

Baton

User manual for macOS, Windows, Linux and the web

Version 0.2.11 · 26.10.2026

Contents

1	About Baton	4
1.1	The pieces	4
1.2	What is in a workspace	4
1.3	What this manual covers	5
2	The window	6
2.1	The sidebar	7
2.2	The command palette	7
2.3	The status rows	9
3	Workspaces	10
3.1	Opening and creating	10
3.2	Settings that belong to the workspace	10
3.3	What a workspace link is	10
4	Collections and requests	11
4.1	Adding things	11
4.2	Bringing a collection in	11
4.3	Following an OpenAPI spec	13
4.4	Taking one back out	15
4.5	Several requests at once	15
5	Sending a request	17
5.1	The URL bar	17
5.2	What the tabs hold	17
5.3	The execution pipeline	18
5.4	Reading the response	18
5.5	The other views of a run	19
5.6	Saved responses	20
6	Variables and environments	22
6.1	Writing a reference	22
6.2	Switching environments	22
6.3	Editing values	23
6.4	Marking production	23
7	Secrets	25
7.1	What is stored where	25
7.2	Declaring one and giving it a value	25
7.3	Values from HashiCorp Vault	27
7.4	When a secret is missing	28
7.5	What never contains a secret	29
8	Authorization	30
8.1	OAuth 2.0	30
9	Request bodies	32
10	Scripts and tests	33
10.1	What a script can do	33
10.2	Tests	34
11	GraphQL and WebSocket	35
11.1	GraphQL	35
11.2	WebSocket	37
12	Running a collection	39

12.1	Reading a failure	39
13	Git sync and the worker	41
13.1	How sync works	41
13.2	The worker	42
13.3	In CI	42
13.4	Mock servers	43
13.5	Why a browser needs one	43
14	AI agents	44
14.1	Connecting an agent	44
14.2	Through the app, or on the files	44
15	Settings, licensing and trouble	46
15.1	Settings	46
15.2	Licensing	47
15.3	When something does not work	47
15.4	Keeping up to date	48
16	The desktop build	49
16.1	A workspace is a real folder	50
16.2	Cookies work	51
16.3	Git without an intermediary	51

1 About Baton

Baton is an API client whose collections, environments and schedules are plain YAML files in a git repository you own. There is no cloud account holding your work and no proprietary sync protocol: the repository is the product’s backbone, so anything git can do — branches, review, history, blame — applies to your requests as well.

It runs from one codebase on macOS, Windows, Linux, iOS, Android and in a browser, and each kind of screen has a guide of its own. This one is for a computer: macOS, Windows, Linux, and the web app. Its screenshots are of the browser build at baton.dudko.dev/app, version 0.2.11, in a desktop-sized window; the desktop builds draw the same screens, and Section 16 says what they add. On a phone, read [the guide for iPhone and Android](#).

1.1 The pieces

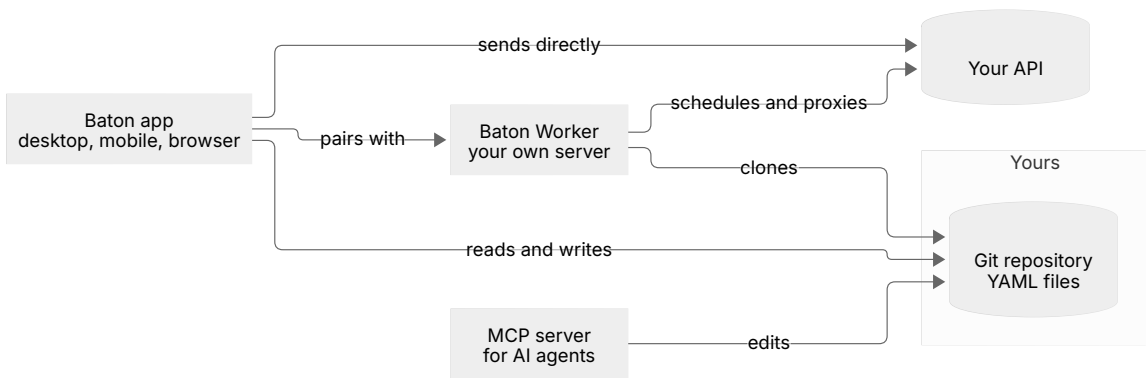


Figure 1: What talks to what. Everything meets at your repository.

