Critical manufacturing
an ASM PT company

Connect IoT - REST Client and Server Tutorial

12.0.1

November 2026

DOCUMENT ACCESS

Public

DISCLAIMER

The contents of this document are under copyright of Critical Manufacturing S.A. it is released on condition that it shall not be copied in whole, in part or otherwise reproduced (whether by photographic, or any other method) and the contents therefore shall not be divulged to any person other than that of the addressee (save to other authorized offices of his organization having need to know such contents, for the purpose for which disclosure is made) without prior written consent of submitting company.

12.0

Copy Page

Connect IoT - REST Client and Server Tutorial

Overview

HTTP reaches a controller through two drivers, one for each direction:

Driver	Direction	What the driver definition holds
REST Client	The controller calls an HTTP API	Commands only: each command is one request (path, method, and where each parameter goes). The reply is the response body
REST Server	Another system calls the controller	Events and properties: each event is one endpoint, and its properties read parts of the request. The package ships the reply command

A REST Client cannot be called, and a REST Server cannot call: a controller that is asked and must ask someone else uses both, as two drivers of one controller. Neither driver reaches MES; for that, use the tasks for MES shown in Connect IoT - Talking to MES Tutorial.

In this tutorial, you build a track-in gateway. A line gateway posts `POST /gateway/lot-start/<lot>` and waits. The controller holds the request, asks a scheduling system `GET /v2/lot-tracking/<lot>?detail=summary`, and answers the gateway with the scheduling system's reply, or with `502` when the scheduling system cannot be reached. The tutorial is for integrators who completed the basic configuration tutorial and build Data Flow workflows.

Info

For more information, see Connect IoT - Rest Client and Connect IoT - Rest Server.

Before You Start

- Complete Connect IoT - Basic Configuration Tutorial: you need an **Automation Manager** that you can start in console mode, the **Resource** entity type enabled for Connect IoT, an integration user, and a **Resource**, such as `Gateway-01`.
- Make the packages `@criticalmanufacturing/connect-iot-driver-restclient` and `@criticalmanufacturing/connect-iot-driver-restserver` available in the Connect IoT repository.
- Choose a free port above 1024 on the machine that runs the **Automation Manager**, for example `8085`. A port in use fails the connection, and a port below 1024, the default `80` included, usually needs rights that the **Automation Manager** does not have.
- Prepare a stand-in for the scheduling system that answers `GET /v2/lot-tracking/B-1001` with JSON, such as `{"lotId": "B-1001", "state": "Staged"}`. Any HTTP service you control works. For automated tests, the Test Orchestrator Rest Server simulator ↗ hosts such an endpoint.
- Prepare a stand-in for the gateway: any HTTP client, such as `curl`. For automated tests, use the Test Orchestrator Rest Client simulator ↗.

Sequence of Steps

Step 1: Create the Two Automation Protocols

Go to **Business Data** > **Automation Protocol** and create one protocol for each package, as described in How to: Create an Automation Protocol:

1. Create `REST Client` with the package `@criticalmanufacturing/connect-iot-driver-restclient` and keep the default parameter values.
2. Create `REST Server` with the package `@criticalmanufacturing/connect-iot-driver-restserver` and keep the default parameter values.

Both protocols are in the `Created` state. You set the values below per connection in Step 4:

Protocol	Parameter	Value in this tutorial	Why it matters
REST Client	`baseUrl`	The host of the scheduling system, such as `127.0.0.1`	A host with an optional base path (`api.example.com/v1`), without the scheme and without a query string

Protocol	Parameter	Value in this tutorial	Why it matters
REST Client	`protocol`	`HTTP`	`HTTPS` for a TLS endpoint, with `clientCertificate`, `clientPrivateKey`, and `certificateAuthority` when the server requires them
REST Client	`port`	The port of the stand-in, such as `8090`	When it is greater than `0`, the driver writes it into every URL, even the usual port of the protocol: set `443` for `HTTPS`, or `0` to add no port
REST Client	`authenticationMode`	`None`	`BasicAuthentication`, `BearerToken`, or `APIKey`, with the matching credential parameters
REST Server	`address`	`0.0.0.0`	`0.0.0.0` listens on every interface. An empty address listens on `127.0.0.1` only
REST Server	`port`	`8085`	The free port you chose
REST Server	`protocol`	`HTTP`	`HTTPS` needs `certificate` and `key`

Step 2: Describe the API You Call

The REST Client package ships no driver definition, so every request the controller makes is a command you declare. The **Endpoint Path** of the command (its Command Device ID) is the endpoint path, with `:<name>` for each segment that varies.

