REST API Integration Guide

Send orders to the cutting machines and read the results from your own software

Interface	REST API
Document version	1.1
Date	October 2026

Contents

1. Introduction 4
1.1 What the API does 4
1.2 How it works 4
1.3 What you need from the Link administrator 5
1.4 Conventions in this guide 5
2. Quick start 6
3. Basics 10
3.1 Which request does what 11
4. Orders 12
4.1 Create an order 12
4.2 Read the state of an order 15
4.3 Cancel an order or a line 16
4.4 Confirm a cut 17
4.5 The life of a line 18
5. Events 19
5.1 What an event is 19
5.2 Read and acknowledge 19
5.3 A polling loop 21
6. Machines and machine functions 22
6.1 List the machines 22
6.2 Remove what the machine has on its screen 22
6.3 Cut screen 23
6.4 Cable diameter 23
6.5 Process time and speed 24
6.6 Errors of machine functions 24
7. Errors 25
7.1 HTTP status 25
7.2 Link codes 25
8. Debugging and troubleshooting 27
8.1 A method: follow the order step by step 27
8.2 Where the Link leaves its traces 27
8.3 My line does not start 28
8.4 Tracing the traffic 28
8.5 What to send to LOIMEX support 28
9. Testing without a machine 29

9.1 Set up the test machine 29
9.2 Useful simulator options 30
10. Going to production 31
Appendix A. Endpoint reference 32
Appendix B. Example client (Python) 33

1. Introduction

1.1 What the API does

The SoftWindingLink REST API lets your own software (an ERP, a WMS, an MES, a web shop, a script) send **cutting orders** to LOIMEX measuring and cutting machines, and read back **what was really produced**: how many cuts were made, how many metres were measured, which reel was used, when it finished and whether something went wrong.

- **Send orders** with one or several lines: product, length of each cut, number of cuts, reel, texts for the label.
- **Read the state** of an order, line by line: pending, claimed by a machine, done, error or canceled.
- **Receive events** (line done, error, canceled, label printed, machine status) with a cursor, so that nothing is lost between two reads.
- **Cancel** a pending line or order, or **confirm** a cut when the machine is configured for external confirmation.
- **Read and drive machine functions**: machine status, cut screen, cable diameter, process time and speed.

It is an ordinary HTTP/JSON interface. Any language that can send an HTTP request can use it: this guide shows `curl`, Python, PowerShell, Node.js and C#, and every example in it was run against a real SoftWindingLink with simulated machines.

1.2 How it works

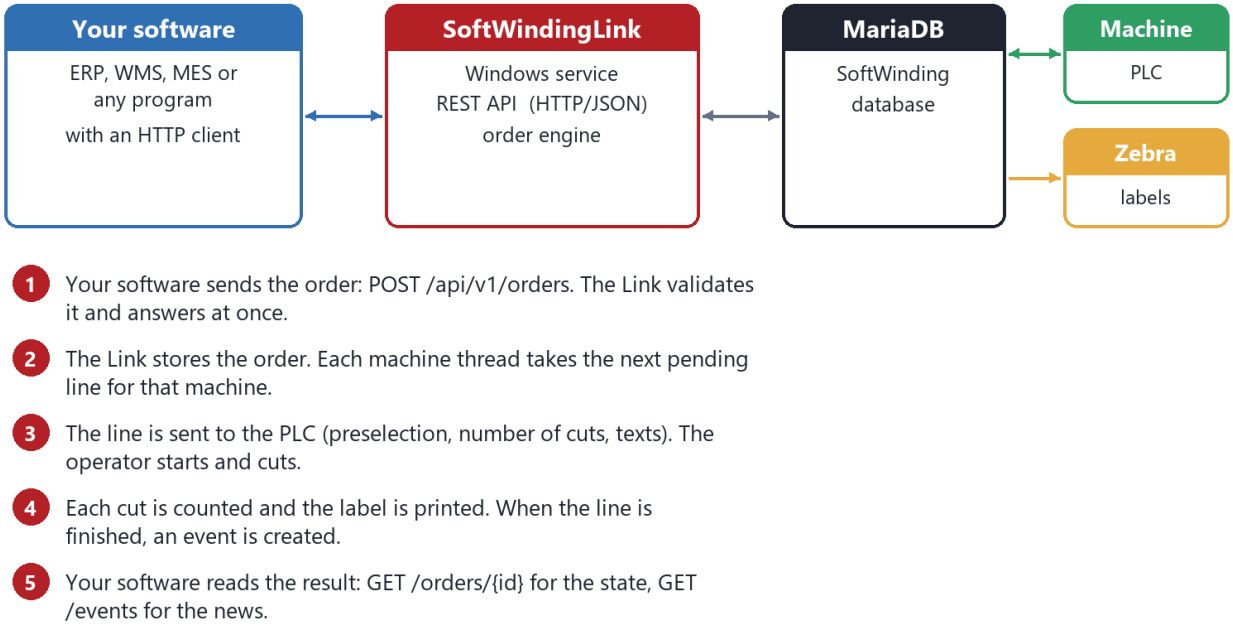


Figure 1. The path of an order through the REST API.

> **Important. The API is asynchronous.** A successful `POST /orders` means the Link has **accepted** the order, not that it has been cut. Cutting happens later, when the operator works on it. Your software finds out by reading the order state or the events: never wait for the cut inside the `POST`.

1.3 What you need from the Link administrator

Table 1. Data to ask the administrator for.

You need	Example	Where the administrator finds it
Base URL	`http://link-server:8090`	Server name or IP and `[Rest] Port` in `SoftWindingLink.ini` (8090 by default).
Token	64 hexadecimal characters	Administration window, **Clients** tab, **New**, or `SoftWindingLinkSvc.exe --newclient=ACME --channels=REST`. It is shown **only once**; the Link stores only its hash.
Machine identifiers	`M1`, `M2`	Administration window, **Machines** tab. `GET /machines` also lists the ones your client may use.
Behaviour of each machine	confirmation by counter, from the screen or external	Machines tab: *confirm mode*. It decides when a cut is considered done (section 4.5).
HTTPS (optional)	`https://link.example.com`	The REST server speaks plain HTTP. If your policy requires HTTPS, the administrator puts a reverse proxy (IIS, nginx, Caddy) with a certificate in front of it.

The REST channel is switched off by default. The administrator enables it with `[Rest] Enabled=1` (and restarts the Link service). The Link engine must be **running**: the REST server can answer while a machine is offline, but nothing is cut until the machine thread is connected to its PLC.

1.4 Conventions in this guide

- Examples use two shell variables: `LINK` (the base URL) and `TOKEN` (your token). `curl` is `curl.exe` on Windows 10 and later.
- The captured responses come from a test system with two simulated machines, `M1` and `M2`, and a client called `ACME`. Your identifiers will differ.
- The Link sends compact JSON and may write numbers in exponent form (for example `1.25E+001` for 12.5). Every standard JSON parser reads that correctly. In this guide the JSON is **pretty-printed** and numbers are written normally so that it is easy to read.

