Web Jobs (JavaScript)

A **Web Job** is a `.js` file that WebHub runs on request and hands back the result over HTTP — a stored procedure written in JavaScript. Jobs can query the IQX database, use a small set of approved npm modules, and return either JSON or a raw file (HTML, CSV, PDF…). They can be called from a browser, from iqxWEB, from a third-party integration, or on a cron schedule.

This page is about the JavaScript jobs served from `/api/webjobs`. The XML **JobRunner** jobs (`<Job>` files, `/api/wpk/job/<name>.xml`) are a different engine with their own page.

How a job runs

1. A request arrives at GET or POST `/api/webjobs/<name>`. The name must match `[a-zA-Z0-9_-]+` (no dots, slashes or spaces) — anything else is a `HUB-007`.
2. Unless the name starts with `public_`, the caller must have a valid session (bearer token). A missing or bad token is refused before the file is even looked at.
3. WebHub reads `jobs/<name>.js` from its working directory (`/app/jobs` in the container, the service folder on Windows). No file → `SYS-404`.
4. The file's text is wrapped in a function and called with a fixed set of globals (below). It is **not** an ES module: there is no `import`, no top-level `await`, and `require` is WebHub's own controlled version.
5. The job does its work and calls `finish ( result )` exactly once. WebHub polls for that call every 100 ms and gives up after the runtime limit (10 s by default) with `JOB-003`.
6. The result is streamed back as **NDJSON**: one metadata line, then the data (see *What the caller receives*). The run is recorded in history, metrics and the daily log file.

The function wrapper matters in practice. Because the body is plain script, the idiomatic shape of every job is an async IIFE:

```
/*
Author: IQX Limited
Name: My Job
Description: One line saying what it does
Version: 1.0.0
Last Updated: 2026-10-03
*/

(async function (params, userID, finish) {
  try {
    // ... work ...
    finish({ success: true, data })
  } catch (error) {
    console.error("my-job failed:", error.message)
    finish({ success: false, error: error.message })
  }
})(params, userID, finish)
```

The leading comment block is optional but parsed: `Key: value` lines become the job's metadata (`author`, `name`, `description`, `version`, `lastupdated`) shown in the jobs list.

What a job can see

`params`	The request parameters. GET → the query string (**every value is a string**, so `Number ( params.limit )` yourself); POST → the parsed JSON body. On a scheduled run it is the schedule's stored JSON. `enableLogger` is stripped before the job sees it.
`userID`	The caller's `IQXNetUserID` for a session call; `null` for an unauthenticated `public_` call. Scheduled runs get the system user ID.
`isAuthenticated`	`true` when `userID` is set.
`getDB ( )`	Returns `{ exec ( sql, params? ) }`. `exec` 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. Statements that return no rows resolve to an affected-rows result instead. A failing statement is logged with the job's prefix and rethrown, so wrap it in `try/catch`.
`finish ( result )`	Ends the job. A plain value/object is sent as JSON; a `rawResponse ( … )` is sent as-is. Only the first call counts — later calls log a warning and are ignored.
`rawResponse ( body, headers )`	Marks `body` (string, `Buffer` or `Uint8Array`) as a raw HTTP response with the given headers. Set `Content-Type`; anything other than `application/json` is sent as bytes.
`require ( name )`	Loads an **approved** module. It is really a dynamic `import ( )`, so it returns a *promise* of the module namespace: `const { format } = await require ( "date-fns" )`. A module with a default export is at `.default`. Anything not on the list throws (`JOB-002`).
`console`	`log`/`info`/`debug`/`warn`/`error`, prefixed `[WebJob:<name>]` in the server log (`debug` level; `warn`/`error` as `warn`, so visible in production). With `enableLogger=1` the lines are also streamed to the caller.
`extendLimits ( cpu, memory, runtime )`	Logs the request. **At present it does not change the 10 s timeout** — the limit is read before the job's code runs — so treat it as documentation of intent until the engine is changed.
`setTimeout` / `clearTimeout` / `setInterval` / `clearInterval`, `Date`, `Math`, `JSON`, `parseInt`, `parseFloat`, `isNaN`, `isFinite`	Passed through from Node.