The **app** is what you are looking at. The **worker** is an optional server you run yourself — it executes scheduled runs and relays requests for the browser build. The **MCP server** lets an AI agent — Claude Code, Cursor, VS Code, Codex — read, edit and run requests, through the open app when there is one (Section 14). None of them is a service of ours; the worker and the MCP server are programs you start.

1.2 What is in a workspace

A workspace is a directory, and everything in it is readable without Baton.

Path	What it holds
collections/	One YAML file per request, in folders that mirror the tree
environments/	One file per environment: variable values, and the <i>names</i> of secrets
schedules/	What the worker should run, and when
.baton/	Run history and local state — not something you edit
baton.yaml	Workspace settings: timeouts, redirects, TLS, proxy

Note

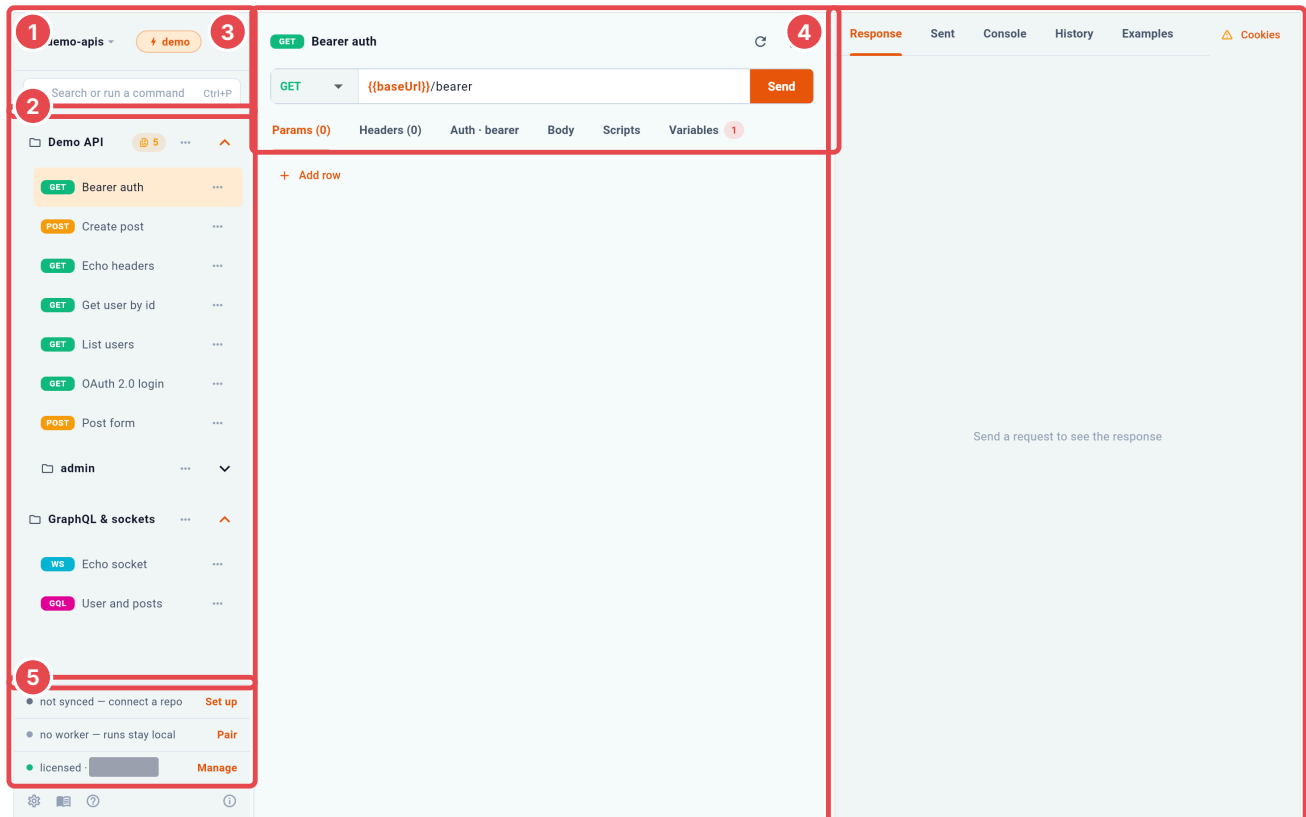
Secret *values* are deliberately absent from that table. They live in the operating system’s secure storage, never in the workspace — see the chapter on secrets 7. A repository that leaks must not leak credentials with it.

