Web Jobs (JavaScript)

A **Web Job** is a `.job.js` file that WebHub runs on request and answers over HTTP — a stored procedure written in JavaScript, or a small web page. Jobs 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. They 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.

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.

Writing a job

A job file 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 line *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`) and limits (`Runtime`, `Memory`, CPU — below). Don't put `*/` inside it.

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. `enableLogger` and `raw` are stripped before the job sees them.
`userID`, `isAuthenticated`	The caller's `IQXNetUserID` for a session call, `null` / `false` for an unauthenticated `public_` call. Scheduled runs get the system user.
`request`	`{ method, url, path, query, headers, ip }` — tells a page job a landing GET from a submitted POST, and lets it read a cookie. The `Authorization` header is withheld.
`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. 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`, prefixed `[WebJob:<name>]` in the server log (debug level; `warn`/`error` as `warn`). With `enableLogger=1` the lines are also streamed to the caller.
`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.
`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.

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.

```
/*
Name: Staff PDF report
Runtime: 60000     ms of wall clock
Memory: 512        MB of heap
CPU: 20000         ms of CPU time
*/
```

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 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 `public_` jobs are rate limited per client IP (`WEBJOB_PUBLIC_RATE`, 60 per minute, with a `Retry-After`).

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. A name starting `public_` runs without authentication (rate limited). Any other name needs a valid session bearer, checked before the file is read.
3. WebHub reads `jobs/<name>.js` (no file → `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 result is recorded in history, metrics and the daily log file, and 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** (`Accept: text/html`, or any caller adding `?raw=1`) gets an ordinary HTTP response: the status, the headers, the body. HTML pages get a job-page `Content-Security-Policy` that allows inline scripts and `https:` assets unless the job sets its own.
- **An API client and the iqxWEB job page** (`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 job page shows a live console. A failure is a single line `{"success":false,"error":"…","code":"JOB-001","ref":"…","runtime":…}` with the matching HTTP status. JSON results are the same on both paths, so existing integrations are unaffected.

```
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 `jobs/public_deeplink.js`:

```
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 IP, not per candidate.

Managing jobs

All under `/api/webjobs`, OWNER session unless noted. The Owner web-jobs page in iqxWEB is built on them.

`GET /available`	Every `.js` in the jobs folder with metadata, resolved limits, whether it is public, run statistics and modified time.
`GET /<name>/code`	The source of one job.
`POST /save { name, code }`	Write (or overwrite) `jobs/<name>.js`.
`POST /upload` (multipart `file`, `name`)	Same, from an uploaded `.js`.
`DELETE /<name>`	Remove the file.
`POST /execute` (multipart `file`, `params`, `enableLogger`)	Run an uploaded file once without saving it — the editor's Run button. Always streams.
`GET /running`, `POST /<jobId>/stop`	Jobs in flight on this node. *Stop* removes the entry; the worker is killed by its own limits.
`GET /history?limit&offset&search`, `GET /<jobId>/logs`, `GET /<jobId>/output`	The last 1,000 runs.
`GET /metrics`	Totals, success rate, runtimes, failures today, and `workersRunning` / `workersQueued` / `workersMax`.
`POST /cron`	Re-read the schedule table.

History and metrics are kept in `node-persist` on each node and appended to `logs/job_YYYY-MM-DD.json`. Both are **per node**: on a two-node site each WebHub has its own jobs folder, history and schedules, so a job saved through one node has to reach the other (shared volume, or save on both).

Scheduled jobs

Schedules are rows read through `NetV2GetWebJobSchedule`: an ID, a cron expression, the display name, the file name (without `.js`), a JSON string of parameters and an enabled flag. WebHub loads them at start-up and on `POST /api/webjobs/cron`; a disabled row stops a running schedule. Each firing runs the job as the system user with the stored parameters (invalid JSON is logged and the job runs with `{}`) and writes the outcome back with `NetV2UpdateJobSchedule`. 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.

Error codes

`JOB-001`	The job failed: it threw, returned `success: false`, a query inside it failed, or it has a syntax error. `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).
`SYS-429`	Too many: the public rate limit, 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 test pack. Run each and compare with its header's "Expect" line: `_test-loop` (CPU

budget kills a busy loop), `_test-memory` (V8 kills a memory hog), `_test-escape` (what the sandbox exposes — everything should read `undefined` or refused), `_test-throw` (fail-fast, with `?mode=async` and `?mode=timer`), `_test-limits` (header-declared runtime), `_test-http` (network policy), `_test-status` (404 without failure), `_test-legacy` (the old IIFE shape still runs), and `public_test-page` (open it in a browser: a real page with a working inline script and a form that posts back; the same URL from curl gives the NDJSON stream). The engine's unit tests are `npx tsx --test 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 difference is that an uncaught error now fails at once instead of waiting for the timeout, and `extendLimits ( )` logs a deprecation pointing at the header.

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`.
- `src/routes/webJobEngine.ts` — the host: starts the worker (empty env, `resourceLimits`), answers bridge calls, enforces limits, the concurrency gate, `resolveJobLimits`. 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/webJobs.ts` — `executeWebJob` wiring (file, metadata, `query` as pears with `userstaffid`, logging), `toJobError`, approved modules.
- `src/routes/jobsManagement.ts` — routes, raw mode vs NDJSON, `request`, history/metrics, the cron scheduler. `src/routes/webJobAuth.ts` — name rules and the public rate limit.
- 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: jobs and history in the database instead of per-node files; scheduler on the cluster leader only; per-job access control beyond public/any-authenticated; DNS-rebinding pinning in `http ( )`.

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

Permanent link:
https://phpddemo.iqxusers.co.uk/iqxhelp/doku.php?id=iqxwebnext:technical:web_jobs&rev=1791200049

Last update: **2026/10/05 11:34**