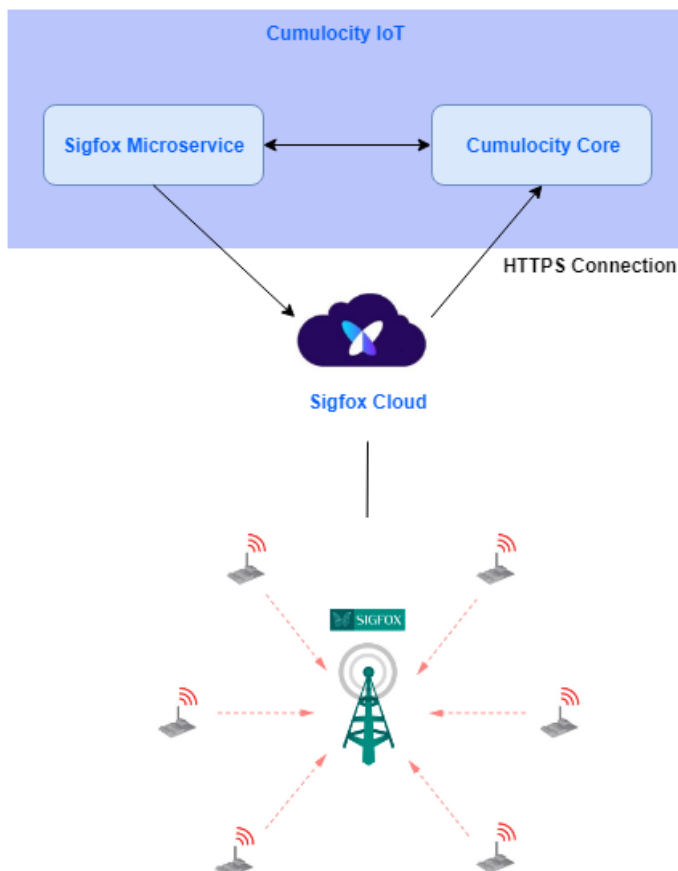
SIGFOX

INTRODUCTION

Cumulocity can interface with Sigfox devices through the Sigfox Cloud. You can:

- Provision Sigfox devices easily using the Cumulocity Device Management application.
- Decode upstream payload packets using a web-based user interface.
- Debug and post-process raw device data through Cumulocity events.
- Send downstream data to the device using Cumulocity operations.
- Make use of existing Cumulocity features with Sigfox devices, for example: connectivity monitoring, device management, data visualization with dashboards, real-time analytics and more.

The following illustration grants you a quick overview of the Cumulocity Sigfox integration:



✓ REQUIREMENTS

To be able to use the Sigfox agent, your tenant must be subscribed to the application sigfox-agent. In case of any issues please contact [product support](#).

MANAGING THE CONNECTIVITY SETTINGS

Before you register a device, you must configure the Sigfox Cloud credentials in the **Connectivity** page in the Administration application. You must set up these Sigfox Cloud credentials in Sigfox.

Before you create API access to Cumulocity, you must have an "Associated user" which is added to the Cumulocity group in Sigfox Cloud and has the following profiles:

- Customer [R]
- Device Manager [W]

❗ IMPORTANT

Without the profiles described below, the required Sigfox API access can not be set up.

Step 1

If you already have an associated user make sure it has the profiles mentioned below and proceed to step 2.

The group name is not constrained. "Cumulocity" is used as a sample group name throughout the remaining steps.

First, enter into your Sigfox Cloud account and create a new user. Add the user to the group and select the "Customer [R]" and "Device Manager [W]" profiles.

New user

User information

First name

Last Name

Email

Position

Timezone

Profiles

Group

Profiles Select the profiles below

Info	Name	Select
?	BILLING_API	<input type="checkbox"/>
?	CUSTOMER [R]	<input type="checkbox"/>
?	CUSTOMER [W]	<input checked="" type="checkbox"/>
?	DEVICE MANAGER [R]	<input type="checkbox"/>
?	DEVICE MANAGER [W]	<input checked="" type="checkbox"/>
?	DEVICE_CREATE_GRNY	<input type="checkbox"/>
?	DEVICE_MESSAGES [R]	<input type="checkbox"/>
?	LIMITED_ADMIN	<input type="checkbox"/>
?	ONLINE_HELP	<input type="checkbox"/>
?	OPT_DEVICETYPE_ORDER [W]	<input type="checkbox"/>
?	OPT_DEVICETYPE_READ	<input type="checkbox"/>
?	OPT_NOC_ENHANCED	<input type="checkbox"/>
?	OPT_SERVICE_MAP	<input type="checkbox"/>
?	PUBLIC_SERVICE_MAP	<input type="checkbox"/>

Step 2

After creating an “Associated user” with the proper group and profiles navigate to the **Groups** page. In the **API access** tab, create a new entry and add the following profiles:

- Customer [R]
- Device Manager [W]

The screenshot shows the Sigfox administration interface. On the left is a dark purple sidebar with a menu containing: INFORMATION, ASSOCIATED USERS, ASSOCIATED DEVICE TYPES, ASSOCIATED CONTRACTS, EVENT CONFIGURATION, and API ACCESS (which is highlighted). The top navigation bar includes links for DEVICE, DEVICE TYPE, USER, GROUP, and BILLING. The main content area is titled 'Api access 'Cumulocity' - List'. It displays a single entry for 'DeviceManagement' with a refresh icon, a link icon, and a close icon. Below the entry title, there is a note: 'To access your group's device types and devices programmatically, please refer to the API documentation.' The entry details are as follows:

- Login:** [REDACTED]
- Password:** [REDACTED]
- Timezone:** UTC
- Creation date:** 2017-04-24 11:28:45
- Created by:** Lars Stuke
- Last edition date:** 2017-04-24 11:28:45
- Last edited by:** Lars Stuke

Step 3

After the API access entry has been created, you can connect your Sigfox Cloud account to Cumulocity via the **Connectivity** page in the Administration application. Navigate to the **Connectivity** page and switch to the **Sigfox provider settings** tab.

The **Connectivity** tab facilitates creating, updating, and deleting multiple Sigfox connections.

