

Service requests (workflows)

Turn a recurring request - a new laptop, an onboarding, a firewall change - into an offer with fixed steps. Everyone can ask for it in one form; the right teams get their step in the right order, and automation such as n8n can take part.

Overview

A **service request offer** (the template) describes the steps, who does them, in which order and what the requester has to tell you. A **request** is one run of an offer. It consists of

- a **master task** that carries the requester's answers and closes itself when the last step is done, and
- one **step task** per step, in the team of that step. Steps without a predecessor are active at once; the others wake up when the step before them is done.

Who	Does what
Designer (role Workflows , or RC Admin)	Builds the offers, chooses who may start them, sets up triggers, watches the trigger dashboard.
Requester	Starts a request from an offer they may use, follows it under My service requests .
Team members	Work their step like any task. Finishing it wakes the next step.

1. Design an offer

Open **Service Requests → Overview** and click **New service request offer** (you need the role **Workflows**).

1 **Name and description** are what requesters see in the list. Tags help to find the offer.

- 2 **Who may run it:** choose the roles that may start the offer, or leave it open for everybody. The list shows an offer only to people who may start it, and the server checks it again when the form is sent. An RC Admin may always start every offer.
- 3 **Request intake fields** are what the requester fills in: *text* (optionally limited by a regular expression, e.g. an IP address), *number*, *date* or *dropdown*. All of them are required, and are checked on the server, not only in the browser. The field's **name** is also a variable for trigger URLs (see below).
- 4 **Steps:** for each step choose the team, a title and description, the estimate, and which step it **depends on** (none = it starts at once). The estimate also drives the progress bar of the request.
- 5 **Step fields** (per step): values the person working the step must fill in before they can finish it, for example a ticket number or a serial number. A step with an empty step field cannot be finished - not on the Done button and not through the API.
- 6 **Trigger** (per step, optional): an address that is called when the step is finished, see section 5.

Changing an offer does not change requests that are already running: they keep the steps they were started with.

2. Start a request

Click an offer in **Service Requests → Overview**. Give the request a title and a description, choose the **solution** (the service it is for) and fill in the request fields.

It is all or nothing: if a field is empty or has the wrong format, you get the reason and nothing has been created. Otherwise you get the master task and every step task; the first steps are active and appear in their teams' pools. Each task gets the usual "task created" entry, and automation is told (section 7).

3. Work the steps

A step is a normal task in its team: it can be taken from the pool, assigned, commented and booked on. It carries the requester's answers in the master task's first comment.

- Fill in the **step fields** of the step, then click **Done**.
- If the step has a trigger of the kind *call before finishing*, it is called now; the step only counts as done when the address answers 200 (section 5).
- The steps that waited for this one become active. When the last open step is done, the master task closes and the request is finished.

4. Follow your requests

Service Requests → **My service requests** shows **one row per request** in two tables: open and finished. A row has

- the title, the offer, the customer and the request number,
- a **progress bar**: the share of the **estimated effort** that is done, not of the steps. If the first of two steps is estimated at 2 minutes and the second at 20 hours, finishing the first is about 0 %. 100 % appears only when everything is done. Without any estimate the bar counts steps instead,
- a chip for every step: done (struck through), being worked on (highlighted, with the person), or waiting for the step before. A step with a warning sign ⚠ has a trigger that failed; hover for the reason and the time.

5. Step triggers

A trigger lets another system react when a step is finished: start a provisioning job, create an account, tell a chat channel. Per step you choose **how** it is sent:

	Call before finishing	Send as webhook
--	-----------------------	-----------------

Sent as	<code>GET</code> , right when someone clicks Done	<code>POST</code> with a JSON body, by the scheduler (within about a minute)
The step is done	only if the address answers 200 within 10 seconds; otherwise it stays open and the person sees why	at once, whatever the receiver does
On failure	nothing is retried; press Done again	retried after 1 minute, 5 minutes, 30 minutes, 2 hours and 12 hours (six attempts); then it is marked <i>given up</i> and the task gets a note
Signed	no	yes (<code>X-RC-Signature</code>)
Use it when	the next step must not start unless the other system has accepted the order	the other system should just be told, and may be down for a while

Every step that existed before this option behaves as *call before finishing*.

