Web Jobs (JavaScript)

A **Web Job** is a short JavaScript program that WebHub runs on request and answers over HTTP — think of it as a stored procedure written in JavaScript, or a small web page. A job can query the IQX database, call the public internet, use a short list of approved npm modules, and return JSON, a file (CSV, PDF…) or an HTML page. Jobs are called from browsers, from iqxWEB, from third-party webhooks, or on a cron schedule. Each run executes in its own worker thread with a runtime, memory and CPU limit, so a badly written job fails on its own without taking the node down.

Jobs are written and managed from the **Web Jobs** pages in iqxWEB (Owner → Web Jobs → *Manage* and *Scheduler*) and served from `/api/webjobs/<name>`.

This page is about the JavaScript jobs served from `/api/webjobs`. The XML **JobRunner** jobs (`<Job>` files) are a different engine with their own page: they run inside IQX with a fixed vocabulary of commands, whereas a Web Job is ordinary JavaScript answering an HTTP request. They share no code, and one does not run the other.

Writing a job

A job is the **body of an async function**. Use `await` and `return` directly; there is no wrapper to remember.

```
/*
Name: Vacancies by department
Description: Live vacancies for one department, newest first
Runtime: 10000
*/

const department = String(params.department || "").trim()
if (!department) return response.json({ error: "department is required" },
400)

const rows = await db.exec(
  `SELECT TOP 50 VacancyID, Position, Town, WhenCreated
     FROM Vacancy WHERE DepartmentID = ? AND Status = 'L'
    ORDER BY WhenCreated DESC`,
  [department]
)

return { success: true, department, count: rows.length, vacancies: rows }
```

The rules that matter:

- **The return value is the result.** A plain value or object is sent as JSON; a `response.*` value is sent as that response. `finish ( result )` still works (jobs written before October 2026 use it) and whichever of the two happens first wins.
- **An uncaught error fails the run immediately** with `JOB-001` and the message — including a

rejected promise nobody caught and a throw inside a timer. You no longer need a `try/catch` just to make failures prompt; catch only where you want to turn an error into a particular answer.

- **`success: false` in a JSON result marks the run as failed.** The caller gets a `JOB-001` error *instead of* your payload, the run is counted as a failure, and a scheduled run is recorded as failed. Use it to mean "this did not work". For "no rows matched" or "not found", return normally, or `response.json ( body, 404 )`.
- **The header comment is the job's contract.** `Key: value` lines become metadata (`Name`, `Description`, `Author`, `Version`, `Last Updated`), limits (`Runtime`, `Memory`, `CPU`), the access level (`Access`) and the kind of job (`Mode: page`). Don't put `*/` inside it.

The header, in full

```
/*
Name: Staff PDF report              shown in the jobs list
Description: One line on what it does and what it expects
Author: IQX Limited
Version: 1.2.0
Last Updated: 2026-10-05
Runtime: 60000                      ms of wall clock (default 10000)
Memory: 512                         MB of heap (default 256)
CPU: 20000                          ms of CPU time (default 5000)
Access: owner                       public | user (default) | owner | key
Mode: page                          when the job renders a browser page
*/
```

What a job can see

Only this. There is no `process`, no `fs`, no `fetch`, no `import`, and `require` knows only the approved list.

`params`	The request parameters. GET → the query string (**every value is a string**, so `Number ( params.limit )` yourself); POST → the parsed JSON or form body. A checkbox group arrives as a string for one tick and an array for several. On a scheduled run it is the schedule's stored JSON. The engine's own flags (`enableLogger`, `raw`, `_console`, `_debug`) are stripped before the job sees them.
`request`	`{ method, url, path, query, headers, ip, caller }`. `method` tells a page job a landing GET from a submitted POST; a scheduled run sees `"SCHEDULE"`. `caller` is who is calling: `{ kind, id, name, userClass }` where `kind` is `anonymous`, `user`, `system`, `schedule` or `apikey`. The `Authorization` header is withheld.
`userID`, `isAuthenticated`	The caller's `IQXNetUserID` and whether there is one — a signed-in user opening a public page is recognised. Scheduled runs get the system user.