The following information must be provided in order to create a connection:

- **Name:** Name of the Sigfox connection being created.
- **Description:** Description of the Sigfox connection being created.
- **Login:** The login token is located in the API access entry in the Sigfox Cloud.
- **Password:** The password token is located in the API access entry in the Sigfox Cloud next to **Password**.
- **Parent Group ID:** This ID is written in your URL when you are logged into your Sigfox account and you have selected the “Cumulocity” group. For example, <https://backend.sigfox.com/group/9823ruj29j9d2j9828hd8/info>.
- **Base URL:** URL that points to the Sigfox Cloud account.

INFO

The group name in the screenshot below is only an example. It does not necessarily have to be “Cumulocity”.

https://backend.sigfox.com/group/9823ruj29j9d2j9828hd8/info

sigfox

DEVICE DEVICE TYPE USER **GROUP** BILLING

INFORMATION

ASSOCIATED USERS

ASSOCIATED DEVICE TYPES

ASSOCIATED CONTRACTS

EVENT CONFIGURATION

API ACCESS

Group 'Cumulocity' - Information

Type: Basic

Name: Cumulocity

Description: nc

Timezone: Europe/Berlin

ADMINISTRATION

Accounts

Ecosystem

Business rules

Management

Settings

Authentication

Remote access

Application

Properties library

Domain name

Connectivity

Localization

Migration

powered by **CUMULOCITY**

Connectivity

Settings > Connectivity > Sigfox

Activity Sigfox

Sigfox connections

sigfox connection	
Name	sigfox connection
Description	e.g. This connection has a built-in functionality to...
URL	https://backend.sigfox.com/api
Parent group ID	58c1793b9e93a15370f71caa
Username	5b991e27c563d654ab2ce4d4
	Change password
	Add connection Cancel Delete Save

Click **Save**. If you have entered the correct information, the message "Credentials successfully saved" will be displayed.

To add another connection, click **Add Connection** and follow the steps above.

To update a connection

Select the connection to be updated, make your edits, and save the connection.

If there are devices associated with the connection, an error message will appear, stating "Can not update the LNS Connection with <name of LNS Connection> as it's associated with <number of devices> ". Click the link to download the file with the details of the associated devices: `/service/<agent-context-path>/lms-connection/<lms-connection-name>/device` ".

ADMINISTRATION

- Accounts
- Ecosystem
- Business rules
- Management
- Settings
- Authentication
- Remote access
- Application
- Properties library
- Domain name
- Connectivity**
- Localization
- Migration

Connectivity

Settings > Connectivity > Sigfox

Activity Sigfox

Sigfox connections

Name
sigfox connection

Description
e.g. This connection has a built-in functionality to...

URL
https://backend.sigfox.com/api

Parent group ID
58c1793b9e93a15370f71caa

Username
5b991e27c563d654ab2ce4d4

[Change password](#)

[Add connection](#) [Cancel](#) [Delete](#) [Save](#)

powered by CUMULOCITY

To delete a connection

Select the connection to be deleted and click **Delete**.

If there are devices associated with the connection, an error message will appear, stating "Can not delete the LNS Connection with <name of LNS Connection> as it's associated with <number of devices>". Click the link to download the file with the details of the associated devices: `/service/<agent-context-path>/lms-connection/<lms-connection-name>/device`."

ADMINISTRATION

- Accounts
- Ecosystem
- Business rules
- Management
- Settings
- Authentication
- Remote access
- Application
- Properties library
- Domain name
- Connectivity**
- Localization
- Migration

Connectivity

Settings > Connectivity > Sigfox

Activity Sigfox

Sigfox connections

Name
sigfox connection

Description
e.g. This connection has a built-in functionality to...

URL
https://backend.sigfox.com/api

Parent group ID
58c1793b9e93a15370f71caa

Username
5b991e27c563d654ab2ce4d4

[Change password](#)

[Add connection](#) [Cancel](#) [Delete](#) [Save](#)

Failed to delete the connection

Can not delete the LNS connection with name 'sigfox connection' as it's associated with '3' device(s).

[Click the link to download the file with the affected devices.](#)

[Cancel](#)

powered by CUMULOCITY

Authentication to the Sigfox platform failed

Authentication to the Sigfox platform failed. Check if the Parent group ID and/or the credentials are correct.

To resolve this, provide a correct baseUrl or parent group ID or username or password and try again.

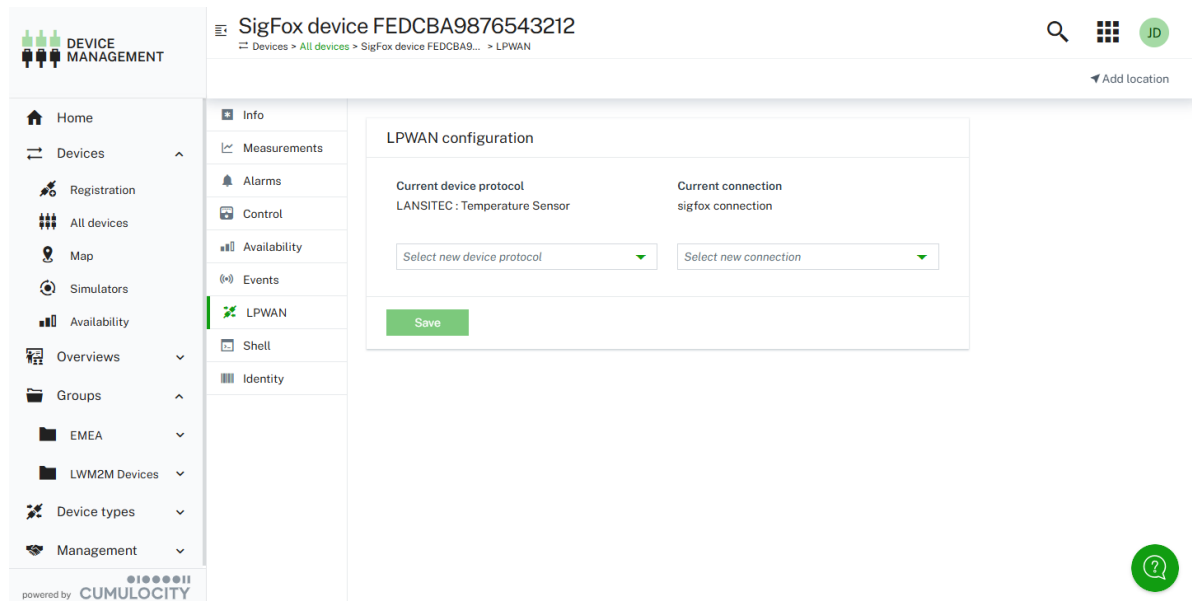
CREATING DEVICE PROTOCOLS

To process data from Sigfox devices, Cumulocity must understand the payload format of the devices. Mapping payload data to

Cumulocity data can be done by creating a Sigfox device protocol.