The address and its variables

Write the address with the field names in angle brackets, for example

`https://automation.example.com/hook/laptop?size=<size>&ticket=<ticket>` . When the trigger runs, each `<name>` is replaced by the URL-encoded value of that field: the requester's request fields and the step fields filled in so far (a step's own value wins over a request field of the same name).

The address must be `http` or `https` without a user name or password, and - unless the instance allows private addresses (setting `webhook_allow_private` or the environment variable `RC_WEBHOOK_ALLOW_PRIVATE=1` , for an intranet n8n) - a public one. The check is made again at delivery, and the connection goes to the address that was checked. Up to three redirects are followed in *call before finishing* mode and each one is checked; the webhook mode follows none.

Anything secret belongs in the signature, not in the address: the address is shown to designers and its query string is never written to the logs or the task history (only scheme, host and path are).

What the webhook carries

```
POST /hook/laptop HTTP/1.1
Content-Type: application/json
X-RC-Event: workflow.step_trigger
X-RC-Delivery: 5f0c...      (the same on every retry)
X-RC-Attempt: 1
X-RC-Signature: t=1760437200,v1=9b1f...

{ "id": "5f0c...", "event": "workflow.step_trigger", "created": "2026-10-14T10:20:00+00:00",
  "instance": "https://rc.example.com",
  "data": {
    "id": 13541, "title": "Order the laptop", "type": "Service Request", "team_id": 7,
    "workflow": { "request_id": 13538, "template_id": 4, "template": "New laptop",
                  "kind": "step", "step": 2, "steps": 3,
                  "fields": { "size": "M", "ticket": "INC-7" } },
    "trigger": { "request_id": 13538, "step": 2 } } }
```

Verify the signature before you trust the body. It is the same scheme as for every webhook of the instance: `HMAC-SHA256` over `t + "." + body` with the instance's trigger secret, hex-encoded. Refuse a timestamp that is more than a few minutes old.

```
# Python
import hmac, hashlib, time
def valid(secret, header, body, tolerance=300):
    parts = dict(p.split("=", 1) for p in header.split(","))
    if abs(time.time() - int(parts["t"])) > tolerance:
        return False
    expected = hmac.new(secret.encode(), (parts["t"] + ".").encode() + body, hashlib.sha256)
    return hmac.compare_digest(expected, parts["v1"])
```

```
// Node
const crypto = require('crypto');
function valid(secret, header, body, tolerance = 300) {
    const p = Object.fromEntries(header.split(',').map(x => x.split('=')));
    if (Math.abs(Date.now() / 1000 - Number(p.t)) > tolerance) return false;
    const expected = crypto.createHmac('sha256', secret).update(p.t + '.' + body).digest('hex');
    return crypto.timingSafeEqual(Buffer.from(expected), Buffer.from(p.v1));
}
```

The **secret** is on the trigger dashboard (next section). An RC Admin can create a new one; every receiver must then be given it.

Answer with any **2xx** status to confirm. Any other answer, no answer within 10 seconds, or no connection counts as a failed attempt; **410 Gone** gives up at once.

6. The trigger dashboard

Service Requests → Workflow Triggers (role **Workflows** or RC Admin) shows how the triggers of all offers went.

- **Tiles** for the chosen period (7, 30 or 90 days, optionally one offer, optionally failures only): runs, succeeded, failed, success rate, average time and the number of steps stuck.
- **Steps stuck on a failed trigger**: open steps of the kind *call before finishing* whose last run failed. **Run again** calls the address without finishing the step, so you can test the fix; when it works, finish the step.
- **Webhook triggers not delivered yet**: deliveries that wait for their first attempt or a retry, or were given up. **Send again** starts a fresh run of attempts.
- **By offer** and **recent runs**: what was called (without query string), the result, the time, who ran it again.
- The **signing secret** (hidden until you ask) and, for RC Admins, the button for a new one.

Failures say **why**:

Reason	Meaning
the address answered 500	It was reached but did not answer 200 (call mode) or 2xx (webhook). Look at the receiving system.
the address is not allowed	Not http/https, has a user name, or is a private address while the instance does not allow those.
no answer within 10 seconds	The receiver is too slow. Let it answer first and work afterwards.
could not connect	Wrong host, port or certificate, or the system is down.
too many redirects	More than three; use the final address.