1. Go to **Business Data** > **Automation Driver Definition**, create a definition named `Scheduler API` on the `REST Client` protocol with the entity type `Resource`, and skip the properties and events.
2. Add the command and its parameters:

Entry	Name	Command Device ID or Type	Extended data
Command	`GetLotStatus`	`/v2/lot-tracking/:lotId`	`requestMethod GET`, `successfulStatusCodes 200`, `timeout 30000`
Parameter	`lotId`	`String`, mandatory	`operationParameters PathParameter`
Parameter	`detail`	`String`, default value `summary`	`operationParameters QueryParameter`

3. Select **Create**.

The driver builds the request from names alone: a `PathParameter` replaces the `:<name>` segment that carries its exact name, and a `QueryParameter` is appended under its name. A call with `lotId` set to `B-1001` requests `/v2/lot-tracking/B-1001?detail=summary` on the base URL. A `GET` sends no body; `RequestBody` and `RequestBodyParameter` parameters apply to `POST`, `PUT`, and `PATCH`.

Note

List in `successfulStatusCodes` every status that the API answers on success, for example `200,201`. A status outside the list fails the command even when the API did the work. The reply holds the response body only: the status code and the response headers are not returned.

Step 3: Describe the Endpoint You Expose

On the REST Server, an event is an endpoint. Its **Endpoint Path** (Event Device ID) is the path in route syntax, starting with `/`, and a request to that path raises the event. Each property reads one part of the request: its **Property Name** (Property Device ID) is the name in the request, and its name is the name the workflow uses.

1. Create a driver definition named `Gateway Endpoint` on the `REST Server` protocol with the entity type `Resource`.
2. Add the properties:

Name	Property Device ID	Type	requestComponentType	fullContent
LotId	lotId	String	Path	false
Operator	badgeId	String	Body	false
RequestId	requestId	String	RequestId	false

3. Add the event `TrackInRequested` with the **Event Device ID** `/gateway/lot-start/:lotId` and the extended data `requestMethod` `POST`, `automaticReply` `false`, and `timeout` `60000`.
4. Add `LotId`, `Operator`, and `RequestId` to the event, skip the commands, and select **Create**.

`LotId` is filled because its **Property Device ID** matches the route segment `:lotId`. `RequestId` carries the identifier that the driver gives each request, and it is the only way a reply finds its request. With `automaticReply` `false`, the driver holds the request until the workflow replies or the event `timeout` passes. With the default `true`, the driver answers `200` with an empty body as soon as the event is raised, which suits a push that only expects an acknowledgment.

You declare no command: the package ships the command `Request Reply Command` (device ID `RequestReplyCommand`), with the parameters `RequestId`, `StatusCode` (default `200`), and `Body`.

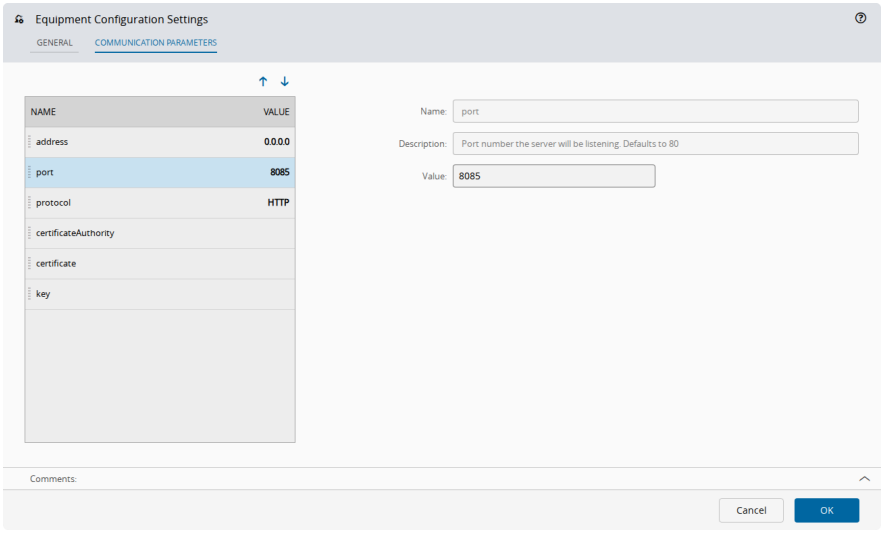
Execute Equipment Command lists the commands of the protocol together with those of the driver definition, so it offers `Request Reply Command` even though the definition does not declare it.