During the [device registration](#), you can associate this device protocol. The received uplink callbacks for this device with a hexadecimal payload will then be mapped to the ones you have configured in your device protocol.

The device protocol assigned during Sigfox device registration can be changed from the **LPWAN** tab in the device details page.



INFO

Device protocol mapping only supports decoding for fixed byte positions based on the message type. The length for the device payload parts, which is set in the **Number of bits** field, can be maximum 32 bits (4 bytes).

To create device protocols, select **Device protocols** in the **Device types** menu in the navigator of the Device Management application. You can either import an existing device protocol or create a new one.


Importing a device protocol

In the **Device protocols** page, click **Import**.

Select the desired predefined device type or upload it from a file. When ready, click **Import** again.

Creating a new device protocol

In the **Device protocols** page, click **New device protocol** and select **Sigfox** from the options list.



ADD DEVICE PROTOCOL

Select one of the available options

Find your protocol in the [user documentation](#) to get more information.

✖ CAN Bus	>
✖ CANopen	>
✖ LoRa	>
✖ LWM2M	>
✖ Modbus	>
✖ OPC UA	>

Cancel

Provide a name for the device protocol and an optional description, and click **Create**.

Under **Message types**, specify the message types. Sigfox devices can send messages of different types with different encodings per type. Depending on the device, the type can be determined by looking either at the FPort parameter of a message (Source: FPort) or at the subset of the message payload itself (Source: Payload).

In the **Source** field, select the way the message type is encoded:


- **Payload:** if the message type can be determined by looking at the subset of the message payload itself




In the following sample payload structure, the first byte indicates the message type source (as highlighted).


Message type source

0	1	2	3	4	5	6	7	8	9	10	11
Code	Status	PAYLOAD									
0x01	Cf Status	Channel 1 Type	Measure sensor 1 (LSB First)			Channel 2 Type	Measure sensor 2 (LSB first)			X	X
0x03	Cf Status	Device Type	Transmit period		Channel On/Off	Channel 1 Type	Channel 2 Type	NA (0x00)	Switch Value	X	X

In the user interface, you can enter this type of message type source information as follows: In the **Start bit** field, indicate where the message type information starts in the payload. In the **Number of bits** field, indicate how long this information is. For example, start bit = "0" and number of bits = "8".

 sigfox protocol
Device types > Device protocols

 sigfox protocol
e.g. My protocol description

ID	557211
Date created	7 Mar 2025, 17:38:00
Last update	7 Mar 2025, 17:42:06
Fieldbus version	4

Message types

Source
Payload

Sigfox devices can send messages of different types with different encodings per type. Type can be determined by looking at the subset of the message payload (**Source: Payload**). Indicate where the type information starts in the payload (**Start bit**) and how long this information is (**Number of bits**).

Start bit ②
0

Number of bits ②
8

Values

ELEMENTS

My Element	MESSAGE TYPE ID 1	START BIT 0	NUMBER OF BITS 16	
------------	-------------------	-------------	-------------------	--

Save

Configuring values

In the **Values** section, click **Add value** to create the value configuration.

In the following window, configure the relevant values as shown in this example.

New value window part 1

New value

MESSAGE TYPE ②

Message ID ②
1

GENERAL

Name
Channel Type 1

Display category
Sensor Measure Data

VALUE SELECTION ②

Start bit
16

Number of bits
8

VALUE NORMALISATION ②

Cancel


Save


New value window part 2


New value

Little-endian?

FUNCTIONALITIES?

☐  Send measurement ?

☐  Raise alarm ?

☒  Send event ?

Event type

c8y_MeasureSensor

Event text


Measurement sensor 1 event

Event fragment

c8y_MeasureSensor

Event property ?

sensor1 Type

☐  Update managed object ?

Cancel

Save

The value configuration maps the value in the payload of a message type to the Cumulocity data.

Under **Message type**, configure the **Message ID** according to your device message specification. The message ID is the numeric value identifying the message type. It will be matched with the message ID found in the source specified on the device protocol main page (that is, Payload or FPort). The message ID must be entered in decimal numbers (not hex).

In this sample payload structure the message ID is "1".

Message ID

0	1	2	3	4	5	6	7	8	9	10	11
Code	Status	PAYLOAD									
0x01	Cf Status	Channel 1 Type	Measure sensor 1 (LSB First)			Channel 2 Type	Measure sensor 2 (LSB first)			X	X
0x03	Cf Status	Device Type	Transmit period	Channel On/Off	Channel 1 Type	Channel 2 Type	NA (0x00)	Switch Value	X	X	

New value

MESSAGE TYPE ?

Message ID ?

1

GENERAL

Name **Display category**

Channel Type 1 Sensor Measure Data

VALUE SELECTION ?

Start bit **Number of bits**

16 8

VALUE NORMALISATION ?

Cancel Save

Under **General**, specify a name for the value and the category under which it will be displayed in the values list.

Under **Value selection**, specify from where the value should be extracted. In the **Start bit** field, indicate where the value information starts and in the **Number of bits** field, indicate the length of the information. The maximum value for the number of bits is 32 bits (4 bytes).

In this example, the "Channel 1 Type" information starts in byte 2 (that means, start bit = "16") and is 1 byte long (that means, number of bits = "8").

Value selection:
start bit and number of bits

0	1	2	3	4	5	6	7	8	9	10	11
Code	Status		PAYLOAD								
0x01	Cf Status	Channel 1 Type	Measure sensor 1 (LSB First)			Channel 2 Type	Measure sensor 2 (LSB first)			X	X
0x03	Cf Status	Device Type	Transmit period		Channel On/Off	Channel 1 Type	Channel 2 Type	NA (0x00)	Switch Value	X	X

New value

MESSAGE TYPE ?

Message ID ?

1

GENERAL

Name Display category

Channel Type 1 Sensor Measure Data

VALUE SELECTION ?

Start bit Number of bits

16 8

VALUE NORMALISATION ?

Multiplier Offset Unit

Cancel Save

The hexadecimal value is converted to a decimal number and afterwards a “value normalization” is applied.