Approved modules right now: `date-fns` and `@iqx-limited/quick-pdf` (HTML → PDF, `const { html2pdf } = await require ( "@iqx-limited/quick-pdf" )`). The list is `APPROVED_MODULES` in `src/routes/webJobs.ts`; a module must also be a dependency in WebHub's `package.json`.

The wrapper is a convenience, not a security boundary — the code runs in the WebHub process with its full privileges. That is why writing, uploading and editing jobs is limited to **OWNER** users, and why a `public_` job should be treated like any other unauthenticated endpoint (validate every parameter,

bind every value, return the minimum).

Success, failure and

The HTTP status does not depend on what the job returns — with one exception. If the JSON result contains `success: false`, WebHub treats the run as **failed**: it is counted as a failure, recorded with the `error` text, and the caller gets a `JOB-001` error line *instead of* the job's own payload. Use that deliberately for "this did not work"; for "no rows matched", return `success: true` with an empty array.

A synchronous throw at the top of the file fails the run at once (`JOB-001`), but an error thrown *inside* the async IIFE with no catch is just an unhandled rejection: nothing calls `finish`, so the job sits there until the 10 s limit and fails with `JOB-003`. Always catch and `finish ( { success: false, error } )` so the failure is immediate and explained.

`JOB-001`	The job failed: it returned `success: false`, threw, or a query inside it failed. `detail` has the job's text.
`JOB-002`	`require ( )` of a module that is not approved or not installed.
`JOB-003`	Timed out (10 s). Usually a missing `finish` on some path, or an uncaught error.
`JOB-004`	`finish ( )` was called with nothing.

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

What the caller receives

The response is `application/x-ndjson`-style: the first line is metadata, then the data.

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

For a `rawResponse` whose `Content-Type` is not JSON, the metadata line is followed by the raw bytes (HTML text, a PDF…). With `enableLogger=1` (query string on GET, body field on POST) each `console.*` call arrives first as `{"type":"log","level":"log","message":[…]}` lines, which is how the iqxWEB job page shows a live console. On failure the single line is `{"success":false,"error":"…","code":"JOB-001","ref":"…","runtime":…}` with the matching HTTP status.

```
# public job, GET parameters
curl "https://hub.example.com/api/webjobs/public_ping?who=gareth"

# authenticated job, POST parameters, streamed console
curl -X POST "https://hub.example.com/api/webjobs/3cx" \
  -H "Authorization: Bearer $TOKEN" -H "Content-Type: application/json" \
  -d '{"limit": 50, "enableLogger": 1}'
```

Managing jobs

All of these live under `/api/webjobs` and (except where noted) need an **OWNER** session. The Owner web-jobs page in iqxWEB is built on them.

`GET /available`	Every `.js` in the jobs folder with its parsed metadata, 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` file (`application/javascript` or `text/javascript`).
`DELETE /<name>`	Remove the file.
`POST /execute` (multipart: `file`, `params` JSON, `enableLogger` `"1"`)	Run an uploaded file **once without saving it** — the scratch-pad behind the editor's Run button. It runs as the caller, named `adhoc_<file>` in history.
`GET /running`, `POST /<jobId>/stop`	Jobs in flight on this node. *Stop* only removes the entry from the list; it cannot interrupt running JavaScript.
`GET /history?limit&offset&search`, `GET /<jobId>/logs`, `GET /<jobId>/output`	The last 1,000 runs with status, runtime, input and output.
`GET /metrics`	Totals, success rate, average/fastest/slowest runtime, failures today.
`POST /cron`	Re-read the schedule table (below).

History and metrics are kept in `node-persist` on each node, and every run is also 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 in the database 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 again on `POST /api/webjobs/cron`; a disabled row stops any running schedule. Each firing runs the job as the system user with the stored parameters (invalid JSON is logged and the job runs with `{}`), then writes the outcome back with `NetV2UpdateJobSchedule` (`pLastStatus` success/failed, `pLastRun`, `pNextRun`).