Step 4: Create the Controller with Both Drivers

1. Create an **Automation Controller** named `Track-In Gateway` with the **Scope** `Connect IoT`, the entity type `Resource`, and the **Workflow Type** `Data Flow`.
2. In the driver definitions step, add `Scheduler` with the `Scheduler API` definition and `Gateway` with the `Gateway Endpoint` definition, give each a different color, and keep the core task package.

   The `Setup` page holds, for each driver, a `Driver Event` (`OnInitialize`), an `Equipment Configuration`, and a `Driver Command` (`Connect`) task, already linked.

3. In the **Communication Parameters** tab of each `Equipment Configuration` task, set the values from the table in Step 1, and select **Save**.



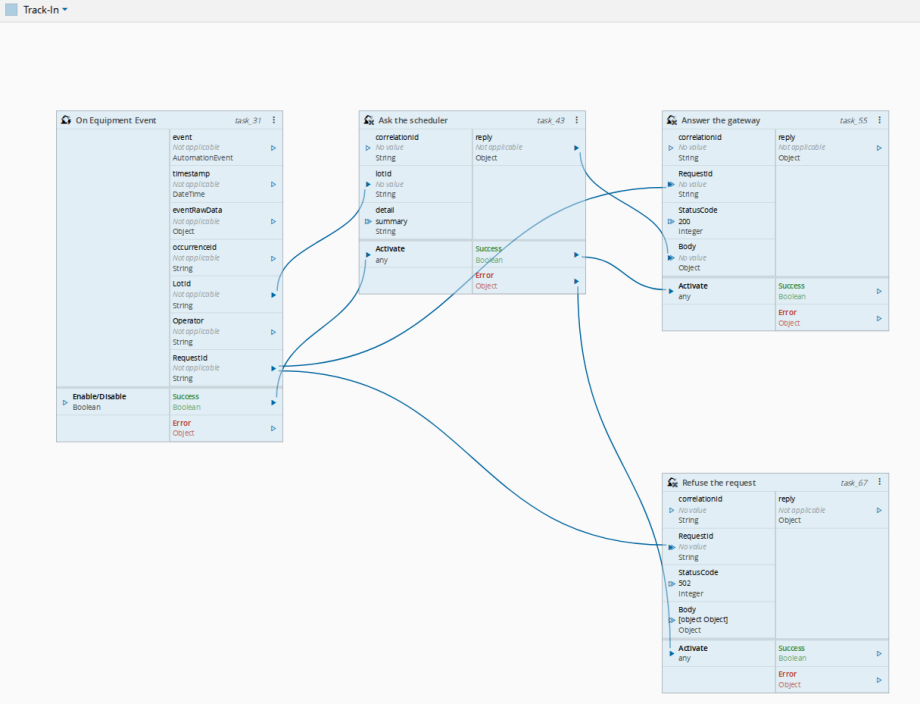
> **Info**
>
> For more information, see Create Automation Controller and Create the Setup Workflow.

Step 5: Build the Track-In Workflow

Add a workflow page named `Track-In`, place the tasks below, and link them as listed. When you drop an equipment task, choose the driver shown in the table.

Task	Settings to set	Inputs and outputs to link
`On Equipment Event` (driver `Gateway`)	**On Equipment Event**: `TrackInRequested`	-
`Execute Equipment Command`, named `Ask the scheduler` (driver `Scheduler`)	**Equipment Command**: `GetLotStatus`	`LotId` of the event to `lotId`; `Success` of the event to `Activate`
`Execute Equipment Command`, named	**Equipment Command**: `Request Reply Command`	`RequestId` of the event to `RequestId`; `reply` of `Ask`

`Answer the gateway` (driver `Gateway`)	**Settings to set**	`the scheduler` to `Body`. **Link** `Success` of `Ask the scheduler` to `Activate`
`Execute Equipment Command`, named `Refuse the request` (driver `Gateway`)	**Equipment Command**: `Request Reply Command`; on **Command Settings**, `StatusCode` `502` and `Body` `{"accepted": false}`	`RequestId` of the event to `RequestId`; `Error` of `Ask the scheduler` to `Activate`



Select **Save**. Each request now follows one of two paths. When the scheduling system answers within the `30000` milliseconds of the command `timeout`, its parsed body goes back to the gateway with `200`. When the call fails, `Refuse the request` answers `502` at once, so the gateway does not wait for the event `timeout` and then receive `500`. All the work between the event and the reply must fit within the event `timeout`.

Step 6: Connect the Instance and Start the Manager

1. Connect the controller to the **Resource** `Gateway-01` and to your **Automation Manager**, and select the same **Resource** for both drivers, as described in How to Create a Controller Instance.