Under **Value normalization**, specify how the raw value should be transformed before being stored in the platform and enter the appropriate values for:

- **Multiplier:** This value is multiplied with the value extracted from the **Value selection**. It can be decimal, negative or positive. By default it is set to 1.
- **Offset:** This value defines the offset that is added or subtracted. It can be decimal, negative or positive. By default it is set to 0.
- **Unit (optional):** A unit can be defined which is saved together with the value (for example temperature unit “C” for degree Celsius).

For detailed information on how to decode the payload, refer to the documentation of the device.

Under Options, select on of the following options, if required:

- **Signed** - if the value is a signed number.
- **Packed decimal** - if the value is BCD encoded.

Under **Functionalities**, specify how this device protocol should behave:

- **Send measurement:** creates a measurement with the decoded value.
- **Raise alarm:** creates an alarm if the value is not equal to zero.
- **Send event:** creates an event with the decoded value.
- **Update managed object:** updates a fragment in a managed object with the decoded value.

You can also have a nested structure with several values within a measurement, event or managed object fragment. In case of a measurement, all the properties of the same type will be merged to create a nested structure. In case of an event or a managed object all the properties with the same fragment are merged to create a nested structure. (Also refer to the [example](#) of a nested structure for a “Position” device protocol below.)

Click **OK** to add the values to your device protocol.

New value window part 2

New value

Multiplier	Offset	Unit
1	0	%


OPTIONS


☐ Signed?


☐ Packed decimal?


☐ Little-endian?

FUNCTIONALITIES?

☐  Send measurement ?

☐  Raise alarm ?

☐  Send event ?

☐  Update managed object ?

Cancel

Save

Example with nested structure

The following images show an example of a nested structure for a device protocol, reporting the current position of a GPS device. The device protocol is named "Position" and contains values for longitude and latitude.

The message ID should be the same for all the values. Enter the rest of the parameters according to the instructions above. Enter "c8y_Position" in the **Managed object fragment** field and create a new value for each: longitude and latitude.





New value window, Longitude

New value

OPTIONS

- ☐ Signed ?
- ☐ Packed decimal ?
- ☐ Little-endian ?

FUNCTIONALITIES ?

- ☐  Send measurement ?
- ☐  Raise alarm ?
- ☐  Send event ?
- ☒  Update managed object ?

Managed object fragment

Managed object property ?

c8y_Position

lng

Cancel

Save

New value window, Latitude

New value


OPTIONS


☐ Signed ?


☐ Packed decimal ?


☐ Little-endian ?

FUNCTIONALITIES ?

☐  Send measurement ?

☐  Raise alarm ?

☐  Send event ?

☒  Update managed object ?




Managed object fragment




c8y_Position




Managed object property ?

lat



This will be the result:

  **GPS Protocol** 
 Device types > Device protocols





 **GPS Protocol** 
 e.g. My protocol description 


Message types

Start bit  Number of bits 

Values

UNCATEGORIZED

 Latitude	MESSAGE TYPE ID 1 START BIT 8 NUMBER OF BITS 32	
 Longitude	MESSAGE TYPE ID 1 START BIT 40 NUMBER OF BITS 32	

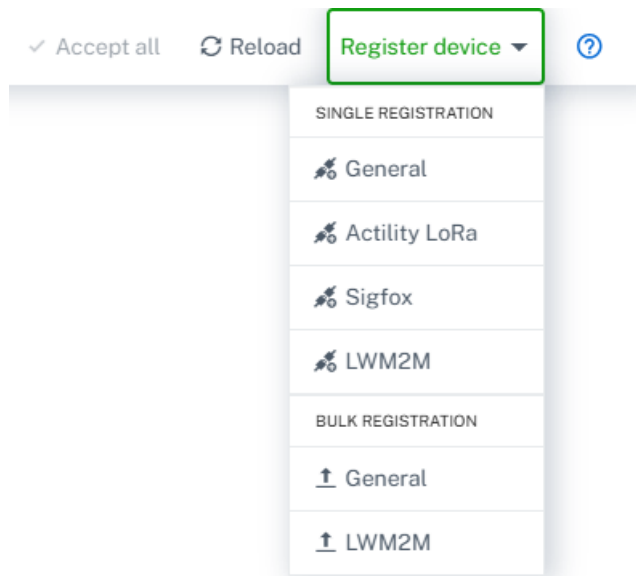


Using custom decoding/encoding

The Sigfox agent also supports the decoding/encoding functionality by plugging in the custom microservice. Refer to [LPWAN custom protocols](#) for further details.

REGISTERING SIGFOX DEVICES

To register a Sigfox device in Cumulocity navigate to **Devices > Registration** in the Device Management application, click **Register device** at the top right and select **Single device registration > Sigfox** from the dropdown.



INFO

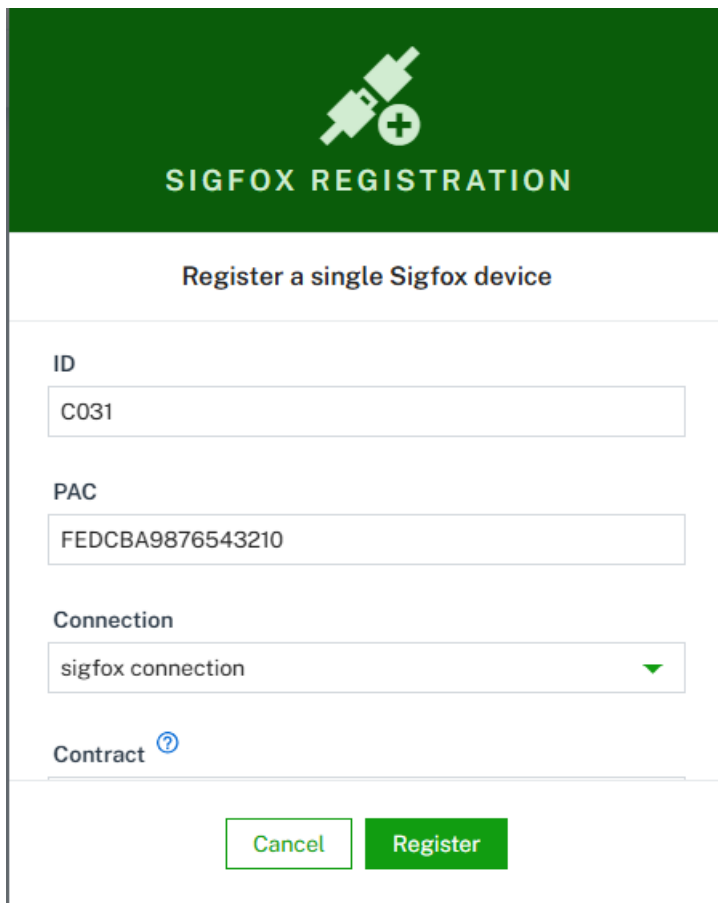
- If Sigfox is not one of the available options, your tenant is not subscribed to the relevant application, see the requirements in the [introduction](#).
- The device type created in the Sigfox Cloud platform has the following naming convention `c8y_{tenantId}_{device-protocol-name}_{contractId}` , for example:
`c8y_myTenant_mySigfoxDeviceProtocol_aabbcc5b78c901d64eecf4faaa`
- If the constructed name exceeds 100 characters it will be truncated until it is less than 100 characters.