A scheduled job has no caller to read its output, so make it report through its own effects — write a row, send an email — and use `success: false` for anything that should show as a failed run.

Examples

1. Minimal public job

Anyone can call `GET /api/webjobs/public_ping?who=Gareth`. No database, no auth.

```
/*
Author: IQX Limited
Name: Ping
Description: Smoke test that the Web Jobs engine is up
Version: 1.0.0
*/

(async function (params, userID, finish) {
  finish({
    success: true,
    message: `pong, ${params.who || "stranger"}`,
    at: new Date().toISOString(),
    authenticated: isAuthenticated
  })
})(params, userID, finish)
```

2. Parameterised query with validation

`POST /api/webjobs/vacancies-by-dept` with `{ "department": "NURS", "limit": 25 }`. Note the `?` binds — never splice a parameter into the SQL string — and that `TOP ?` is bound like any other value.

```
/*
Author: IQX Limited
Name: Vacancies by department
Description: Live vacancies for one department, newest first
Version: 1.0.0
*/

(async function (params, userID, finish) {
  const department = String(params.department || "").trim()
  const limit = Number(params.limit) || 50

  if (!department) {
    return finish({ success: false, error: "department is required" })
  }
  if (!Number.isInteger(limit) || limit < 1 || limit > 500) {
    return finish({ success: false, error: "limit must be 1-500" })
  }

  const db = getDB()
  try {
    const rows = await db.exec(
      `SELECT TOP ? VacancyID, Position, Town, Salary, WhenCreated
         FROM Vacancy
        WHERE DepartmentID = ? AND Status = 'L'
        ORDER BY WhenCreated DESC`,
```

```
      [limit, department]
    )
    finish({ success: true, department, count: rows.length, vacancies: rows
})
  } catch (error) {
    console.error("vacancies-by-dept:", error.message)
    finish({ success: false, error: "Query failed", details: error.message
})
  }
})(params, userID, finish)
```

3. Returning a file — CSV download

`rawResponse` turns the result into a download. The same pattern with `Content-Type: text/html` serves a report page, and with a `Buffer` from `html2pdf` serves a PDF.

```
/*
Author: IQX Limited
Name: Staff CSV
Description: Current staff list as a CSV file
Version: 1.0.0
*/

(async function (params, userID, finish) {
  const csvCell = (v) => {
    const s = v === null || v === undefined ? "" : String(v)
    return /[",\n]/.test(s) ? `"${s.replace(/"/g, '""')}"` : s
  }

  try {
    const rows = await getDB().exec(
      "SELECT StaffID, Name, EMail, manager FROM Staff ORDER BY Name"
    )
    const header = rows.length ? Object.keys(rows[0]) : []
    const lines = [header.join(",")]
      .concat(rows.map((r) => header.map((k) => csvCell(r[k])).join(",")))

    finish(
      rawResponse(lines.join("\r\n"), {
        "Content-Type": "text/csv",
        "Content-Disposition": `attachment;
filename=staff_${Date.now()}.csv`
      })
    )
  } catch (error) {
    finish({ success: false, error: error.message })
  }
```

```
})(params, userID, finish)
```

4. Using an approved module and the caller's identity

`date-fns` is awaited out of `require`. The job also refuses to run for an anonymous caller, which is only possible because its name does *not* start with `public_` — WebHub has already checked the session, so `userID` is always set here; the check is belt and braces.

```
/*
Author: IQX Limited
Name: My week
Description: The calling consultant's placements starting in the next 7 days
Version: 1.0.0
*/

(async function (params, userID, finish) {
  if (!userID) {
    return finish({ success: false, error: "Authentication required" })
  }

  try {
    const { format, addDays } = await require("date-fns")
    const from = format(new Date(), "yyyy-MM-dd")
    const to = format(addDays(new Date(), 7), "yyyy-MM-dd")

    const rows = await getDB().exec(
      `SELECT p.PlacementID, p.StartDate, c.Name AS Company
         FROM Placement p
         JOIN Company c ON c.CompanyID = p.CompanyID
         JOIN IQXNetUser u ON u.StaffID = p.StaffID
        WHERE u.IQXNetUserID = ?
          AND p.StartDate BETWEEN ? AND ?
        ORDER BY p.StartDate`,
      [userID, from, to]
    )

    finish({ success: true, from, to, placements: rows })
  } catch (error) {
    console.error("my-week:", error.message)
    finish({ success: false, error: error.message })
  }
})(params, userID, finish)
```