`db.exec ( sql, params? )`	Runs the SQL as the pears (business-table) user with ? placeholders bound from the array, and resolves to an **array of row objects** for a SELECT (`{ affectedRows }` for insert/update/delete). The connection carries the caller's `userstaffid`, so `ExternalCredentialGet ( )` and division-aware procedures work. A failing statement is logged with the job's prefix and rethrown with its SQL code. `getDB ( )` returns the same object.
`http ( url, options? )`	Outbound HTTP. `options`: `method`, `headers`, `body` (a string, bytes, or an object which is sent as JSON), `username`/`password` (basic auth), `timeoutMs`, `followRedirects`. Returns `{ status, ok, statusText, headers, text, bytes, json ( ) }`. Only `http`/`https` to the **public internet**: loopback, private ranges, link-local, the hub's own database server and anything on `WEBJOB_HTTP_DENY` are refused, every redirect hop is re-checked, and a site can narrow it to named hosts with `WEBJOB_HTTP_ALLOW`. 10 s timeout and 10 MB response by default, at most 50 calls per run.
`require ( name )`	Loads an **approved** module synchronously: `const { format } = require ( "date-fns" )`. Approved today: `date-fns`, `@iqx-limited/quick-pdf`. Anything else throws (`JOB-002`). `await require ( … )` from older jobs still works.
`console`	`log`/`info`/`debug`/`warn`/`error`. Lines go to the server log prefixed `[WebJob:<name>]`, are kept with the run in history (see *Run history*), stream live to the run dialog, and can be shown under a page with `_console=1`.
`response`	Builders for non-JSON answers — see *Responses*. `rawResponse ( body, headers )` is the older form of `response.raw`.
`sign ( payload, ttlSeconds? )`, `verify ( token )`	A signed, expiring token (HMAC, keyed from the server secret). `verify` returns the payload or `null`. This is how a multi-step page proves that an earlier step passed — see *Serving pages*. Default TTL 15 minutes.
`breakpoint ( )`	Pauses the job when it is being run under the debugger; a no-op otherwise (so is a plain `debugger` statement). See *Debugging*.
`limits`	The resolved `{ runtimeMs, memoryMb, cpuMs }` for this run.
`Buffer`, `URL`, `URLSearchParams`, `TextEncoder`/`TextDecoder`, `atob`/`btoa`, `structuredClone`, `AbortController`, timers	Passed through.
`extendLimits ( )`	Deprecated — declare limits in the header instead. Still extends the runtime for old jobs and logs a warning.

Who may call a job

WebHub works out **who is calling** before anything runs: a signed-in user (from the bearer token or the iqxWEB session cookie), the hub's system token, the scheduler, or nobody. It then checks that against the job's **access level**, the `Access:` header line:

`public`	Anyone, with no credentials. Rate limited per caller (`WEBJOB_PUBLIC_RATE`, 60 a minute, with a `Retry-After`). A job named `public_…` is `public` when the header is absent.
`user`	Any signed-in user. **The default** for every other job.
`owner`	OWNER users only.
`key`	Reserved for an API key pinned to this job (keys are planned, not yet issued). Owners can run it; everyone else is refused.

The system token and the scheduler always pass. A refused call gets `AUTH-SESSION-001` (401, nobody signed in) or `SYS-403` (signed in, but not enough). Inside the job, `request.caller.kind` says which of these happened, so one job can serve a visitor and a consultant differently, and history records the caller for every run.

For an external system that must call a non-public job today, the honest options are the system token, or a `public` job that checks a signed token or shared secret in its parameters with `verify ( )`.

Limits

Jobs are meant to be snappy. The defaults are 10 seconds of wall clock, 256 MB of heap and 5 seconds of CPU time. A job that needs more declares it in its header, which is deliberate: it is an acknowledgement that this job is slow, and it is visible in the jobs list (an *extended* badge).

Each value is capped by a server maximum (`WEBJOB_MAX_RUNTIME_MS` 300 000, `WEBJOB_MAX_MEMORY_MB` 2048, `WEBJOB_MAX_CPU_MS` 120 000); a higher declaration is clamped and logged. The runtime limit terminates the worker. The memory limit is enforced by V8 on the worker's own heap. The CPU budget is measured per thread from the main thread while the job runs, so a synchronous busy-loop is caught by it rather than by the runtime limit. All three end the run with `JOB-003` and a message naming which limit was hit.

Two more guards are per node rather than per job. At most `WEBJOB_MAX_CONCURRENT` (8) workers run at once; further requests queue for up to 15 seconds and are then refused with `SYS-429`. And callers without credentials are rate limited as above.

One deployment caveat: a `--max-old-space-size` in the WebHub process's `NODE_OPTIONS` overrides per-worker heap limits. The server logs a clear warning on the first run if that is the case; remove the flag to get memory limits back.