2. Quick start

Ten minutes, six requests. If you do not have a real machine, follow chapter 9 first to start the simulator, then come back here. The commands below are also in `docs\examples\rest\quickstart.sh`, and the same cycle is written in Python, PowerShell, Node.js and C# in the same folder.

```
export LINK=http://link-server:8090
export TOKEN=<your token>
```

2.1 Step 1. Is the Link there?

`GET /api/v1/health` needs no token. It tells you the version, the database schema and whether the Link is licensed (`false` means DEMO mode: it works the same and only shows a warning in the Link window).

```
curl $LINK/api/v1/health
```

```
HTTP/1.1 200 OK
{"status": "ok", "version": "0.1.0", "schema": 6, "licensed": false}
```

2.2 Step 2. Which machines can I use?

```
curl -H "Authorization: Bearer $TOKEN" $LINK/api/v1/machines
```

```
HTTP/1.1 200 OK
{
  "machines": [
    {
      "machineId": "M1",
      "description": "Cutting machine 1",
      "state": "I",
      "stateText": "ready",
      "counterM": 0.0,
      "cutNr": 0,
      "heartbeat": "2026-10-28T12:46:14"
    },
    {
      "machineId": "M2",
      "description": "Cutting machine 2",
      "state": "I",
      "stateText": "ready",
      "counterM": 0.0,
      "cutNr": 0,
      "heartbeat": "2026-10-28T12:46:14"
    }
  ]
}
```

`state` is a letter and `stateText` its explanation. `I` (ready) means the machine is connected and free. The table *Machine states* in section 4.5 lists all the states.

2.3 Step 3. Send an order

```
curl -i -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
     -d @order.json $LINK/api/v1/orders
```

`order.json` contains the request below: one order, two lines, both for machine `M1`.

```
POST /api/v1/orders
Content-Type: application/json

{
  "orderId": "SO-100234",
  "customer": "ACME Cables Ltd",
  "lines": [
    {
      "item": 10,
      "machineId": "M1",
      "productId": "H07V-K-1.5-BK",
      "reelId": "R-77881",
      "lengthM": 12.5,
      "cuts": 3,
      "description": "H07V-K 1.5 mm2 black"
    },
    {"item": 20, "machineId": "M1", "productId": "H07V-K-1.5-BU", "lengthM": 8.4, "cuts": 2}
  ]
}

HTTP/1.1 201 Created
{
  "orderId": "SO-100234",
  "batchId": "ACME_20261028",
  "lines": [
    {"item": 10, "status": "accepted", "machineId": "M1"},
    {"item": 20, "status": "accepted", "machineId": "M1"}
  ]
}
```

`201 Created` and `"status": "accepted"` mean that both lines were stored. `batchId` is the work package the Link created for your client and day.

2.4 Step 4. Follow the order

```
curl -H "Authorization: Bearer $TOKEN" $LINK/api/v1/orders/SO-100234
```

Right after the `POST` the lines are `pending`:

```
HTTP/1.1 200 OK
{
  "orderId": "SO-100234",
  "batchId": "ACME_20261028",
  "lines": [
    {
      "item": 10,
      "status": "pending",
      "machineId": "M1",
      "reelId": "R-77881",
      "lengthM": 12.5,
      "cuts": 3,
      "cutsDone": 0,
      "measuredM": 0.0
    },
    {
      "item": 20,
      "status": "pending",
      "machineId": "M1",
      "reelId": "",
      "lengthM": 8.4,
      "cuts": 2,
      "cutsDone": 0,
      "measuredM": 0.0
    }
  ]
}
```

A second or two later the machine has taken the first line (`claimed`) and the operator can start cutting:

```
HTTP/1.1 200 OK
{
  "orderId": "SO-100234",
  "batchId": "ACME_20261028",
  "lines": [
    {
      "item": 10,
      "status": "claimed",
      "machineId": "M1",
      "reelId": "R-77881",
      "lengthM": 12.5,
      "cuts": 3,
      "cutsDone": 0,
      "measuredM": 0.0,
      "startTime": "2026-10-28T12:46:17"
    },
    {
      "item": 20,
      "status": "pending",
      "machineId": "M1",
      "reelId": "",
      "lengthM": 8.4,
      "cuts": 2,
      "cutsDone": 0,
      "measuredM": 0.0
    }
  ]
}
```

When the machine has finished, both lines are `done`, with the metres really measured and the times:

```
HTTP/1.1 200 OK
{
  "orderId": "SO-100234",
  "batchId": "ACME_20261028",
  "lines": [
    {
      "item": 10,
      "status": "done",
      "machineId": "M1",
      "reelId": "R-77881",
      "lengthM": 12.5,
      "cuts": 3,
      "cutsDone": 3,
      "measuredM": 37.5,
      "startTime": "2026-10-28T12:46:17",
      "endTime": "2026-10-28T12:46:38"
    },
    {
      "item": 20,
      "status": "done",
      "machineId": "M1",
      "reelId": "",
      "lengthM": 8.4,
      "cuts": 2,
      "cutsDone": 2,
      "measuredM": 16.8,
      "startTime": "2026-10-28T12:46:38",
      "endTime": "2026-10-28T12:46:50"
    }
  ]
}
```

Line 10 asked for **3 cuts of 12.5 m**, so `measuredM` is 37.5. The value comes from the machine counter, not from your request.

2.5 Step 5. Read the events

```
curl -H "Authorization: Bearer $TOKEN" "$LINK/api/v1/events?limit=100"
```

Events are the news about everything the Link does for you. These are the ones of line 10 (the full list also has the `machine.status` events of the machine):

```
HTTP/1.1 200 OK
{
  "events": [
    {
      "eventId": 3,
      "type": "machine.status",
      "created": "2026-10-28T12:46:17",
      "machineId": "M1",
      "orderId": "SO-100234",
      "batchId": "ACME_20261028",
      "item": 10,
      "data": {"state": "P", "text": "prepared SO-100234/10", "counterM": 0.0, "cutNr": 0}
    },
    {
      "eventId": 5,
      "type": "label.printed",
      "created": "2026-10-28T12:46:25",
      "orderId": "SO-100234",
      "batchId": "ACME_20261028",
      "item": 10,
      "data": {"cutNr": 1, "printerId": "ZEB1"}
    },
    {
      "eventId": 7,
      "type": "label.printed",
      "created": "2026-10-28T12:46:32",
      "orderId": "SO-100234",
      "batchId": "ACME_20261028",
      "item": 10,
      "data": {"cutNr": 2, "printerId": "ZEB1"}
    },
    {
      "eventId": 8,
      "type": "line.done",
      "created": "2026-10-28T12:46:38",
      "machineId": "M1",
      "orderId": "SO-100234",
      "batchId": "ACME_20261028",
      "item": 10,
      "reelId": "R-77881",
      "data": {"measuredM": 37.5, "cuts": 3, "endTime": "2026-10-28T12:46:38"}
    }
  ]
}
```

2.6 Step 6. Acknowledge what you processed

```
curl -X POST -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
     -d '{"lastEventId": 15}' $LINK/api/v1/events/ack
```

The Link answers `204 No Content` and remembers the cursor. The next `GET /events` without `after` starts at event 16. Chapter 5 explains the choices.

3. Basics

Table 2. Rules that apply to every request.