In the next window, fill in the required information:

- **ID:** Unique device ID. The value must be a hexadecimal number.
- **PAC:** Porting authorization code for your device. The value must be a hexadecimal number.
- **Connection:** Lists all configured Sigfox connections in the tenant. The following contract option is populated based on the selected Sigfox connection.
- **Contract:** Select your desired contract (all contracts are listed including active and expired).
- **Device protocol:** Select your desired device protocol from the drop-down list.
- **Product certificate key:** This key can be located in <https://partners.sigfox.com/>. Navigate to your device and copy the certificate key. If the checkbox is not selected and no product certificate key is specified, the device will be considered a prototype.

INFO

The term “Device type” is used both by Sigfox and Cumulocity, but with different meaning. In Sigfox, a device type specifies how to route data from devices. In Cumulocity, a device type describes the data that is sent by devices of a particular type.



The image shows a web form titled "SIGFOX REGISTRATION" with a green header. Below the header, the title "Register a single Sigfox device" is centered. The form contains four input fields: "ID" with the value "C031", "PAC" with the value "FEDCBA9876543210", "Connection" with a dropdown menu showing "sigfox connection", and "Contract" with a question mark icon. At the bottom, there are two buttons: "Cancel" and "Register".

Click **Register** to submit the device registration request and create the device.

You can verify that the device is really connected by checking that events are actually coming in. You can do so by clicking on a device and opening its **Events** tab. All events related to this device are listed here.

For more information on viewing and managing your connected devices, also refer to the [Device Management application](#).

In order to migrate the device from one LNS connection to another, the device must be re-registered. Navigate to the **LPWAN** tab of the Device. Click on the **Provider connection** dropdown. A prompt will appear stating that in order to migrate the device from one LNS connection to another, you must re-register the device. Click on the **Re-Register** button.

The user is directed to the device registration page where he can perform the re-registration following the steps above and selecting the desired LNS connection.

Updating devices registered with the general device registration

If devices have previously been registered via the general device registration the following URLs must be manually changed in the Sigfox Cloud:

- `https://sigfox-agent.cumulocity.com/sigfoxDataCallback` to `https://<tenantId>.cumulocity.com/service/sigfox-agent/sigfoxDataCallback` .
- `https://sigfox-agent.cumulocity.com/sigfoxServiceAcknowledgeCallback` to `https://<tenantId>.cumulocity.com/service/sigfox-agent/sigfoxServiceAcknowledgeCallback` .
- `https://sigfox-agent.cumulocity.com/sigfoxServiceStatusCallback` to `https://<tenantId>.cumulocity.com/service/sigfox-agent/sigfoxServiceStatusCallback` .
- `https://sigfox-agent.cumulocity.com/sigfoxErrorCallback` to `https://<tenantId>.cumulocity.com/service/sigfox-agent/sigfoxErrorCallback` .

INFO

General device registration for Sigfox devices is no longer supported.

SENDING OPERATIONS

If the device supports sending hexadecimal commands, you can send them using shell operations.

In order to send an operation, navigate to the device you want to send an operation to in the Device Management application under **All devices** and switch to the **Shell** tab.

INFO

Operations do not go to a status of EXECUTING immediately. They go to EXECUTING when the device is expecting the downlink message. Afterwards, the pending operation which is created first goes to a status of EXECUTING.

UPLINK MESSAGE PROCESSING

On receiving an uplink message, the Cumulocity platform creates the following measurements and events, and updates the corresponding device managed object.

- **Unprocessed data** - An event of type `com_sigfox_UnprocessedDataEvent` is created with the unprocessed data.
- **Position** - The `c8y_Position` fragment of the device managed object is updated to capture the latitude, longitude, altitude and accuracy information of the device.
- **Signal strength** - A measurement is created with RSSI and SNR values of the device signal strength.

TROUBLESHOOTING

Device registration

No active contracts with free slots available

Active contracts with free slots are filtered based on the activation end time and tokens in use. Contracts in which the activation end time is higher than the current time or the activation end time is unlimited, and contracts in which the max tokens are higher than the tokens in use or the max tokens are unlimited will be considered.

In order to resolve this error, please contact support.sigfox.com to create a contract for your Sigfox account.

No Sigfox provider settings are found

This warning message shows up when there are no connections set up for the sigfox connectivity.

To resolve this, refer to [Configure Sigfox credentials](#).

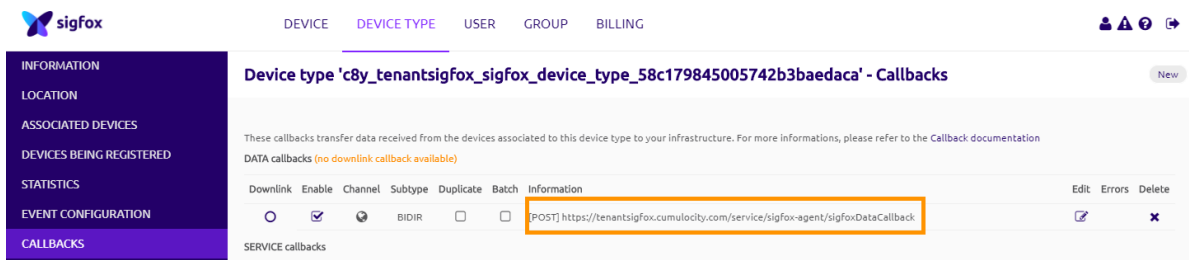
No Sigfox device type configured

This warning message shows up when no Sigfox device protocol exists to be used for device registration.