How a run works

1. A request arrives at `GET` or `POST /api/webjobs/<name>`. The name must match `[a-zA-Z0-9_-]+` or it is a `HUB-007`.
2. WebHub resolves the caller and checks the job's access level (previous section).
3. It reads the job's source from the job store (no such job → `SYS-404`), parses the header, resolves the limits, and starts a **worker thread** with an empty environment, its own heap, and a context holding exactly the globals above. Everything the job does to the outside world — `db`, `http`, `console` — is a message to the host, which applies the policy and does the real work.
4. The job runs; the host kills it on the runtime or CPU limit, V8 on the memory limit.
5. The run — parameters, result, console output, outcome, caller — is recorded in history, and the

answer is sent back (next section).

The worker is a containment boundary for mistakes, not a wall against a hostile author — someone who can write a job already has full SQL against the database, and `vm` contexts are escapable by design. That is why writing and uploading jobs is limited to **OWNER** users, and why a `public` job should treat every parameter as untrusted input.

Responses

`return` a plain value for JSON. For anything else use `response`:

```
return response.json({ status: "not_found" }, 404)          // JSON with a
real status
return response.html(page)                                   // text/html;
charset=utf-8
return response.text("ok")
return response.csv(text, "staff.csv")                       // download
return response.pdf(bytes, "report.pdf")                     // bytes or a
base64 string
return response.file(bytes, "x.png", "image/png")
return response.redirect("/api/webjobs/public_next?token=" + token)   // 302
+ Location
return response.raw(body, { "Content-Type": "…" }, 200)
```

How that reaches the caller depends on who is asking:

- **A browser** (a navigation, or `Accept: text/html`) gets an ordinary HTTP response: for a `response.*` result the status, the headers and the body; for a plain JSON result just the JSON, pretty-printed as `application/json`, so a job URL pasted into a tab shows its output rather than downloading a file. HTML pages get a job-page `Content-Security-Policy` that allows inline scripts and `https:` assets unless the job sets its own. A failure comes back as a JSON error object with the matching status.
- **An API client and the iqxWEB run dialog** (`Accept: */*`, or `enableLogger=1`) get the **NDJSON** form: one metadata line, then the data.

```
{"success":true,"jobId":"job_1759…","isMetaData":true,"type":"json","format"
:"application/json","status":200,"contentLength":123,"runtime":84}
{"contacts":[…],"count":3}
```

For a raw response the metadata line is followed by the bytes. With `enableLogger=1` each `console.*` call arrives first as a `{"type":"log",…}` line, which is how the run dialog shows a live console. A failure is a single line `{"success":false,"error":"…","code":"JOB-001","ref":"…","runtime":…}` with the matching HTTP status. A script that wants the plain response rather than the stream adds `?raw=1`.

```
curl "https://hub.example.com/api/webjobs/public_ping?who=gareth"
curl -X POST "https://hub.example.com/api/webjobs/3cx" -H "Authorization:
Bearer $TOKEN" \
```

```
     -H "Content-Type: application/json" -d '{"limit": 50, "enableLogger":
1}'
curl -o report.pdf "https://hub.example.com/api/webjobs/pdf-report?raw=1" -H
"Authorization: Bearer $TOKEN"
```

Serving pages

A job can be a small web application: GET renders a page, the page POSTs back to the same job (or another), and the job decides what to render next. There is no server-side session, so **state travels in the request**, and anything that must not be forged — "this person passed the date-of-birth check" — travels as a token from `sign ( )`. The pattern, from `public_deeplink`:

```
if (request.method !== "POST") {                  // landing: ask for the DOB
  return page(T("dob", { uid, error: "" }))
}
if (params.step === "dob") {                      // check it, mint a token,
show the shifts
  if (params.dob !== record.dateofbirth) return page(T("dob", { uid, error:
"No match" }))
  const token = sign({ uid, step: "shifts" }, 15 * 60)
  return page(T("shifts", { uid, token, shiftrows }))
}
if (params.step === "shifts") {                   // only a valid token gets
here
  const claim = verify(params.token)
  if (!claim || claim.uid !== uid) return page(T("dob", { uid, error:
"Session expired" }))
  // … record the selection …
  return page(T("done", { count }))
}
```

To chain into a *different* job, redirect to it with the token in the URL (`response.redirect`) and `verify` on arrival; a browser can only be redirected with a GET, so the token is the only trustworthy thing it can carry.