Topic	Rule
Base URL and version	`http://<server>:<port>/api/v1/...`. The version is in the path. New fields can appear in responses without a version change: ignore the ones you do not know.
Transport	HTTP/1.1, **no TLS**. One request per connection (the Link closes it after answering). `Transfer-Encoding: chunked` and `Expect: 100-continue` are supported, so `HttpClient.PostAsJsonAsync` of .NET works.
Authentication	Header `Authorization: Bearer <token>` in every request except `GET /health`. One token per client. A missing or wrong token is `401`.
Content	Request and response bodies are JSON in UTF-8. Send `Content-Type: application/json` with every body.
Units	Lengths in **metres** with up to 3 decimals (`12.5`, `0.35`). Counters and measured lengths come back in metres too.
Dates	ISO 8601, **local time of the Link server, without time zone**: `2026-10-28T12:46:38`.
Identifiers	`orderId`, `batchId`, `productId`, `reelId`, `machineId`: printable ASCII (`!` to `~`), **no spaces** and none of `' " \ `` . Maximum lengths: order 30, batch 30, product 50, reel 20, machine 30. Anything else is rejected with code 4010 (or 4004 for `orderId`).
Free texts	`customer`, `description`, `notes`, `free1`...`free10`: the database is Latin-1. Accented Western European letters are kept; characters outside Latin-1 become `?`.
Idempotency	Sending the same order twice is safe (section 4.1). Use it to retry after a timeout.
Size	Request bodies up to **1 MB**.
Rate limit	**600 requests per minute** per client. Beyond that: `429`.
Authentication failures	After a run of wrong tokens from one address (in our test, the 8th attempt in a row) the Link answers `429` to **everything from that address**, with the right token too, for **5 minutes**. Do not retry a `401`: fix the token.
Error body	Always JSON with `code` and `error`: `{"code": 4001, "error": "unknown machine", ...}`. For generic HTTP errors (401, 403, 404, 429, 500) `code` is the HTTP status. Link-specific codes are 4001 to 4090 (chapter 7).

3.1 Which request does what

Table 3. Summary of the endpoints. Appendix A has the details.

Request	What it does	Answer
`POST /orders`	Create (or repeat, or change) an order	201 / 200 / 400 / 409 / 503
`GET /orders/{orderId}`	State of the order and of each line	200 / 404
`DELETE /orders/{orderId}[?item=n]`	Cancel an order or one line	200 / 404 / 409
`POST /orders/{orderId}/lines/{item}/confirm`	Confirm a cut (external confirmation)	202 / 404 / 409
`POST /orders/{orderId}/lines/{item}/cut-screen`	Show or hide the cut screen of the line	202 / 404 / 409
`GET /events[?after=n&limit=n]`	Read events after a cursor	200
`POST /events/ack`	Move your cursor	204 / 400
`GET /machines`	State of the machines you may use	200
`DELETE /machines/{id}/order`	Remove what the machine has on its screen	200 / 404 / 409
`GET` or `PUT /machines/{id}/cable`	Read or set the cable diameter	200 / 400 / 404 / 409 / 501 / 502 / 504
`GET /machines/{id}/process`	Process time and speed	200 / 404 / 409 / 501 / 502 / 504
`GET /health`	Link alive, version, schema, licence	200

4. Orders

4.1 Create an order

`POST /api/v1/orders` creates an order with one or more lines. A **line** is one cutting job: *N cuts of L metres of product P*. The pair (`orderId`, `item`) identifies the line for ever.

4.1.1 Order fields

Field	Type	Required	Meaning
`orderId`	string, 1-30	yes	Your order number. Identifies the order in every later request.
`customer`	string, up to 50	no	Customer name, stored in the order header (it can appear on labels).
`batchId`	string, up to 30	no	Work package. Default: `<clientId>_<yyyymmdd>` (for example `ACME_20261028`). Orders sent the same day share the batch.
`lines`	array	yes	One or more lines (next table).

4.1.2 Line fields

Field	Type	Required	Meaning
`item`	integer 1-99999	yes	Position of the line in the order. Unique in the order. Lines of an order are processed by `item` order (after `priority`).
`machineId`	string, up to 30	no	The machine that must cut it. Empty, missing or `ALL`: any machine that accepts general lines (the first one that is free).
`productId`	string, up to 50	yes	Product (cable) code. Printed on the label.
`lengthM`	number > 0, up to 99999.999	yes	Length of **one** cut, in metres.
`cuts`	integer 1-99999	yes	Number of cuts of that length.
`reelId`	string, up to 20	no	Reel the cable comes from. It is only a label for traceability: the Link prints it and returns it, but does not know which reel is mounted on the machine.
`priority`	integer, default 0	no	Lower numbers go first. Lines with the same priority go by `item`.
`description`	string, up to 100	no	Text for the label and for the machine screen.
`notes`	string	no	Free note.
`free1` ... `free10`	string, up to 100	no	Free fields, available to the label template.
`cableDiameterMm`	number 0-1000	no	Cable diameter in mm. The Link sends it to the machine when it prepares the line and leaves the machine in manual diameter mode.

The total length of a line is `lengthM x cuts`. The line of the example, 12.5 m x 3 cuts, ends with `measuredM` = 37.5.

4.1.3 Answers

Table 4. Answers of POST /orders.

HTTP	Meaning	What to do
201 Created	All lines were stored. Each line is `accepted`.	Done. Follow the order with `GET /orders` or the events.
200 OK	The same order, with the same content, already existed. Nothing was written.	Done. This is what a retry gets.
400 Bad Request	Validation failed: unknown machine, invalid field, invalid JSON, machine not allowed...	Read `code` and `error` (chapter 7). Fix the data. **Nothing was written.**
409 Conflict	The line already exists with other content and cannot be changed any more (4003).	Use another `item`, or accept that the first version was cut.
503 Service Unavailable	The machine is disabled or offline (4002, 4007).	Try again later or send the line to another machine.

4.1.4 Example: a rejected order

The request asks for a machine that does not exist. The answer carries the order-level `code` and, for each line, its own status:

```
POST /api/v1/orders
Content-Type: application/json

{
  "orderId": "SO-X1",
  "lines": [{"item": 10, "machineId": "M9", "productId": "P1", "lengthM": 5, "cuts": 1}]
}

HTTP/1.1 400 Bad Request
{
  "orderId": "SO-X1",
  "batchId": "ACME_20261028",
  "lines": [{"item": 10, "status": "rejected", "code": 4001, "error": "unknown machine"}],
  "code": 4001,
  "error": "unknown machine"
}
```

> **Warning.** **An order is written all or nothing.** The Link first validates **all** the lines. If any one fails, the HTTP status is not 2xx and **nothing is stored**, even if the answer shows other lines as `"status": "accepted"`: in a `4xx` answer `accepted` only means that line was valid. Fix the rejected line and send the whole order again.

```
POST /api/v1/orders
Content-Type: application/json

{
  "orderId": "SO-X5",
  "lines": [
    {"item": 10, "machineId": "M1", "productId": "P1", "lengthM": 5, "cuts": 1},
    {"item": 20, "machineId": "M9", "productId": "P1", "lengthM": 5, "cuts": 1}
  ]
}

HTTP/1.1 400 Bad Request
{
  "orderId": "SO-X5",
  "batchId": "ACME_20261028",
  "lines": [
    {"item": 10, "status": "accepted", "machineId": "M1"},
```

```
    {"item": 20, "status": "rejected", "code": 4001, "error": "unknown machine"}
  ],
  "code": 4001,
  "error": "unknown machine"
}
```

A check with `GET /orders/SO-X5` proves it: the order does not exist.

```
HTTP/1.1 404 Not Found
{"code": 404, "error": "order not found", "orderId": "SO-X5"}
```

4.1.5 Repeating and changing an order (idempotency)

The key of a line is (`client`, `orderId`, `item`). The Link keeps a fingerprint of the content of every line, so you can send the same order as many times as you need:

Table 5. What happens when an order is sent again.