1.3 What this manual covers

The chapters follow the order you meet things in: the window, then a workspace and the requests in it, then sending one and reading what came back, then the parts that make a request reusable — variables, secrets, auth, bodies, scripts — and finally collection runs, git sync, the worker, AI agents and settings.

2 The window

Baton puts three columns on screen and keeps them there: what you have, what you are editing, and what came back.

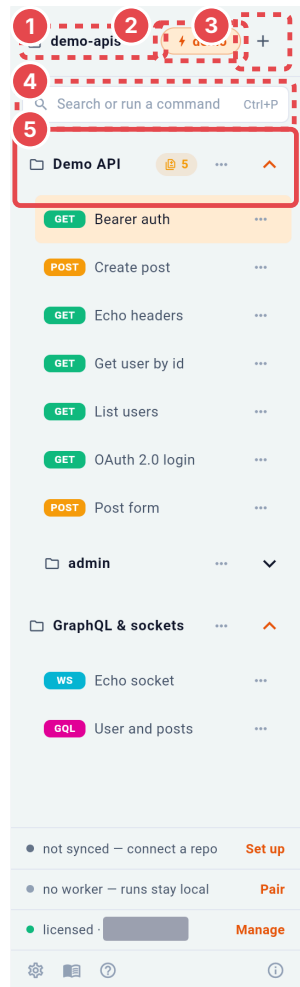


- 1 Workspace, environment and new-item menu
- 2 Collections and requests
- 3 The request: method, URL and Send
- 4 The response panel
- 5 Sync, worker and license status

Figure 2: The whole window, with a request open and nothing sent yet.

1 names the workspace and the active environment, and opens the menu for new collections and imports. 2 is the tree of collections and requests. 3 is the request itself — method, URL, **Send**, and the tabs under them. 4 is the response panel, which also holds what was actually sent, the console, the history and saved examples. 5 reports three things that are easy to forget: whether the workspace is synced, whether a worker is paired, and whether a license key is present.

2.1 The sidebar



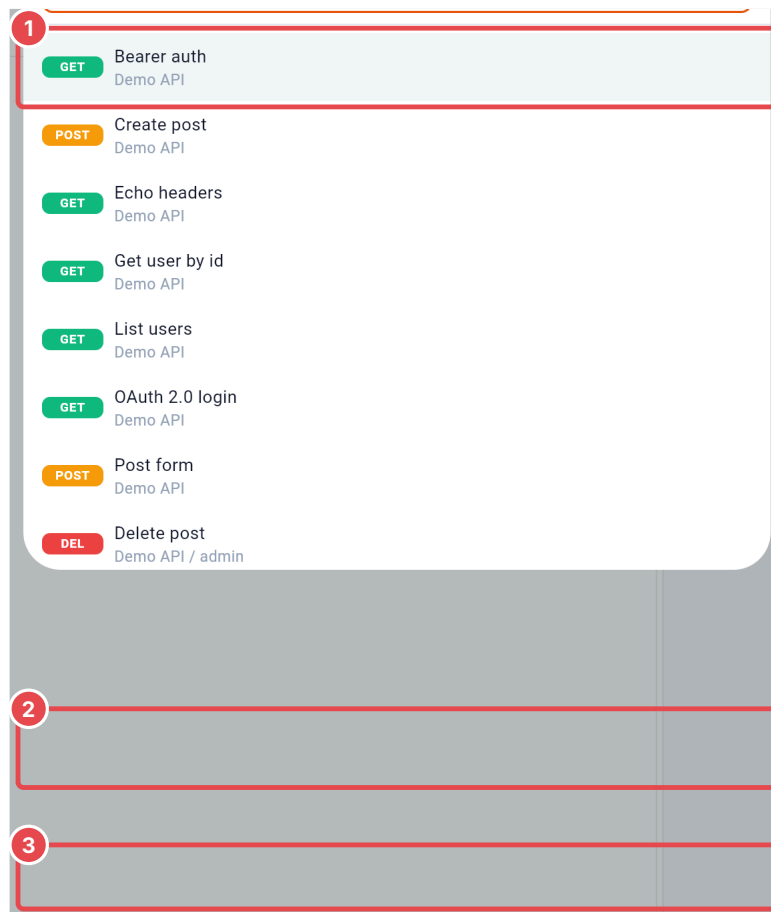
- 1 Switch or create a workspace
- 2 Active environment
- 3 New collection, or import
- 4 Command palette
- 5 A collection — click to fold

Figure 3: The sidebar, top to bottom.