5. A job written to be scheduled

Scheduled with cron `0 7 * * 1-5` and parameters `{"days": 3}`. It has no caller, so it leaves a trail in the log and signals failure the only way a schedule can see — `success: false` — which `NetV2UpdateJobSchedule` records as a failed run.

```
/*
Author: IQX Limited
Name: Expiring compliance reminder
Description: Flags candidate documents expiring within N days (default 3)
Version: 1.0.0
*/

(async function (params, userID, finish) {
  const days = Number(params.days) || 3
  const db = getDB()

  try {
    const expiring = await db.exec(
      `SELECT PersonID, DocumentTypeID, ExpiryDate
         FROM PersonDocument
        WHERE ExpiryDate BETWEEN today() AND dateadd(day, ?, today())
          AND ReminderSent = 0`,
      [days]
    )

    let flagged = 0
    for (const doc of expiring) {
      await db.exec(
        `UPDATE PersonDocument SET ReminderSent = 1
          WHERE PersonID = ? AND DocumentTypeID = ?`,
        [doc.PersonID, doc.DocumentTypeID]
      )
      flagged++
    }

    console.log(`compliance reminder: ${flagged} document(s) flagged
(${days} days)`)
    finish({ success: true, flagged, days })
  } catch (error) {
    console.error("compliance reminder failed:", error.message)
    finish({ success: false, error: error.message })
  }
})(params, userID, finish)
```

(Table and column names in examples 2, 4 and 5 are illustrative — check them against the schema before using.)

Things that bite

- **Top-level `await` does not work.** The file is a function body, not a module. Wrap in `(async function () { … })()` as every example does. The old README example without the wrapper will not run.

- **`require` is async.** `const { format } = await require ( "date-fns" )`. Forgetting the `await` gives you a promise with no `format` on it.
- **Call `finish` on every path**, including inside `catch`. A path that doesn't is a 10-second wait and a `JOB-003`.
- **`success: false` replaces your payload** with an error for the caller. Don't use it for an empty-but-valid result.
- **GET parameters are strings.** `params.limit === 50` is never true on a GET; convert first.
- **The 10 s limit is real** and `extendLimits` does not currently raise it. Keep jobs short; for long work, write the outcome somewhere and poll.
- **Two nodes, two folders.** Save or upload a job on both, or put `jobs/` on a shared volume; schedules fire on every node that has the row loaded.
- **Hot reload is free.** The file is read on every request, so a saved change is live immediately — on that node.
- **No `process`, `fs` or arbitrary `require`** are offered, and nothing in the wrapper stops a determined author reaching them. Jobs are trusted code; keep authoring to OWNER.

For developers

- Engine: `src/routes/webJobs.ts` (`executeWebJob`, `APPROVED_MODULES`, `rawResponse`). Routes, history, metrics and the cron scheduler: `src/routes/jobsManagement.ts`, mounted at `/api/webjobs` in `src/routes.ts`. Name and `public_` rules: `src/routes/webJobAuth.ts`.
- The scheduler is started from `server.ts` (`updateSchedulesFromDb`) at boot.
- Error classification is `toJobError`; a new failure kind needs a `JOB-00n` entry in `@iqx-limited/web-errors` and a row on **Web error codes**.
- Tests: `src/test/webjobs.ts` (note it still calls `test` unauthenticated, which the `public_` rule now refuses — rename the fixture or add a token).

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

Permanent link:
https://iqxusers.co.uk/iqxhelp/doku.php?id=iqxwebnext:technical:web_jobs&rev=1791045002

Last update: **2026/10/03 16:30**