You send	Result	HTTP
A new line	It is created.	201
The same line, same content	Nothing changes. The answer is the same as the first time.	200
The same line with different content, **not yet taken by a machine**	The line is updated.	201
The same line with different content, **already taken or finished**	Rejected with code 4003. The line stays as it was.	409

Use this to **retry safely** after a network error or a timeout: send exactly the same body again. If the first request had arrived, you get `200`; if not, `201`.

```
HTTP/1.1 200 OK
{
  "orderId": "SO-100234",
  "batchId": "ACME_20261028",
  "lines": [
    {"item": 10, "status": "accepted", "machineId": "M1"},
    {"item": 20, "status": "accepted", "machineId": "M1"}
  ]
}
```

Changing a line that is already done is refused:

```
POST /api/v1/orders
Content-Type: application/json

{
  "orderId": "SO-100234",
  "lines": [
    {"item": 10, "machineId": "M1", "productId": "H07V-K-1.5-BK", "lengthM": 13, "cuts": 3}
  ]
}

HTTP/1.1 409 Conflict
{
  "orderId": "SO-100234",
  "batchId": "ACME_20261028",
  "lines": [
    {"item": 10, "status": "rejected", "code": 4003, "error": "duplicate or unchangeable order"}
  ],
  "code": 4003,
  "error": "duplicate or unchangeable order"
```

```
}
```

4.1.6 Which machine cuts a line, and in which order

- With `machineId` the line goes to that machine and to no other. The machine must exist, be enabled and be allowed to your client.
- Without `machineId` (or `ALL`) the line goes to **any machine that accepts general lines** and is free. The answers show `"machineId": "ALL"` while the line is pending and the real machine once a machine has taken it. Your administrator can set a machine to take only lines addressed to it.
- Each machine takes **one line at a time**, the next one in this order: batch priority, then line `priority` (lowest first), then `item`, then order number.
- A line is *claimed* by the machine when the Link sends it to the PLC. From then on no other machine will take it.

4.2 Read the state of an order

`GET /api/v1/orders/{orderId}` returns the order and every line with its state. It is a cheap request: use it to show progress to your users or to check one order. To follow **many** orders use the events (chapter 5) instead of polling each one.

Table 6. Fields of a line in GET /orders.

Field of each line	Meaning
`item`, `machineId`, `reelId`	As sent. `machineId` is the machine that has the line (or the one you asked for while it is pending).
`status`	`pending`, `claimed`, `done`, `error` or `canceled` (next table).
`lengthM`, `cuts`	What was ordered: length of one cut and number of cuts.
`cutsDone`	Cuts made so far.
`measuredM`	Total metres measured so far, all cuts together.
`startTime`, `endTime`	When the machine took the line, and when it finished. `startTime` appears when the line is claimed, `endTime` when it is done.
`code`	Error code when `status` is `error` (chapter 7).

Table 7. Line status.

status	Meaning	Final?
`pending`	Stored and waiting for a machine.	no
`claimed`	A machine has it: it was sent to the PLC and it is being prepared or cut. `cutsDone` grows.	no
`done`	All the cuts were made and confirmed.	yes
`error`	The Link could not cut it (PLC refused it, data invalid, communication lost). See `code`.	yes
`canceled`	Canceled before cutting. A line canceled while pending disappears from the database, so `GET` answers `404` for it: the cancellation is reported by the event `line.canceled`.	yes

If the order does not exist the answer is `404` with `code` 404.

```
HTTP/1.1 404 Not Found
{"code": 404, "error": "order not found", "orderId": "NOPE"}
```

4.3 Cancel an order or a line

`DELETE /api/v1/orders/{orderId}` cancels the whole order. Add `?item=20` to cancel one line. What happens depends on the state of each line:

Table 8. Cancel, by line state.

Line state	What the Link does	How you know
`pending`	Deletes the line (and the order if it has no more lines) **immediately**.	The answer shows the remaining lines. Event `line.canceled`.
`claimed`, the machine has not measured anything yet	Asks the machine thread to release the line, remove it from the screen and delete it.	Event `line.canceled` a moment later.
`claimed`, measurement already started	The Link does **not** cancel a measurement in progress.	The request still answers `200`; shortly after, the event `line.error` with code **4011** (`measurement already started`) tells you it was refused. The line goes on.
`done`	Cannot be canceled.	`409`, code 4011.

Cancel one pending line:

```
DELETE /api/v1/orders/SO-201?item=20

HTTP/1.1 200 OK
{
  "orderId": "SO-201",
  "batchId": "ACME_20261028",
  "lines": [
    {
      "item": 10,
      "status": "pending",
      "machineId": "M1",
      "reelId": "",
      "lengthM": 5.0,
      "cuts": 2,
      "cutsDone": 0,
      "measuredM": 0.0
    }
  ]
}
```

Cancel the rest of the order (the answer is an empty list; a following `GET` is `404`):

```
DELETE /api/v1/orders/SO-201

HTTP/1.1 200 OK
{"orderId": "SO-201", "lines": []}
```

Trying to cancel a line that is being measured. The answer is `200` and shows the line still `claimed`; the refusal arrives as an event:

```
DELETE /api/v1/orders/SO-200?item=10

HTTP/1.1 200 OK
{
  "orderId": "SO-200",
  "batchId": "ACME_20261028",
  "lines": [
    {
      "item": 10,
      "status": "claimed",
      "machineId": "M1",
```

```
      "reelId": "",
      "lengthM": 30.0,
      "cuts": 4,
      "cutsDone": 0,
      "measuredM": 0.0,
      "startTime": "2026-10-28T12:46:53"
    }
  ]
}
```

```
HTTP/1.1 200 OK
{
  "events": [
    {
      "eventId": 20,
      "type": "line.error",
      "created": "2026-10-28T12:47:01",
      "machineId": "M1",
      "orderId": "SO-200",
      "batchId": "ACME_20261028",
      "item": 10,
      "code": 4011,
      "data": {"text": "measurement already started"}
    }
  ]
}
```

> **Tip. Always read the event after a cancel of a line that may be running.** The answer of `DELETE` only says that the request was accepted. To stop what the machine has on its screen, whatever the line state, use `DELETE /machines/{id}/order` (section 6.2).

4.4 Confirm a cut

Normally the Link decides that a cut is done by itself (the counter reached the preselection and the machine stopped). Machines configured with **external confirmation** wait for your system instead: the machine state is `S` (`measure reached; waiting for confirmation`) until you confirm.

```
curl -X POST -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" -d "{}" \
     $LINK/api/v1/orders/SO-300/lines/10/confirm
```

```
HTTP/1.1 200 OK
{
  "machines": [
    {
      "machineId": "M2",
      "description": "Cutting machine 2",
      "state": "S",
      "stateText": "measure reached (6.000 m); waiting for confirmation",
      "counterM": 6.0,
      "cutNr": 0,
      "currentOrderId": "SO-300",
      "currentItem": 10,
      "heartbeat": "2026-10-28T12:47:48"
    }
  ]
}
```

```
POST /api/v1/orders/SO-300/lines/10/confirm
Content-Type: application/json

{}
```

```
HTTP/1.1 202 Accepted
{"status": "requested", "orderId": "SO-300", "item": 10}
```

`202 Accepted` means that the confirmation was **requested**; the machine thread performs it in the next second. Check the line state or wait for `line.done`. If the line does not exist the answer is `404`; if the line is not waiting for a confirmation (for example it is still `pending`) the answer is `409` with code 4011. Only machines set to external confirmation act on the request.

> **Note.** The optional body field `measuredM` of the OpenAPI definition is accepted but the recorded measure is always the one of the machine counter.