The workspace name 1 opens the list of workspaces and the way to add one. The chip beside it 2 is the active environment: every `{{variable}}` in every request resolves against whichever one is selected here. 3 adds a collection or imports one. 4 opens the command palette, which is usually faster than the tree. A collection header 5 folds and unfolds.

2.2 The command palette

`Ctrl+P` — `Cmd+P` on macOS — opens one list over everything else.



- 1 Every request, with its collection
- 2 Run a whole collection
- 3 Jump to variables

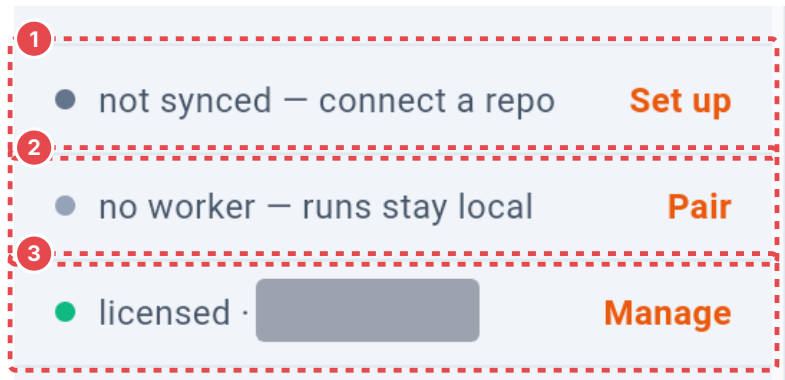
Figure 4: The palette searches requests, and also runs them.

It lists every request with the collection it belongs to 1, including ones inside folded folders. Below them it offers to run a whole collection 2, and to jump straight to the variables and secrets of the active environment 3.

Tip

The palette is the quickest route to a request in a workspace too large to scroll. Type part of a name; the collection shown beside each result tells two similarly named requests apart.

2.3 The status rows



- 1 Connect a repository
- 2 Pair a worker
- 3 License state

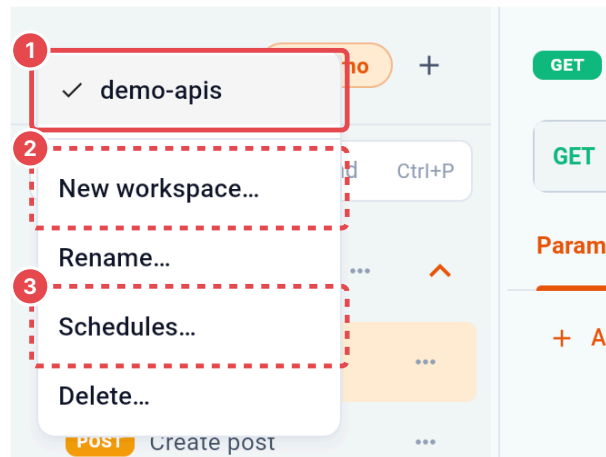
Figure 5: Three states, each with the action that changes it.

These are not decoration — each row is a button. 1 opens git sync, 2 opens worker pairing, 3 opens the license dialog. The row shows the state first and the action second, so a glance is enough.

3 Workspaces

A workspace is a directory of YAML files. On desktop you point Baton at a folder and that folder is the workspace; in a browser there is no folder to point at, so the build keeps one in the browser's own storage and git sync is how it leaves.

3.1 Opening and creating



- 1 The workspaces you have opened
- 2 On the desktop, pick a folder
- 3 What the worker runs, and when

Figure 6: The workspace menu.

The workspace name at the top left opens the list. 1 is what you have open. 2 adds one — on the desktop it asks for a folder, because a workspace *is* a folder, so there is nothing else to name. 3 opens the schedules the worker runs. The bundled `demo-apis` workspace is always there to try things against; it is the one this manual uses throughout.

Warning

The browser build keeps your workspace in the browser. It survives a reload, but clearing site data clears it. Connect a repository (Git sync and the worker 13) before you put work into it that you would mind losing.