2. Start the **Automation Manager** in console mode.

In **Automation** > **Connect IoT**, both driver instances reach `Communicating`. For the REST Client, this state says nothing about the scheduling system: connecting sends no request, and an unreachable server shows only as the error of a command. The REST Server listens from the moment it connects, and registers the routes of its events only when the setup completes.

Validate the Result

1. Start the scheduling system stand-in.
2. From any machine that reaches the **Automation Manager**, send a track-in request:

   ```
   curl -i -X POST http://<manager-host>:8085/gateway/lot-start/B-1001 -H
   "Content-Type: application/json" -d "{\"badgeId\": \"OP-17\"}"
   ```

   The answer is `200` with the body of the scheduling system, such as `{"lotId": "B-1001", "state": "Staged"}`. The **Automation Manager** console shows the event `TrackInRequested`, the command `GetLotStatus`, and the command `Request Reply Command`.
3. Stop the scheduling system stand-in and send the same request.

   The answer is `502` with `{"accepted": false}`, right after the failed call to the scheduling system.

Troubleshooting

Symptom	Cause	Fix
Every command fails, although the API answers from a browser	`protocol` is `HTTPS` and `port` is still `80`, or `baseUrl` holds a scheme	Write `baseUrl` as a host without a scheme, and set `port` to the real port (`443`) or `0`
The API does the work and every command still reports `Error`	The API answers a status outside `successfulStatusCodes`, such as `201` or `204`	List every success status, for example `200,201`

Symptom	Cause	Fix
Every request is answered `404`	The **Event Device ID** is not a path starting with `/`, the request arrived before the setup completed, or two events share a path and only the first one is matched	Give each event its own path, starting with `/`
The caller waits the whole event `timeout` and receives `500` with `Timeout occurred while waiting for a reply to your request!`	The event has no `RequestId` property, the reply task does not receive it, or a path of the workflow ends without a reply	Link `RequestId` into every reply task, and reply on the failure path too
`LotId` is empty, and `Ask the scheduler` fails	The **Property Device ID** of the `Path` property does not match the route segment	Use the segment name, `lotId` for `:lotId`
A reply fails with `Unable to find a pending request with id '<id>'. Maybe it timed out?`	The reply was sent after the event `timeout`, or twice for one request	Keep the work within the `timeout` and send one reply per request

Next Steps

Reuse the same pattern for HTTPS by setting `protocol`, `port`, and the certificate parameters of each driver, and set `authenticationMode` with its credentials when the API requires them. To build the same flow in Control Flow, use the `TrackInRequested` event as the trigger and place the command and the reply in its handler; see Connect IoT - Control Flow Configuration Tutorial. In 12.0, the `connect.iot.driver.template` topics of both REST drivers are deprecated and log a deprecation warning on each message, so declare every endpoint in the driver definition. To exchange data with MES itself, continue with Connect IoT - Talking to MES Tutorial, and for socket devices, see Connect IoT - TCP/IP Tutorial.

Legal Information

Disclaimer

The information contained in this document represents the current view of Critical Manufacturing on the issues discussed as of the date of publication. Because Critical Manufacturing must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Critical Manufacturing, and Critical Manufacturing cannot guarantee the accuracy of any information presented after the date of publication. This document is for informational purposes only.

Critical Manufacturing makes no warranties, express, implied or statutory, as to the information herein contained.

Confidentiality Notice

All materials and information included herein are being provided by Critical Manufacturing to its Customer solely for Customer internal use for its business purposes. Critical Manufacturing retains all rights, titles, interests in and copyrights to the materials and information herein. The materials and information contained herein constitute confidential information of Critical Manufacturing and the Customer must not disclose or transfer by any means any of these materials or information, whether total or partial, to any third party without the prior explicit consent by Critical Manufacturing.

Copyright Information

All title and copyrights in and to the Software (including but not limited to any source code, binaries, designs, specifications, models, documents, layouts, images, photographs, animations, video, audio, music, text incorporated into the Software), the accompanying printed materials, and any copies of the Software, and any trademarks or service marks of Critical Manufacturing are owned by Critical Manufacturing unless explicitly stated otherwise. All title and intellectual property rights in and to the content that may be accessed through use of the Software is the property of the respective content owner and is protected by applicable copyright or other intellectual property laws and treaties.

Trademark Information

Critical Manufacturing is a registered trademark of Critical Manufacturing.

All other trademarks are property of their respective owners.

Copyright © 2023 Critical Manufacturing. All rights reserved.