Templates live in the database, not in files: a `WebJobTemplate ( Name, Html )` table read with `db.exec` and a few lines of `{{placeholder}}` replacement (the deep-link job shows one with escaped and raw placeholders). What you owe a public page, because the engine cannot do it for you: HTML-escape every value that came from the database or the request; check posted IDs against what you offered; keep tokens short-lived; and remember the rate limit is per caller, not per candidate.

Declare a page job with `Mode: page` in its header. The engine treats it like any other job, but the Owner page does the right thing with it: *Open* opens the page in a browser tab instead of the run dialog, the list shows a *page* badge, a *Pages* filter finds them, and *Debug* attaches to the real page (see *Debugging*). A job page is a real page with its own forms and scripts, so it always opens in its own tab — it is not embedded inside iqxWEB.

To see what a page job logged while using it, add `_console=1` to its URL: the HTML comes back with

a collapsible drawer underneath holding the job's console lines with timings, the run time, the method and parameters it saw, and the request `ref`. The drawer is only added for an **OWNER** session (or on a dev server with `WEBJOB_PAGE_CONSOLE=1`); a visitor adding the flag gets the page unchanged.

The Owner pages

Manage lists every job on one line — name, badges (`public`, `page`, `extended`, and green *scheduled* / grey *schedule off*), description, last run with its result and a live *running · scheduled · 12 s* indicator when something is in flight — with filters (Public, Pages, Scheduled, Extended limits, Last run failed) and a search box. Expand a row for its limits, access, source and schedules. The **Run** button runs the job in a dialog that shows the console as it happens and then the result (JSON, a text or HTML preview, or a download); its menu offers *Run with parameters...* (remembered per job), *Open in browser*, *Debug...*, *Edit code*, *History* and *Delete*. **Create**, **Upload** (a `.job.js` file) and **Ad-hoc** (run a file once without saving it) are in the toolbar; the editor has *Run* and *Debug* buttons that act on the unsaved code.

History (per job, or per schedule from the Scheduler page) lists the recent runs with status, duration, who or what started them and the node that ran them; click one to replay its console output with timings and see the result it returned, or why it failed.

Scheduler lists schedules the same way: a switch to enable or disable each one, what it runs and when in plain words, the last run (click it for that schedule's history) and the real next fire time, *Run now* with the schedule's parameters, and a menu for history, editing, running with other parameters and deleting. *Add schedule* saves a new schedule switched **off** unless *Enabled* is ticked.

Debugging

A job can be stepped through line by line from the Owner page: *Debug* in a job's Run menu, or the *Debug* button in the editor (which runs the unsaved code). The dialog shows the source with the paused line highlighted, every variable in scope at that point with its value (expand objects and arrays), the console so far, and **Step**, **Continue** and **Stop**.

Put a breakpoint where you want to stop:

```
const rows = await db.exec("SELECT TOP 5 * FROM Person")
breakpoint()            // or a plain `debugger` statement
const names = rows.map(r => r.Name)
```

Without any breakpoint the run pauses before the first line. Stepping pauses before each statement in the job body and inside `async` functions (including `if`/`for`/`try` blocks within them); code inside synchronous callbacks such as `rows.map ( r => … )` runs without pausing, because the pause is an `await`. A not-yet-initialised `let` shows as `<tdz>`, long strings and deep objects are truncated, and *Stop* ends the run with a `JOB-001` "Stopped by the debugger at line N". The runtime limit is held while the job is parked, so sitting on a breakpoint does not time the job out; a session left alone for 10 minutes is cleaned up.

For a **page job**, *Debug* attaches to the real flow: the dialog opens *live*, waiting; **Open page** opens the job in a new tab with the debugger attached, and every request that tab makes — the landing

GET, each form POST back, a redirect — runs under the debugger and pauses in the dialog while the browser waits for its page. The console shows each request arriving and being answered. The tab is tagged with the session (first in the URL as `_debug=`, then in a cookie scoped to `/api/webjobs` so the page's own posts keep it); other visitors are untouched. The switch at the bottom chooses between pausing on the first line of every request and only at `breakpoint ( )` / `debugger`. Closing the dialog detaches, stops any paused request, and the tab's next request runs normally.

`breakpoint ( )` and `debugger` are **no-ops in a normal run** — a normal run is never instrumented and `breakpoint` is an empty function in the sandbox — so one left in a job does nothing in production; only an OWNER starting a debug session makes them pause.