4.5 The life of a line

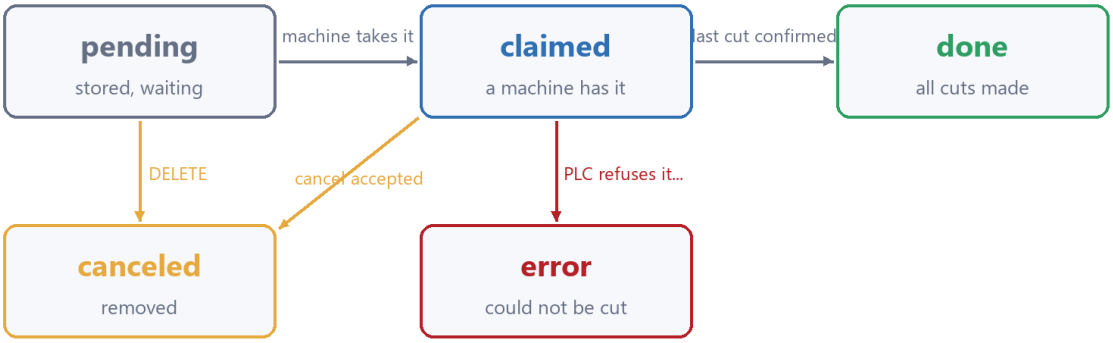


Figure 2. Status of a line as GET /orders reports it.

While a line is `claimed`, the **machine** goes through its own states, which you can read in `GET /machines` and in the `machine.status` events:

Table 9. Machine states (GET /machines and machine.status).

State	stateText	Meaning
`I`	ready	Connected and free.
`P`	prepared …	The line was sent to the PLC and the machine waits for the operator to start (or to start the next cut).
`R`	measuring x of y m	The cable is running. `counterM` grows.
`S`	measure reached; waiting for confirmation	The length was reached. With external confirmation, now it is your turn.
`E`	error…	Emergency stop, guard open or a PLC error.
`O`	offline	The Link cannot talk to the PLC.
`D`	disabled	Switched off by the administrator.

> **Warning. The operator stays in charge.** The Link prepares the machine but, by default, **never starts it**: the operator starts it, cuts and presses reset. An order can sit in `prepared` as long as nobody starts the machine. This is a safety rule, not a fault.

5. Events

5.1 What an event is

Every time something happens to your orders the Link writes an **event**. Events are kept in a queue (30 days by default) in order of an ever-growing `eventId`. You read them with `GET /api/v1/events`, which is the efficient way to follow many orders at once.

Table 10. Event types.

Event type	When	data
`line.done`	A line was finished. `machineId`, `orderId`, `item`, `reelId` are at the top level.	`measuredM`, `cuts`, `endTime`
`line.error`	The Link could not cut the line, or refused a cancel (4011). `code` at the top level.	`text`
`line.canceled`	A line was canceled.	`{}` (`measuredM` and `cuts` if it had been measured)
`label.printed`	A label was sent to the printer (one per cut).	`cutNr`, `printerId`
`machine.status`	A machine changed state (at most one every 5 seconds per machine).	`state`, `text`, `counterM`, `cutNr`
`line.waste`	The operator reset the counter before the preselection was reached: a waste, not a cut.	`valid`, `reason`, `waste`, `cutNr`, `measuredM`, `presetM`, `shortM`, `seconds`

Common fields of every event: `eventId`, `type`, `created`; and when they apply `machineId`, `orderId`, `batchId`, `item`, `reelId`, `code`, `data`. You receive the events of **your** orders and the `machine.status` events of **all** the machines (they are not filtered by the machines your client may use).

> **Warning.** Events are delivered **at least once** and in `eventId` order. Make your handler idempotent: store the last `eventId` you processed and ignore anything not greater. The `label.printed` of the last cut can arrive **after** `line.done`.

5.2 Read and acknowledge

Table 11. Reading events.

Request	Behaviour
`GET /events?after=N&limit=M`	Events with `eventId` greater than N, up to M (default 100, maximum 1000). Without `after`, the Link uses **your cursor stored in the Link**.
`POST /events/ack {"lastEventId": N}`	Moves your cursor stored in the Link to N. Returns 204.

There are two ways to work, and you must choose one:

- **You keep the cursor** (recommended when your software has a database): call `GET /events?after=<last you processed>`, process, and store the highest `eventId` yourself. You never need `/events/ack`.
- **The Link keeps the cursor**: call `GET /events` without `after` and, once the events are safely processed, `POST /events/ack`. If you forget to acknowledge, the next call returns the same events again.

The first call, without cursor, returns the oldest events:

```
HTTP/1.1 200 OK
{
  "events": [
    {
      "eventId": 1,
      "type": "machine.status",
      "created": "2026-10-28T12:45:38",
      "machineId": "M1",
      "data": {"state": "I", "text": "ready", "counterM": 0.0, "cutNr": 0}
    },
    {
      "eventId": 2,
      "type": "machine.status",
      "created": "2026-10-28T12:45:38",
      "machineId": "M2",
      "data": {"state": "I", "text": "ready", "counterM": 0.0, "cutNr": 0}
    }
  ]
}
```

After acknowledging event 20, the same call without `after` starts at 21:

```
POST /api/v1/events/ack
Content-Type: application/json

{"lastEventId": 20}

HTTP/1.1 204 No Content
```

```
HTTP/1.1 200 OK
{
  "events": [
    {
      "eventId": 21,
      "type": "label.printed",
      "created": "2026-10-28T12:47:05",
      "orderId": "SO-200",
      "batchId": "ACME_20261028",
      "item": 10,
      "data": {"cutNr": 1, "printerId": "ZEB1"}
    },
    {
      "eventId": 22,
      "type": "machine.status",
      "created": "2026-10-28T12:47:06",
      "machineId": "M1",
      "orderId": "SO-200",
      "batchId": "ACME_20261028",
      "item": 10,
      "data": {"state": "P", "text": "prepared; counter 0.000 m", "counterM": 0.0, "cutNr": 1}
    }
  ]
}
```

With an explicit cursor you read from any point:

```
HTTP/1.1 200 OK
{
  "events": [
    {
      "eventId": 31,
      "type": "label.printed",
      "created": "2026-10-28T12:47:39",
      "orderId": "SO-200",
      "batchId": "ACME_20261028",
      "item": 10,
      "data": {"cutNr": 4, "printerId": "ZEB1"}
    }
  ]
}
```

A bad acknowledgement body is a `400`:

```
POST /api/v1/events/ack
Content-Type: application/json

{"x": 1}

HTTP/1.1 400 Bad Request
{"code": 4004, "error": "lastEventId is required"}
```

5.3 A polling loop

```
last = load_cursor()                        # your stored cursor, 0 the first time
while True:
    events = GET /events?after={last}&limit=200
    for e in events:
        handle(e)                           # idempotent; e.type == "line.done" -> update your
order
        last = e.eventId
    save_cursor(last)                       # after handling, never before
    sleep(2 if events else 5)
```

Poll every 1 to 5 seconds. The limit of 600 requests per minute leaves room for many orders; do not poll each order separately when events can tell you.

6. Machines and machine functions

6.1 List the machines

`GET /api/v1/machines` returns the machines **your client may use** with their current state. It is a good first call to check connectivity and permissions.

```
HTTP/1.1 200 OK
{
  "machines": [
    {
      "machineId": "M1",
      "description": "Cutting machine 1",
      "state": "R",
      "stateText": "measuring 0.000 of 12.500 m",
      "counterM": 0.0,
      "cutNr": 0,
      "currentOrderId": "SO-100234",
      "currentItem": 10,
      "heartbeat": "2026-10-28T12:46:17"
    },
    {
      "machineId": "M2",
      "description": "Cutting machine 2",
      "state": "I",
      "stateText": "ready",
      "counterM": 0.0,
      "cutNr": 0,
      "heartbeat": "2026-10-28T12:46:16"
    }
  ]
}
```

Table 12. Fields of a machine.