To resolve this, configure at least one device protocol in the [Device database](#).

Connectivity

Sigfox callbacks in `backend.sigfox.com` are not created correctly



Device type 'c8y_tenantsigfox_sigfox_device_type_58c179845005742b3baedaca' - Callbacks

These callbacks transfer data received from the devices associated to this device type to your infrastructure. For more informations, please refer to the [Callback documentation](#)

DATA callbacks (no downlink callback available)

Downlink	Enable	Channel	Subtype	Duplicate	Batch	Information	Edit	Errors	Delete
<input type="radio"/>	<input checked="" type="checkbox"/>		BIDIR	<input type="checkbox"/>	<input type="checkbox"/>	[POST] https://tenantsigfox.cumulocity.com/service/sigfox-agent/sigfoxDataCallback			

SERVICE callbacks

The information for the callback setup is retrieved by a microservice.

To verify whether your setup is correct, execute the following REST API request:

```

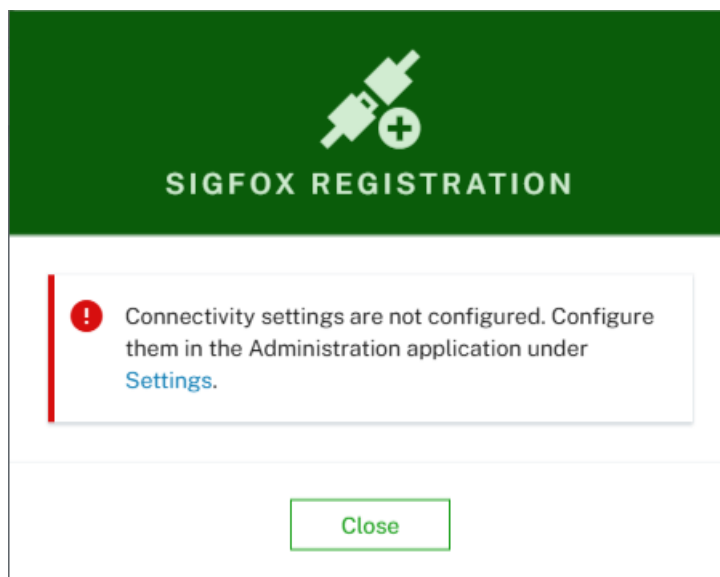
http
GET {{url}}/tenant/currentTenant

```

INFO

The request above is simply an example API request that could be used. For more info on REST API requests, refer to the [Tenants](#) in the Cumulocity OpenAPI Specification.

This warning message shows up when there are no connections set up for the Sigfox account. To resolve this click **Settings** to navigate to the Administration application where the connections are configured.



SIGFOX REGISTRATION

Connectivity settings are not configured. Configure them in the Administration application under [Settings](#).

Close

No device protocols configured

This warning message shows up when no Sigfox device protocol exists to be used for device registration. To resolve this, click **Device protocols** to navigate to the **Device protocols** page where the protocols are configured.



Issues with alarm provisioning

Transfer operation failed

The screenshot shows the 'SigFox device FEDCBA9876543210' page in the 'DEVICE MANAGEMENT' section. The left sidebar contains navigation links: Home, Devices, Registration, All devices, Map, Simulators, Availability, Overviews, Groups, EMEA, LWM2M Devices, Device types, and Management. The main content area has tabs for Info, Measurements, Alarms, Control, Availability, Events, LPWAN, Shell, and Identity. The 'Alarms list' tab is active, showing an alarm triggered on 25 Mar 2025 at 17:33:27. The alarm message states: 'Sigfox device [FEDCBA9876543210] was provisioned but transfer operation failed in Sigfox Cloud. Please transfer the device to new type c8y_jd_sigfox-device-type manually in Sigfox Cloud.' The right panel provides details about the alarm, including its status (ACTIVE, triggered in 7 hours), severity (Critical), source (SigFox device FEDCBA9876543210), and type (c8y_SigfoxUpdateDevice). It also includes buttons for 'Reload audit logs', 'Acknowledge', 'Create smart rule', and 'Clear'. The 'AUDIT LOGS' section at the bottom indicates 'No audit logs found.'

If the “transfer operation failed” alarm is triggered, the device is already provisioned in the Sigfox platform and changing the device type in the Sigfox platform failed. In order to fix this issue, you must manually change the device type in the Sigfox platform to the intended one.

Provisioned status is set to false

The screenshot displays the 'SigFox device FEDCBA9876543210' page in a management dashboard. The left sidebar contains navigation options: Home, Devices, Overviews, Groups, EMEA, LWM2M Devices, Device types, and Management. The main content area is divided into several sections:

- Info:** Shows device details like Owner (service_s...), Required interval (--- minutes), and a 'Provisioned' status checkbox which is currently unchecked.
- Measurements:** A timeline graph showing data points for 'cBy_LocationUpdate' and 'cBy_UnavailabilityAlarm'.
- ACTIVE, CRITICAL ALARMS:** A section with an 'AUTO REFRESH' button and a red alarm icon. The alarm message states: 'Sigfox device provisioning has failed with the error message: 0002: Invalid PAC code: 290348238 (please contact your device)'. The timestamp is '25 Mar 2025 17:33:27'.
- GROUP ASSIGNMENT:** A section indicating 'Device not assigned. Assign the device to a group below.' with a search bar.
- LOCATION:** A map showing the device's location.

At the bottom left, it says 'powered by CUMULOCITY'.

In case of this alarm, you can see that the **Provisioned** status is set to "false" which means that no data is coming from the Sigfox platform. In the alarm message there is more information regarding the error. In this case the PAC code given during registration was invalid.

INFO

If the provisioning process has been completed, but has failed, information is returned as an alarm with the reason of the failure provided.

The **Provisioned** status is set to true when the device provisioning process is completed and success information is received from the Sigfox platform. Additionally, it is set to true when uplink messages are retrieved from the device.

INFO

The status is updated asynchronously which means that sometimes you might have to wait a bit until it is set to true.

Callback creation failed

The screenshot shows the 'Alarms list' section of a web interface. On the left is a sidebar with navigation links: Info, Measurements, Alarms (highlighted), Control, Availability, Events, Shell, and Identity. The main area displays an alarm titled 'Callback creation failed in Sigfox platform for device type 5cd3d97ee833d9746698b27d'. The alarm details include a list of failed callback properties, such as DATA_BIDIR, SERVICE_ACKNOWLEDGE, and SERVICE_STATUS. A large block of JSON data is shown, representing the callback properties that failed to be created in the Sigfox platform. A green question mark icon is visible in the bottom right corner of the alarm details panel.