Scheduled jobs

Schedules are rows of the `IQXNetWebJobSchedule` table: an ID, a cron expression, a display name, the job's name, a JSON string of parameters and an enabled flag. The Scheduler page manages them through the `NetV2…JobSchedule` procedures as the signed-in user; WebHub itself reads and updates the table directly, because those procedures are gated on the caller's user class and the hub runs as the SYSTEM user. WebHub loads the enabled rows at start-up and on POST `/api/webjobs/cron` (which reports how many it loaded); a disabled or deleted row stops its cron job.

Each firing runs the job as the system user with the stored parameters (invalid JSON is logged and the job runs with `{}`), sees `request.method === "SCHEDULE"` and `request.caller.kind === "schedule"`, is recorded in the job's history with the schedule's name, and writes `LastRun`, `NextRun` and `LastStatus` back to the row. A scheduled job has no caller to read its output, so it should report through its own effects and use `success: false` for anything that should show as a failed run — the console output and result are kept in history either way, which is where to look when a scheduled job starts going wrong.

On a multi-node site every node loads every schedule today, so a schedule fires once per node.

Run history

Every run — manual, scheduled, from the editor or under the debugger — is recorded with its parameters, its result (text results up to 64 KB), what it wrote to the console (up to 300 lines / 32 KB, with timings), how it ended, who or what started it, and the node it ran on. The History dialog replays this, and the per-job stats on the Manage page and the metrics cards are computed from it.

History lives in **MongoDB** by default (`WEBJOB_HISTORY_STORE=mongo`): the `webjob_runs` collection in the hub's own `IQXWebHub` database on the Mongo server WebHub already uses. It is not the recruitment `IQXWeb` database, and nothing goes to the IQX SQL database. Every node writes there and reads the same history; runs expire after `WEBJOB_HISTORY_RETENTION_DAYS` (90) via a TTL index. `WEBJOB_HISTORY_STORE=local` keeps a per-node `node-persist` file instead (newest `WEBJOB_HISTORY_MAX` runs).

Where the jobs live

`WEBJOB_STORE=file` (the default) keeps jobs as `jobs/<name>.job.js` on each node's disk, so on a multi-node or multi-container site a job saved through one node has to reach the others (shared volume, or save on both). A plain `.js` in the folder is not a job and is reported once in the log at start-up.

`WEBJOB_STORE=db` keeps them in the `IQXNetWebJob` table instead (`Name`, `Code`, `UpdatedAt`, `UpdatedBy`), created on first use and read per run, so every node sees the same jobs the moment one is saved and nothing needs to be on disk. To move an existing node over: set the variable, restart, then `POST /api/webjobs/store/import-files` once (OWNER) to copy its folder into the table (`{"overwrite": true}` to replace rows that already exist). `GET /api/webjobs/store` says which store a node is using, and the Manage page shows it in the list header.

Managing jobs over HTTP

All under `/api/webjobs`, OWNER session unless noted. The Owner pages are built on them.

`GET /available`	Every job with metadata, resolved limits, `access`, `mode`, run statistics, where it is stored and when it changed.
`GET /<name>/code`	The source of one job.
`POST /save { name, code }`	Write (or overwrite) a job.
`POST /upload` (multipart `file`, `name`)	Same, from an uploaded `.job.js`.
`DELETE /<name>`	Remove a job.
`POST /execute` (multipart `file`, `params`, `enableLogger`)	Run an uploaded file once without saving it — the editor's Run and the Ad-hoc button. Always streams.
`GET /running`, `POST /<jobId>/stop`	Runs in flight on this node (`jobName`, `source`, `runtime`); the Owner pages poll it. *Stop* removes the entry; the worker is killed by its own limits.
`GET /history?limit&offset&search`, `GET /<jobId>/output`, `GET /<jobId>/logs`	Run history. `/output` returns a run's parameters, result, console lines and outcome.
`GET /metrics`	Totals, success rate, runtimes, failures today, `workersRunning` / `workersQueued` / `workersMax`, and which stores this node uses.
`POST /debug/<name> { params, code?, stepFromStart?, method? }`	Run one simulated request under the debugger (streams `debug` / `pause` lines).
`POST /debug/<name>/attach { stepFromStart? }`	Attach the debugger to real requests tagged with the session it returns.
`POST /debug-session/<id>/command { command }`	`step`, `continue`, `stop`, `config` (`stepFromStart`) or `detach`.
`GET /store`, `POST /store/import-files`	Which job store this node uses; copy its jobs folder into the database store.
`GET /schedules/status`	The schedules this node has loaded, with each one's real next fire time.