Field	Meaning
`machineId`, `description`	Identifier to use in `machineId` and a text.
`state`, `stateText`	Letter and text (see *Machine states* in section 4.5).
`counterM`, `cutNr`	Metres on the counter and number of cuts of the current line, when known.
`currentOrderId`, `currentItem`	The line the machine has now (missing when it has none).
`heartbeat`	Last time the Link updated the machine. If it is older than about 30 seconds the Link engine is not running.

6.2 Remove what the machine has on its screen

`DELETE /api/v1/machines/{machineId}/order` clears the order the machine shows, **without** giving an order number, and cancels the line even if it has already been measured or cut. Use it when a screen is left with an order nobody wants. The PLC is left clean.

- It is refused with **409, code 4014** if the machine is **running**: stop it first.
- It needs the Link engine to be running and connected to the machine (409 if the machine has no heartbeat, 504 if it does not answer in 6 seconds).
- Answer: `{"machineId": "M1", "canceled": true, "detail": "canceled SO-1/10"}`, or `"detail": "cleared"` if the screen had no line of the Link.

```
DELETE /api/v1/machines/M1/order
```

```
HTTP/1.1 409 Conflict
{"code": 4014, "error": "running: stop the machine first"}
```

6.3 Cut screen

`POST /api/v1/orders/{orderId}/lines/{item}/cut-screen` with `{"on": true}` shows the **cut screen** of the machine for that line, and `{"on": false}` hides it. Without a body it means `true`. The Link also switches the screen on by itself when the counter reaches the preselection; this request lets you force it. It answers `202` and works only on a line a machine has now (`409`, code 4011, `the line is not active on a machine`, otherwise).

```
POST /api/v1/orders/SO-300/lines/10/cut-screen
Content-Type: application/json

{"on": true}

HTTP/1.1 202 Accepted
{"status": "requested", "orderId": "SO-300", "item": 10, "cutScreen": true}
```

6.4 Cable diameter

Machines with a cable gauge can receive the diameter of the cable. Either send it with every line (`cableDiameterMm`) or set it directly:

Table 13. Cable diameter requests (POST works like PUT).

Request	Effect
`GET /machines/{id}/cable`	Current diameter (`diameterMm`), the raw PLC value (`raw`) and the mode: `laser` true when the laser gauge measures it, `manual` true when a diameter was sent.
`PUT /machines/{id}/cable` `{"diameterMm": 10.5}`	Sends the diameter (0 to 1000 mm) and switches to manual mode.
`PUT /machines/{id}/cable` `{"laser": true}`	Back to the laser gauge.

```
PUT /api/v1/machines/M2/cable
Content-Type: application/json

{"diameterMm": 10.5}

HTTP/1.1 200 OK
{"machineId": "M2", "laser": false, "manual": true, "diameterMm": 10.5, "raw": 105}
```

```
PUT /api/v1/machines/M2/cable
Content-Type: application/json

{"laser": true}

HTTP/1.1 200 OK
{"machineId": "M2", "laser": true, "manual": false, "diameterMm": 12.5, "raw": 125}
```

```
PUT /api/v1/machines/M2/cable
Content-Type: application/json
```

```
{"diameterMm": 5000}

HTTP/1.1 400 Bad Request
{"code": 400, "error": "\"diameterMm\" must be between 0 and 1000"}
```

If the machine profile has no cable parameters the Link answers `501` (code 4015): the function does not exist on that machine.

6.5 Process time and speed

`GET /api/v1/machines/{id}/process` returns the process time of the current order (`mm:ss`) and the speed in metres per minute. Fields that the machine does not provide are missing.

```
HTTP/1.1 200 OK
{
  "machineId": "M1",
  "processTime": "00:03",
  "processSeconds": 3,
  "speedMPerMin": 240.0,
  "speedRaw": 240,
  "currentOrderId": "SO-200",
  "currentItem": 10
}
```

6.6 Errors of machine functions

Table 14. Errors of the machine requests.

HTTP	Cause
404	The machine does not exist or is not allowed to your client (code 4001).
409	The Link has no heartbeat from the machine (engine stopped), or the machine is running (4014).
501	The machine profile does not support the function (4015).
502	The PLC refused the request.
504	The machine did not answer in 6 seconds. The request is canceled and will not run later.

7. Errors

7.1 HTTP status

Table 15. HTTP status codes.

HTTP	Meaning	Retry?
200, 201, 202, 204	Success. 202: the request was accepted and is being performed.	-
400	Invalid data, invalid JSON, or the order was rejected (see `code`).	No: fix the data.
401	Missing or wrong token.	No: fix the token.
403	The client is not allowed from this IP address.	No: ask the administrator.
404	Order, line or machine not found (or not yours).	No.
409	Conflict: the line cannot be changed or canceled, the machine is running, or the machine is not waiting.	Depends on `code`.
429	Too many requests or too many failed authentications.	Yes, after waiting (up to 5 minutes).
501	The function does not exist on that machine.	No.
502, 503, 504	PLC refused, machine unavailable, machine did not answer.	Yes, with a wait.
500	Internal error of the Link (code 4090).	Yes, and report it if it persists.

7.2 Link codes

Table 16. Error codes of the Link.

Code	Meaning	What to do
`4001`	Unknown machine	Check `machineId` with `GET /machines`.
`4002`	Machine disabled or offline	Retry later, or use another machine.
`4003`	Order duplicated or no longer modifiable	The line was already taken: do not resend it with other content.
`4004`	Invalid field (length, cuts, product, item, JSON, `cableDiameterMm`)	Read `error` for the field. `lengthM` must be > 0 and <= 99999.999; `cuts` 1 to 99999; `item` >= 1.
`4005`	The reel belongs to another product	Only when the administrator turned on the reel check: send the right `reelId`/`productId`.
`4006`	Line lost (another station changed it)	Report to the administrator.
`4007`	Communication error with the PLC	Check the machine network. The Link retries.
`4008`	Emergency stop or guard open	The operator must clear it.
`4009`	No confirmation in the expected time	Confirm the cut (external confirmation) or ask the operator.
`4010`	Invalid character in an identifier	Use only printable ASCII without spaces.
`4011`	The line cannot be canceled	It is being measured or already done.
`4012`	Machine not allowed to this client	Ask the administrator to allow it, or use another machine.
`4013`	Stock control is off	Only for stock entries (database channel).

Code	Meaning	What to do
4014	The machine is running	Wait until it stops.
4015	The machine does not support the function	The machine profile lacks the parameters.
4090	Internal error	Retry; report it.

8. Debugging and troubleshooting

8.1 A method: follow the order step by step

When "nothing happens", do not guess: follow the order along its path, from your software to the machine, and find the first step where it stops.

Table 17. The five steps of a diagnosis.

Step	Question	Where to look
1	**Did the Link receive it?** Look at the HTTP status of your own request and read the body: a `2xx` means it was stored; a `4xx` carries `code` and `error`.	Your own HTTP log of the exchange. On the Link: the log line `REST <ip> POST /api/v1/orders 201 31ms`.
2	**Is the order stored?**	Administration window, **Orders** tab (filter by order number), or the view `sw_link_v_orders` in the database.
3	**Is a machine working on it?**	Administration window, **Machines** tab: state and current order. In the database: the view `sw_link_v_machines`.
4	**What did the Link do?**	Administration window, **Log** and **Events** tabs; the daily log file `logs\link_YYYYMMDD.log`.
5	**What did the machine answer?**	The log lines tagged with the machine, for example `[M1] ...`, and the machine state text.