3.2 Settings that belong to the workspace

`baton.yaml` carries the settings every request in that workspace inherits: request timeout, whether redirects are followed, whether TLS certificates are validated, an extra CA bundle or client certificate, and a proxy. They are edited in **Settings**, and because they live in the workspace they travel with it through git — a colleague who clones the repository gets the same behaviour.

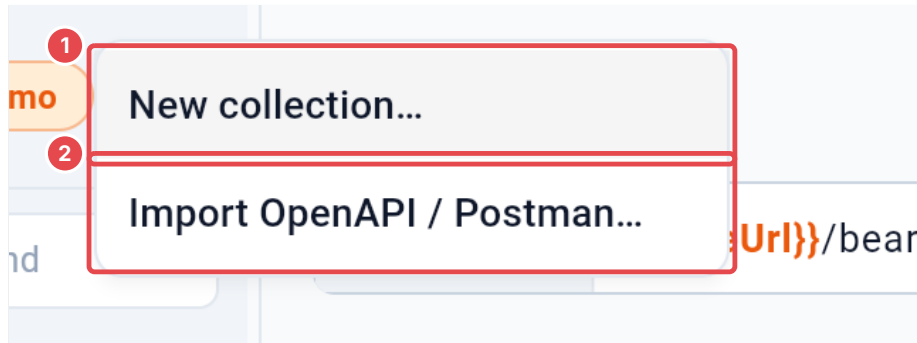
3.3 What a workspace link is

A workspace can be published as a link that opens it: an `https://` link to the site, which renders the repository's coordinates and offers the app, and a `baton://` scheme link for platforms with no universal links. Both carry coordinates only. A token, a password or anything resembling a credential in such a link is refused outright rather than quietly stripped.

4 Collections and requests

A collection is a folder of requests, and it may contain folders of its own. The tree in the sidebar is the directory on disk — renaming a collection renames a directory, and git sees exactly that.

4.1 Adding things

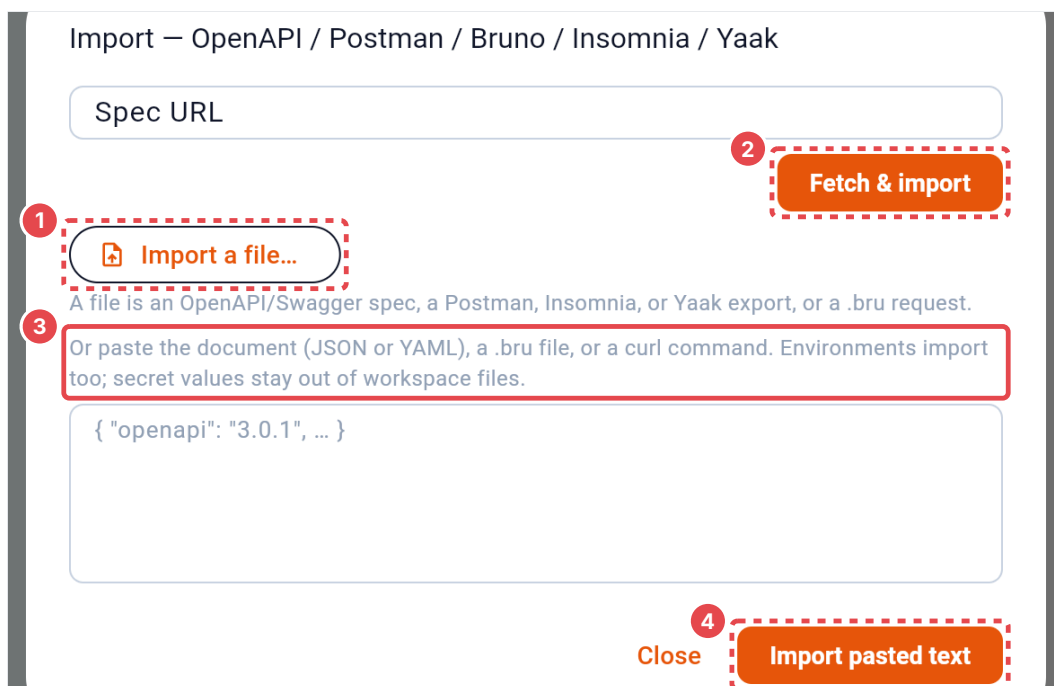


- 1 A new collection
- 2 Bring one in from elsewhere

Figure 7: The plus button: start something, or bring it in.

1 creates an empty collection. 2 imports one — from OpenAPI, Postman, Bruno, Insomnia, Yaak or a `curl` command — which is usually how an existing project arrives.