This alarm is created when one or more callback creation requests have failed in the Sigfox platform. You can view the alarm either in the **Alarms** page or in the **Home** page.

In order to fix this issue, navigate to the Sigfox platform web interface and check the device type with the id mentioned in the alarm.

This screenshot is similar to the one above, showing the same alarm details. However, the device ID '5cd3d97ee833d9746698b27d' in the alarm title is highlighted with an orange box. Additionally, a green question mark icon is present in the bottom right corner of the alarm details panel.

In this case navigate to the following address: <https://backend.sigfox.com/devicetype/5cd3d97ee833d9746698b27d/callbacks>

If the mentioned callbacks cannot be located in the Sigfox platform, you must create them manually. All of the required information needed for the creation of the callbacks is already given in the alarm description. In the case of the above alarm, the following callback is listed first:

- `[[callback=[type=DATA_BIDIR, url=<tenant_url>/service/sigfox-agent/sigfoxDataCallback, httpMethod=POST, bodyTemplate={\"device\": \"{device}\", \"time\": \"{time}\", \"snr\": \"{snr}\", \"station\": \"{station}\", \"data\": \"{data}\", \"rssi\": \"{rssi}\", \"seqNumber\": \"{seqNumber}\"}, \"ack\": \"{ack}\"}, contentType=application/json, headers={Authorization=Basic ...}]]`

In order to manually create the callback, the following properties must be filled:

- type
- url
- httpMethod
- bodyTemplate
- contentType
- headers

The Authorization header displayed in the alarm does not show the user credentials.

Non-mentioned properties from the alarm are:

- sendSni
- sendDuplicate

These properties will be set to false.

LPWAN CUSTOM PROTOCOLS

INTRODUCTION

Cumulocity can interface with LPWAN devices through LPWAN network providers via Cumulocity LPWAN agents, such as [Activity LoRa](#).

Our LoRa integration allows you to define device protocols in a self-service manner to create a binary mapping of the LoRa sensor data to the Cumulocity data model. However, this approach does not work for LoRa devices sending dynamic payloads. To integrate LoRa devices with dynamic payloads, a custom codec for payload decoding and command encoding can be created in form of a microservice. This microservice will be referred to as a custom codec microservice. A custom codec microservice is a typical Cumulocity microservice which conforms to a specific contract for decoding and encoding LoRa payloads and commands.

When an LPWAN agent receives an uplink message, it forwards the device data to a REST endpoint (such as `/decode`) exposed by the custom codec microservice for decoding. Similarly, when the user executes a device command through the device shell, the LPWAN agent forwards the command text to a REST endpoint (such as `/encode`) exposed by the custom codec microservice for encoding.

IMPLEMENTING A CUSTOM CODEC MICROSERVICE

A custom codec microservice is a typical Cumulocity microservice, which can be implemented and enabled as follows.

1. Create a microservice which exposes the `/encode` and `/decode` REST endpoints conforming to the [OpenAPI Specification](#), implementing the encoding and decoding functionality.
2. The microservice must create device protocols for each LPWAN device type it supports. If you use the `lpwan-custom-codec` library the device protocols will be created automatically for you. Otherwise, you must use the Inventory API to create a new managed object describing the device protocol with the following JSON structure:

You must create a device protocol (with `type` and `fieldbusType` properties, and the `c8y_LpwanCodecDetails` fragment) as well as an external ID for every device manufacturer and device model combination that this codec microservice supports:

- `type` is always "c8y_LpwanDeviceType".
- `fieldbusType` is always "lpwan".
- The `c8y_LpwanCodecDetails` fragment contains:
 - `codecServiceContextPath` - Custom codec microservice context path.
 - `supportedDevice` - Supported device information.
 - `deviceManufacturer` - Device manufacturer.
 - `deviceModel` - Device model.
 - `supportedDeviceCommands` - A list of commands which this device supports:
 - `name` - Command name, matching the name of the predefined command template you create (see below).
 - `category` - Command category, matching the category of the predefined command template you create (see below).

Example for the JSON structure for creating a device protocol using the Inventory API:

```
{
  "name": "<<Name of the LPWAN device protocol>>",
  "description": "<<Description of the LPWAN device protocol>>",
  "type": "c8y_LpwanDeviceType",
  "fieldbusType": "lpwan",
  "c8y_IsDeviceType": {},
  "c8y_LpwanCodecDetails": {
    "codecServiceContextPath": "<<Custom Codec microservice context path>>",
    "supportedDevice": {
      "deviceManufacturer": "<<Supported device manufacturer>>",
      "deviceModel": "<<Supported device model>>",
      "supportedDeviceCommands": [
        {
          "name": "<<Command name, matching the name of the Predefined Command template you create>>",
          "category": "<<Command category, matching the category of the Predefined Command template you create>>"
        }
      ]
    }
  }
}
```

Example for the JSON structure for creating an external ID for the device protocol using the Identity API:

```
{
  "externalIds": [
    {
      "managedObject": "<<ID of the Device protocol managed object>>",
      "externalId": "<<Device Protocol Name>>",
      "type": "c8y_SmartRestDeviceIdentifier"
    }
  ]
}
```

- Based on the device protocol assigned to a device, the LPWAN agent automatically routes the request to the corresponding microservice. For device downlink commands, the LPWAN agent forwards the device shell command request to the `/encode` endpoint only when a predefined command listed as "supportedDeviceCommands" in the `c8y_LpwanCodecDetails` fragment of the device protocol is executed.

You must create a predefined command template for every supported device command (`supportedDeviceCommands`) specified in the device type.

The following example shows the JSON structure for creating a predefined command template using the Inventory API:

```
{
  "type": "c8y_DeviceShellTemplate",
  "name": "<<Command name, matching the name of the supported command mentioned in the device protocol>>",
  "deviceType": [
    "<<Device Protocol Name>>"
  ],
  "category": "<<Command Category>>",
  "command": "<<Command string which gets copied to the device shell command prompt when the user chooses this Predefined command>>",
  "deliveryTypes": [
    "Default"
  ]
}
```

USING THE LPWAN CUSTOM CODEC LIBRARY

For convenience, Cumulocity provides the Java library `com.nsn.cumulocity.clients-java:lpwan-custom-codec` to develop the custom codec microservice in Java as a SpringBoot application. When subscribed, such a custom codec microservice automatically creates the required device protocols and predefined command templates as described in [Implementing a custom codec microservice](#).