8.2 Where the Link leaves its traces

Table 18. Diagnostic tools.

Place	What it shows
Administration window	**Status**: engine, database and licence. **Machines**: state, counter and current line of each machine, with a connection test. **Orders**: every line with its status, searchable. **Clients**: your client, its channels, allowed IPs and machines. **Events**: the events your system receives. **Log**: the Link log with a level filter and a button that opens the log folder.
Log file `logs\link_YYYYMMDD.log`	One line per fact: starts, connections, every request of the channel, every order accepted, every line sent to the PLC, every cut confirmed, warnings and errors. One file per day, kept 30 days.
Table `sw_link_log`	The engine part of the log, queryable with SQL: `SELECT * FROM sw_link_log ORDER BY LogID DESC LIMIT 50`.
Events (`sw_link_events`, API, files, messages)	The business-level history: `line.done`, `line.error`, `line.canceled`, `label.printed`, `machine.status`, `line.waste`.
Views `sw_link_v_orders`, `_cuts`, `_waste`, `_machines`	The current state of lines, cuts made, wastes and machines, as plain SQL.
`SoftWindingLinkSvc.exe --console`	Runs the engine in a window instead of as a service, to watch it live (stop with Enter). Combine with `--ini=<path>` to use a test configuration.

Typical engine lines for an order that goes well (excerpt of the log of the test system):

```
12:46:16 INF REST order SO-100234 from ACME: 2 line(s) accepted
12:46:17 INF [M1] claimed ACME_20261028|SO-100234|H07V-K-1.5-BK|10
12:46:17 INF [M1] line ACME_20261028|SO-100234|H07V-K-1.5-BK|10 sent to the PLC:
              H07V-K-1.5-BK x 3 cuts of 12.500 m (preselection 1250)
12:46:25 INF [M1] cut 1/3 confirmed: 12.500 m
```

```
12:46:38 INF [M1] line ACME_20261028|SO-100234|H07V-K-1.5-BK|10 DONE: 3 cuts, 37.500 m
```

8.3 My line does not start

The line is stored (`pending`) but no machine takes it, or the machine takes it and nothing is cut. Check, in this order:

Table 19. Why a line does not start.

What you see	Probable cause	What to do
Machine state `O` (offline) or `D` (disabled), or a heartbeat older than 30 s	The Link cannot reach the PLC, the machine is switched off in the Link, or **the Link service is not running**.	Check the service, the machine network and the **Machines** tab (use the connection test).
The line stays `pending`; the machine is free (`I`)	The line is for another machine, or has no `machineId` and the machine accepts only addressed lines, or an earlier line is still ahead of it (batch priority, `priority`, `item`).	Compare `machineId` with the machine; look at the **Orders** tab ordering.
The machine is `P` (`prepared ...`) and nothing moves	The line was sent to the PLC and the machine waits for the **operator to start it**. The Link does not start machines.	Ask the operator to start. This is normal.
The machine shows the order list on its screen and waits	The machine works in **screen mode**: the operator chooses the order on the machine panel.	The operator picks and starts the order on the screen.
The machine is `S` (`waiting for confirmation`)	The machine is set to external confirmation and waits for your system.	Confirm the cut (REST `confirm`, mailbox action `K`...).
The machine is `E`	Emergency stop, open guard or a PLC error.	The operator clears the cause; the Link continues.
The line is `error`	The PLC refused it or the data were invalid (see `code` and the `line.error` event).	Read the code in chapter 7 (REST guide) / the error table of this guide and the Link log.
The first line works but the next one never starts	The machine waits for the operator after each line (reset, next order) or a previous line is still `claimed`.	Look at the machine state and current order.

8.4 Tracing the traffic

8.5 What to send to LOIMEX support

- Link version (`GET /health` or the window title) and Windows version.
- The order number, the line (`item`) and the machine.
- The exact request and the **complete answer** (headers and body), or the row / file / telegram / message you sent.
- The log file of the day (`logs\link_YYYYMMDD.log`) and, if possible, an export of the **Events** tab.
- What you expected and what happened, with the time (local time of the server).

9. Testing without a machine

You do not need a real machine to develop. LOIMEX provides **simulators** that behave like a cutting machine and like a customer system. With them you can run every example in this guide at your desk, repeat a failing case as many times as you like and see events appear in the Link window.

Table 20. Simulators (installed in the Link folder with the "Simulators and diagnostic tools" component of the installer).

Program	What it is	Typical use
`SoftWindingLinkSim.exe`	Machine simulator (console). Speaks the PLC protocols of the Omron NX/CP (FINS over UDP, port 9600) and of PLCs without FINS (SFW over TCP, port 63000) and simulates a Zebra printer on TCP 9100. With `--auto` an automatic "operator" starts, measures and resets the counter by itself.	Unattended tests and scripts.
`SoftWindingLink_MachineSimulator.exe`	The same simulator with a **window**: counter, preselection and cut count, with buttons to start, stop, reset the counter, reach the length, cut and edit values. The right panel shows the order the Link writes to the machine.	You play the operator by hand.
`SoftWindingLink_ServerSimulator.exe`	Customer-system simulator with a window: sends orders by database, REST, TCP (AHP) and MQTT and shows the answers.	Compare your integration with a known good client.

9.1 Set up the test machine

1. **Start the machine simulator** in a command prompt: `SoftWindingLinkSim.exe --auto --speed=400 --start-delay=1500 --cut-delay=2000 --printer-port=9100`. It listens on `127.0.0.1`. `--speed` is the counter speed in counter units per second (hundredths of a metre: 400 is 4 m/s; default 2000), `--start-delay` the time before the automatic operator starts, `--cut-delay` the time the operator takes to cut and reset. Add `--noconsole` to run it in the background (otherwise it waits for commands typed in its console) and `--run=SECONDS` to make it stop by itself.
2. **Add the machine to the Link.** In the administration window, **Machines** tab, create `M1`: PLC type *Omron NX1 (FINS)*, address `127.0.0.1`, port `9600`, profile `OMRON_CP`, confirm mode *counter*, cut mode *cut by cut*, enabled. Use the connection test: it must answer. (The database equivalent is the `INSERT INTO sw_link_machines ...` shown in the Link manual, section 4.)
3. **Create your client** in the **Clients** tab (or `SoftWindingLinkSvc.exe --newclient=ACME --channels=REST`) and keep the access data it shows.
4. **Check the channel** is enabled in the **Settings** tab and that the engine is running (Status tab).
5. **Run the examples.** The line goes `pending`, `claimed` and `done` in a few seconds; the simulated machine counts every cut and the simulated Zebra receives the labels.

> **Warning.** **Never point a test client at a Link that drives real machines**, unless the machine identifiers you use are reserved for tests: on a real machine an operator could start your test line. Use a separate test Link, or machines that exist only for testing.

9.2 Useful simulator options

Table 21. Machine simulator options.