Every run also leaves a short note in the request's history, and a failed *call before finishing* shows the reason on the page where the person clicked Done.

7. Webhook events and n8n

Besides the step trigger, the normal webhooks (**Global settings** → **External services** → **Webhooks**, or per team) carry the workflow:

Event	When
<code>workflow.started</code>	a request was started (data: the master task)
<code>workflow.step_ready</code>	a step can be worked on: the first steps at the start, later ones when the step before is done
<code>workflow.completed</code>	the last step was finished and the master task closed
<code>task.created</code> , <code>task.updated</code> , <code>task.done</code>	for the master task and every step, like any task

Every task event of a workflow task carries the `workflow` block shown above, with the request fields and step fields in `fields` . Private tasks never leave the system.

An n8n flow

- 1 In n8n add a **Webhook** node (method `POST` for the webhook mode, `GET` for *call before finishing*), copy its production URL and paste it as the step's trigger address. Add `?size=<size>` if you want a value in the query.
- 2 Add a **Code** node that verifies `X-RC-Signature` with the secret (snippet above; `docs/webhooks/n8n-task-done.json` shows the pattern for `task.done`).
- 3 Do the work (create the account, order the device, open a ticket in another system).

- 4 Optionally report back: put the result in the step's fields with `PUT /tasks/{id}/workflow-fields` and finish the step with `PUT /tasks/{id} ( {"done": 1} )` - section 8. In webhook mode the step is already done, so use this for the *next* step, which a `workflow.step_ready` subscription tells you about.

8. REST API and MCP

Call	What it does	MCP tool
<code>GET /workflows</code>	the offers the caller may start, with request fields, steps and step fields (never a trigger address)	<code>list_workflows</code>
<code>GET /workflows/{id}</code>	one offer	<code>get_workflow</code>
<code>POST /workflows/{id}/start</code>	start a request	<code>start_workflow</code>
<code>GET /tasks/{id}/workflow</code>	the request, step and fields of a task	<code>get_task_workflow</code>
<code>PUT /tasks/{id}/workflow-fields</code>	fill a step's fields	<code>set_task_workflow_fields</code>
<code>PUT /tasks/{id}</code> with <code>done: 1</code>	finish a step: checks the step fields, runs a <i>call before finishing</i> trigger, wakes the next steps, closes the request after the last	<code>update_task</code>

```
curl -u USER:TOKEN -X POST https://YOUR-INSTANCE/api/v1/workflows/4/start \
-H 'Content-Type: application/json' \
-d '{"title": "Laptop for Max", "solution_id": 3, "fields": {"size": "M"}}'
# 201 {"request_id": 13538, "step_ids": [13539, 13540, 13541], "url": "..."}

curl -u USER:TOKEN -X PUT https://YOUR-INSTANCE/api/v1/tasks/13540/workflow-fields \
-H 'Content-Type: application/json' -d '{"fields": {"ticket": "INC-7"}}'

curl -u USER:TOKEN -X PUT https://YOUR-INSTANCE/api/v1/tasks/13540 \
-H 'Content-Type: application/json' -d '{"done": 1}'
```

The token needs the **read and write** scope for everything but the two **GET** calls. Finishing a step is refused with **400** and the reason while a step field is empty or a *call before finishing* trigger fails.

Rules and limits

- Who may start an offer is the designer's role list; no list or "everybody" means everybody. RC Admins always may.
- The trigger dashboard and its buttons are for RC Admins and the Workflows role. Only an RC Admin can create a new signing secret.
- A step's trigger address is never shown by the API or in logs beyond scheme, host and path; the query string holds filled-in values.
- Trigger log lines are kept for 180 days, finished deliveries for 30 days.
- Webhook triggers are delivered by the scheduler job that also delivers the other webhooks (every minute); without a running scheduler nothing is sent.
- A *call before finishing* trigger blocks the person's click for up to 10 seconds. If the receiver is slow or unreliable, use the webhook mode.
- An agile project's tasks and service request tasks are ordinary tasks: private tasks, permissions and the audit log apply as everywhere.