4.2 Bringing a collection in



- 1 A file from disk
- 2 Or the URL above, fetched now
- 3 Or pasted text — a spec, or a curl command
- 4 Import what was pasted

Figure 8: Four ways in; every one of them ends as files in the workspace.

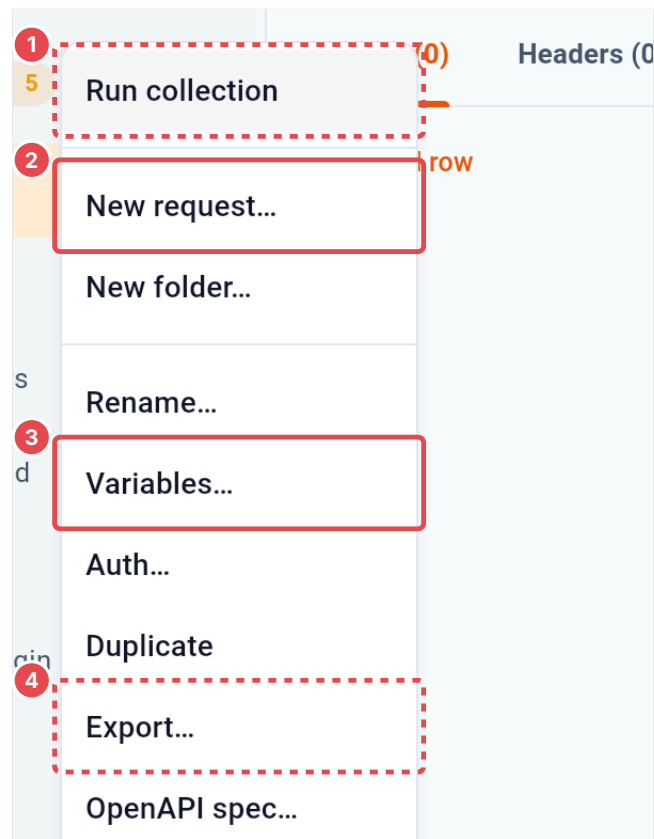
❶ reads a file from disk. ❷ fetches the URL typed in the field above — a hosted OpenAPI document is usually one link. ❸ is for pasted text: a spec, or a `curl` command, which is the shortest path from a request copied out of a browser's network tab to a saved one. ❹ imports what was pasted.

The format is detected, so there is nothing to pick. OpenAPI 3 and Swagger 2, Postman collections and environments, Insomnia and Yaak exports, a Bruno collection, one `.bru` file and a `curl` line all become the same thing — request files in Baton's format, written into the workspace and visible to git. On the desktop a whole folder can be chosen as well, which is how a Bruno collection, or an Insomnia or Yaak folder kept in git, comes in.

Secrets stay out of the files on the way in too. Where the source marks a value as secret — Bruno, a Postman environment — only the name lands in the environment file, and the value, if there was one, goes to this device's secure store (Section 7). Insomnia and Yaak exports mark nothing, so their values arrive as plain variables, and the import names the ones that look like credentials so they can be made secrets before the first commit.

What cannot be represented is reported rather than dropped. Scripts written for another client's API are counted and left out — Baton's `baton.*` API is different on purpose — and file parts or binary bodies, which point at paths on someone else's machine, arrive as names under `files/` for you to copy in.

Every collection header carries its own menu.