`POST /cron`	Re-read the schedule table; reports how many schedules loaded.

Error codes

`JOB-001`	The job failed: it threw, returned `success: false`, a query inside it failed, it has a syntax error, or it was stopped from the debugger. `detail` has the message.
`JOB-002`	`require ( )` of a module that is not approved or not installed.
`JOB-003`	A limit was hit: runtime, memory or CPU. The message says which.
`JOB-004`	The job finished without a result (`finish ( )` with nothing, or nothing returned and `finish` never called).
`AUTH-SESSION-001`, `SYS-403`	The caller may not run this job (nobody signed in / not enough) — see *Who may call a job*.
`SYS-429`	Too many: the rate limit for callers without credentials, or the node's worker queue is full.

Full table on **Web error codes**; every failure carries a `ref` to search the server logs with.

Trying it

The repo ships a small test pack; run each and compare with its header. `_test-limits` returns the resolved limits, and `?what=runtime`, `?what=cpu` or `?what=memory` shows each limit ending a run with `JOB-003`. `_test-failures` shows how runs fail and how they don't: `?mode=throw` (default), `async` and `timer` all end at once with `JOB-001`, `?mode=false` is a `success: false` failure, `?mode=404` is a 404 that is *not* a failure, `?mode=ok` succeeds and echoes the caller. `_test-sandbox` lists what the sandbox exposes — every probe should read `undefined` or refused. `_test-http` exercises the outbound network policy. `_test-legacy` proves the pre-October-2026 IIFE shape still runs. `public_test-page` is a page job to open in a browser (inline script, a form that posts back, a console link), and `public_ping` is the smallest possible public job. The engine's unit tests are `npm run test:unit` (`src/routes/webJobEngine.test.ts`).

Jobs written before October 2026

The earlier engine ran the file as a synchronous function body, so every job wrapped itself in `(async function (params, userID, finish) { … })(params, userID, finish)` and used `await require ( )`, `getDB ( )` and `finish ( )`. All of that still works unchanged inside the new wrapper; the only behavioural differences are that an uncaught error now fails at once instead of waiting for the timeout, that `extendLimits ( )` logs a deprecation pointing at the header, and that files must be named `<name>.job.js`.

For developers

- `src/routes/webJobWorker.ts` — the JavaScript that runs inside the worker (kept as a string, started with `eval: true`): the sandbox, the async wrapper, the bridge stubs, `response`, `sign`/`verify`, `breakpoint`.
- `src/routes/webJobEngine.ts` — the host: starts the worker (empty env,

`resourceLimits`), answers bridge calls, enforces limits, the concurrency gate, `resolveJobLimits`, the debugger pause protocol. No database or Fastify imports, so it is unit-testable (`webJobEngine.test.ts`).

- `src/routes/webJobHttp.ts` — the network policy and `http ( )` implementation.
- `src/routes/webJobDebug.ts` — the debugger's source instrumentation (acorn) and value serialiser.
- `src/routes/webJobAuth.ts` — `JobCaller` and `resolveJobCaller` (the one place identity is read), the `Access:` policy, the rate limit. An API-key scheme is one new branch here.
- `src/routes/webJobStore.ts` — where job source lives: the `jobs/` folder or the `IQXNetWebJob` table.
- `src/routes/webJobHistory.ts` — run history and metrics: MongoDB (`IQXWebHub.webjob_runs`) or `node-persist`.
- `src/routes/webJobs.ts` — `executeWebJob` wiring (store, metadata, `query` as pears with `userstaffid`, logging), `toJobError`, approved modules.
- `src/routes/jobsManagement.ts` — the routes: raw mode vs NDJSON, the `_console` drawer, the debug sessions, the cron scheduler.
- Settings are `WEBJOB_*` in `.env.sample`. A new failure kind needs a `JOB-00n` entry in `@iqx-limited/web-errors` and a row on **Web error codes** — memory and CPU currently share `JOB-003`.
- Not done yet, by design: API keys (`Access: key`); scheduler on the cluster leader only; DNS-rebinding pinning in `http ( )`.

From:
https://iqxusers.co.uk/iqxhelp/ - **IQX**

Permanent link:
https://iqxusers.co.uk/iqxhelp/doku.php?id=iqxwebnext:technical:web_jobs&rev=1791224611

Last update: **2026/10/05 18:23**