To create a custom codec microservice using this library, do the following:

1. Add the following dependency to the pom.xml file:

```

<?xml
<dependency>
  <groupId>com.nsn.cumulocity.clients-java</groupId>
  <artifactId>lpwan-custom-codec</artifactId>
  <version>${c8y.version}</version>
</dependency>

```

2. Create a Spring Boot application and annotate its main class with:

```
@CodecMicroserviceApplication `com.cumulocity.microservice.lpwan.codec.annotation.CodecMicroserviceApplication`
```

3. Implement the following Java interfaces and annotate them with:

```
@Component `org.springframework.stereotype.Component`
```

a. Implement the `supportedDevices` method of `com.cumulocity.microservice.lpwan.codec.Codec` to specify the device manufacturer, device name and the commands this custom codec service supports:

```

java
package com.cumulocity.microservice.lpwan.codec;

public interface Codec {
    Set<DeviceInfo> supportedDevices();
}

```

b. Implement the `decode` method of `com.cumulocity.microservice.customdecoders.api.service.DecoderService` to provide the decode functionality. The decode method receives the following input:

- `inputData` - device payload to be decoded.
- `deviceId` - device managed object ID.
- `args` - a map which contains keys:
 - `deviceManufacturer` - device manufacturer.
 - `deviceModel` - device model.
 - `sourceDeviceEui` - device eui.
 - `fport` - fport (optional).

The library also provides the wrapper class

`com.cumulocity.microservice.lpwan.codec.decoder.model.LpwanDecoderInputData` to extract the decoder inputs.

```

java
package com.cumulocity.microservice.customdecoders.api.service;

public interface DecoderService {
    DecoderResult decode(String inputData, GId deviceId, Map<String, String> args) throws DecoderServiceException;
}

```

c. Implement the `encode` method of `com.cumulocity.microservice.customencoders.api.service.EncoderService` to provide the encode functionality:


```
```java
package com.cumulocity.microservice.customencoders.api.service;

public interface EncoderService {
 EncoderResult encode(EncoderInputData encoderInputData) throws EncoderServiceException;
}
```
```

4. Add the following permissions in the microservice manifest file `cumulocity.json` . Note that the respective field in the manifest is called `requiredRoles` , but what you actually add is a list of required permissions strings.

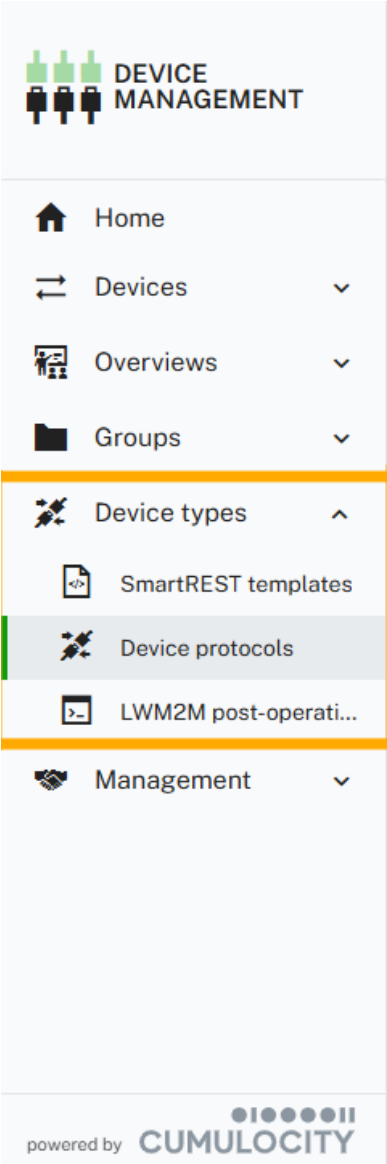
```
"requiredRoles": [
  "ROLE_INVENTORY_ADMIN",
  "ROLE_INVENTORY_READ"
]
```

DEPLOYING THE SAMPLE CODEC MICROSERVICE







Steps to build the example codec `lora-codec-lansitec` microservice.

1. Clone the <https://github.com/Cumulocity-IoT/cumulocity-examples.git> repository.
2. Build the microservice using `mvn clean install` . This creates a ZIP file of the lansitec codec microservice.
3. Deploy the microservice by uploading the ZIP file using the Cumulocity Administration UI.
4. Open the Device Management application. Under **Device protocols**, you should now see the device protocols with type "lpwan" created by the lansitec codec microservice.


The image below shows the device protocols option in the Device Management application.



The image below shows an example of the device protocols created by the custom codec microservice on subscription.

| | | | | |
|---|--|-------|---|---|
|  LANSITEC : Asset Tracker | Device protocol that supports device model Asset Tracker manufactured by LANSITEC | LPWAN |  |  |
|  LANSITEC : Temperature Sensor | Device protocol that supports device model Temperature Sensor manufactured by LANSITEC | LPWAN |  |  |

The created device protocols will be listed in the dropdown during the device registration of LPWAN with any of the LPWAN agents. The Activity LoRa device registration is shown below.



ACTIVITY LORA REGISTRATION

Register a single Activity device

Device profile

e.g. IWM-LR3 (required) ▼

Device protocol

Start typing to search, for example, LANSITEC : Asset Ti ▼

LANSITEC : Asset Tracker

LANSITEC : Temperature Sensor

Test


e.g. 70B3D53260000003 (required)


Cancel


Register


You can also assign the device protocol from the **LPWAN** device tab. To do so, navigate to a particular device. Then, switch to the **LPWAN** tab and click **New device protocol** to view the device protocols created above.


  **Activity_ABCDEF1234567833**
[Devices](#) > [All devices](#) > [Activity_ABCDEF123456...](#) > **LPWAN**


 Info


 Measurements


 Alarms


 Control

 Availability

 Events

 **LPWAN**

 Shell

 Identity

LPWAN configuration

Current device protocol

Test

Select new device protocol ▼

LANSITEC : Asset Tracker

LANSITEC : Temperature Sensor

Supported device commands are available in the **Predefined commands** option from the **Device shell** tab.