Option	Effect
`--auto`	Automatic operator: starts, measures up to the preselection, confirms the cut and resets.
`--speed=N`, `--start-delay=MS`, `--cut-delay=MS`	Counter speed and operator delays. Slow values let you watch a line in progress.
`--fins-port=N`, `--sfw-port=N`	Ports. Use different ports to run several simulated machines on one PC (`M1` on 9600, `M2` on 9601).
`--printer-port=9100`, `--printer-file=labels.txt`	Simulated Zebra: the received ZPL is appended to the file.
`--waste=N`, `--wastes=K`	The cut is interrupted at N counter units and the operator resets the counter: a waste (a reset before the preselection). K = how many times (1 by default).
`--noconsole`, `--run=SECONDS`	Without them the simulator waits for typed commands in its console. Use `--noconsole` to run it in the background, and `--run` to make it stop after that many seconds.
`--listen=IP`	Address to listen on (default `127.0.0.1`).

10. Going to production

A checklist for your software:

Table 22. Production checklist.

Topic	Recommendation
Timeouts and retries	Use a timeout of 10 to 15 seconds. Retry `POST /orders` with exactly the same body on timeout or `5xx`: it is idempotent. Never retry a `4xx`.
Do not wait in the request	`POST /orders` returns when the order is stored. Follow the result with events.
Cursor	Keep your own event cursor in your database and save it **after** processing. Handle duplicates.
Polling	1 to 5 seconds for events. Do not poll every order. Respect 600 requests per minute.
Identifiers	Build `orderId` from your own order number. Keep `item` stable if you ever resend a line.
Clock	Dates are the local time of the Link server, without a zone. Do not compare them with UTC.
Security	Store the token as a secret. Use HTTPS through a reverse proxy if the network is not trusted. Ask the administrator to restrict the client to your IP address (`AllowedIPs`) and to your machines (`AllowedMachines`).
Monitoring	Call `GET /health` and `GET /machines` regularly. A `heartbeat` older than 30 seconds, or state `0`, needs attention.
Logging	Log the whole response body of every `4xx`: the reason of a rejection is **not** written in the Link log (section 8.2).
Separation	One client per system. The same machine can be used by several clients: it works on one line at a time.

Appendix A. Endpoint reference

Table 23. All the endpoints.

Endpoint	Request	Success answer	Errors
`GET /health`	no token	200 `{status, version, schema, licensed}`	-
`POST /orders`	order JSON	201 / 200 `{orderId, batchId, lines[{item, status, machineId}]}`	400 401 403 409 429 503
`GET /orders/{orderId}`	-	200 `{orderId, batchId, lines[...]}`	401 404
`DELETE /orders/{orderId}?item=`	-	200 order state (or `lines: []`)	401 404 409
`POST /orders/{o}/lines/{i}/confirm`	`{}`	202 `{status: requested, orderId, item}`	401 404 409
`POST /orders/{o}/lines/{i}/cut-screen`	`{"on": true}`	202 `{status, orderId, item, cutScreen}`	400 401 404 409
`GET /events?after=&limit=`	-	200 `{events: [...]}`	401 429
`POST /events/ack`	`{"lastEventId": n}`	204	400 401
`GET /machines`	-	200 `{machines: [...]}`	401
`DELETE /machines/{id}/order`	-	200 `{machineId, canceled, detail}`	401 404 409 502 504
`GET /machines/{id}/cable`	-	200 `{machineId, laser, manual, diameterMm, raw}`	401 404 409 501 502 504
`PUT` or `POST /machines/{id}/cable`	`{"diameterMm": n}` or `{"laser": true}`	200 as GET	400 401 404 409 501 502 504
`GET /machines/{id}/process`	-	200 `{machineId, processTime, processSeconds, speedMPerMin, ...}`	401 404 409 501 502 504

Everything except `/health` needs `Authorization: Bearer <token>`. The formal definition is in `docs\openapi.json` (OpenAPI 3): import it into Postman, Insomnia, Swagger UI or a code generator. The behaviour described in this guide is the one verified on a running system; where the OpenAPI text differs in a detail (for example the answer code of `confirm`), follow this guide.

Appendix B. Example client (Python)

A complete client in a single file, standard library only. The same cycle in PowerShell, Node.js and C# is in `docs\examples\rest`. The code below is the reusable part (`link_client.py`); the file also has a `main()` that sends a demo order and prints the result.

```
import json
import os
import sys
import time
import urllib.error
import urllib.request



class LinkError(Exception):
    """The Link answered with an error. .status is the HTTP status, .code the Link code (4001...), .body the
JSON."""

    def __init__(self, status, body):
        self.status, self.body = status, body
        self.code = body.get('code') if isinstance(body, dict) else None
        super().__init__('HTTP %s: %s' % (status, body))



class LinkClient:
    def __init__(self, base_url, token, timeout=15):
        self.base = base_url.rstrip('/') + '/api/v1'
        self.token = token
        self.timeout = timeout


    def _call(self, method, path, body=None):
        data = None if body is None else json.dumps(body).encode('utf-8')
        req = urllib.request.Request(self.base + path, data=data, method=method)
        req.add_header('Authorization', 'Bearer ' + self.token)
        if data is not None:
            req.add_header('Content-Type', 'application/json')
        try:
            with urllib.request.urlopen(req, timeout=self.timeout) as resp:
                raw = resp.read().decode('utf-8')
                return resp.status, (json.loads(raw) if raw.strip() else None)
        except urllib.error.HTTPError as e:           # 4xx and 5xx also carry a JSON body
            raw = e.read().decode('utf-8', 'replace')
            try:
                payload = json.loads(raw)
            except ValueError:
                payload = {'error': raw}
            raise LinkError(e.code, payload) from None


    # ---- orders ----
    def create_order(self, order_id, lines, customer='', batch_id=None):
        """POST /orders. Safe to repeat: the same content answers 200 instead of 201."""
        body = {'orderId': order_id, 'lines': lines}
        if customer:
            body['customer'] = customer
        if batch_id:
            body['batchId'] = batch_id
        return self._call('POST', '/orders', body)


    def order(self, order_id):
        return self._call('GET', '/orders/' + order_id)[1]


    def cancel(self, order_id, item=None):
        return self._call('DELETE', '/orders/%s%s' % (order_id, '' if item is None else '?item=%d' % item))[1]


    def confirm(self, order_id, item):
        return self._call('POST', '/orders/%s/lines/%d/confirm' % (order_id, item), {})[1]


    # ---- events ----
    def events(self, after=None, limit=100):
        q = '?limit=%d' % limit + ('' if after is None else '&after=%d' % after)
```

```
        return self._call('GET', '/events' + q)[1]['events']

    def ack(self, last_event_id):
        self._call('POST', '/events/ack', {'lastEventId': last_event_id})

    # ---- machines ----
    def machines(self):
        return self._call('GET', '/machines')[1]['machines']

    def health(self):
        # no token needed, but sending it does no harm
        return self._call('GET', '/health')[1]



def wait_until_finished(link, order_id, timeout=120, poll=2.0):
    """Polls the order until every line is done, in error or canceled. Returns the order state."""
    end = time.time() + timeout
    while time.time() < end:
        state = link.order(order_id)
        if all(l['status'] in ('done', 'error', 'canceled') for l in state['lines']):
            return state
        time.sleep(poll)
    raise TimeoutError('order %s did not finish in %s s' % (order_id, timeout))
```

Use it like this:

```
link = LinkClient("http://link-server:8090", token)
status, accepted = link.create_order("SO-100234", [
    {"item": 10, "machineId": "M1", "productId": "H07V-K-1.5-BK", "lengthM": 12.5, "cuts": 3}],
    customer="ACME Cables Ltd")
state = wait_until_finished(link, "SO-100234")
for e in link.events(after=last_cursor):
    handle(e)
```