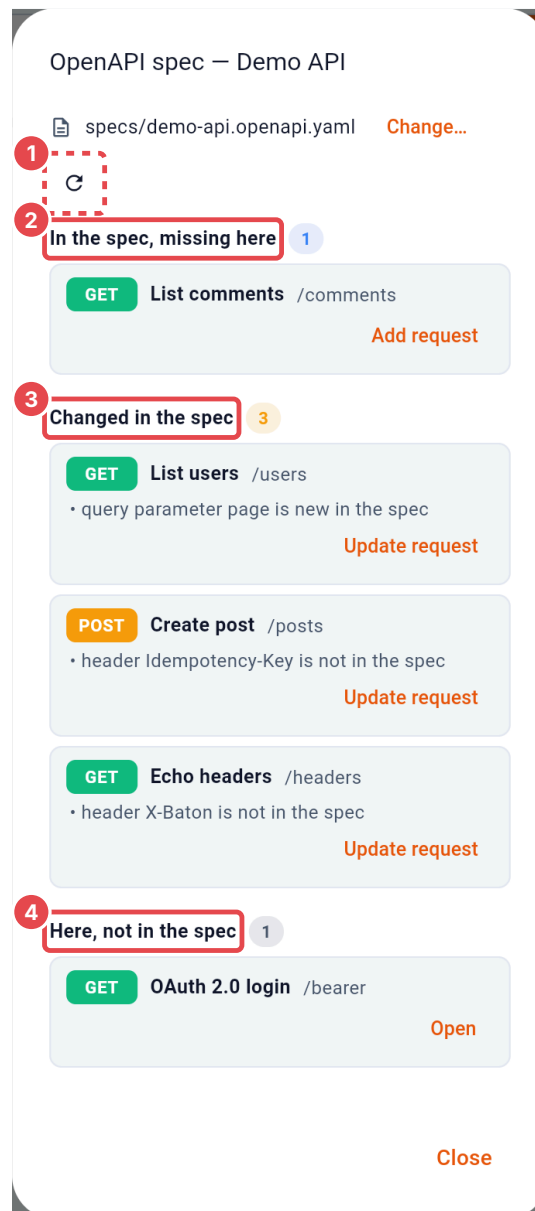
- 1 Run every request in order
- 2 Add a request or a folder
- 3 Variables and auth for the whole collection
- 4 Export to OpenAPI or Postman

Figure 9: What a collection can do.

1 runs every request in the collection in order — see Running a collection 12. 2 adds a request or a folder inside it. 3 opens variables and auth that apply to the whole collection, so a base URL or a bearer token is written once. 4 exports the collection back out to OpenAPI or Postman. At the foot of the menu, **OpenAPI spec...** links the collection to the document it follows — the next section.

4.3 Following an OpenAPI spec

A collection can name the OpenAPI document it follows: a file in the workspace, or an `https` address. An import from an address sets this by itself. From then on Baton holds the collection against the spec — when the workspace opens, after every reload and sync, and on request — and a badge on the collection header counts the differences.



- 1 Compare again
- 2 Operations with no request: Add request
- 3 Requests the spec moved on from: Update request
- 4 Requests the spec does not know

Figure 10: The demo collection against its spec: one operation missing, three that moved on, one the spec does not know.

1 compares again. What it found is sorted into three lists. 2 is operations the spec has and the collection does not; **Add request** writes the request an import would have written — the address, the documented parameters, a sample body, the spec's response examples — and opens it. 3 is requests the spec has moved on from: a new query parameter, a renamed path parameter, a body where there was none. **Update request** opens the request with those additions as an unsaved draft, which the banner over it accepts or discards; your own values are left as they are. 4 is requests the spec does not describe. Deleting one is your decision, so the only action there is **Open**.

Note

Names, auth, scripts and the values you typed are never counted as drift — only the shape the spec documents. The spec is read and never written: changing it is a change in whatever repository it lives in. A spec at an address is fetched only when this panel opens or **Recheck** is pressed, so opening a workspace never calls out on its own.

4.4 Taking one back out



- 1 Either format; the preview follows
- 2 What is left out, and what stays a reference
- 3 To the clipboard

Figure 11: Export shows you the document before you take it.

1 chooses the format and the text below follows the choice. 3 copies it; there is no file written and nothing sent anywhere.

2 is the part worth reading twice. An export carries the shape of the requests and none of the credentials: `{{baseUrl}}` leaves as `{{baseUrl}}`, and a secret is never resolved into the document, because the document is about to travel somewhere the workspace rules no longer apply.

4.5 Several requests at once

Opening a request adds a tab above the editor, and the tabs stay until you close them. The tab keeps unsaved edits, so moving to another request and back does not lose anything.

Note

A dot beside the name means the request has edits that are not in the file yet. **Save** writes

them; the reload button next to it re-reads the file and drops them, which is also how you pick up a change that arrived by git pull or from an agent.

5 Sending a request

5.1 The URL bar



- 1 Method
- 2 URL, with `{{variables}}` highlighted
- 3 Send — a square while it runs, which stops it
- 4 Save to the workspace file
- 5 Re-read the file, dropping edits

Figure 12: Method, URL, and the two buttons that matter.

The method 1 is a menu. The URL 2 takes `{{variables}}`, and shows them in colour: orange when the reference resolves against the active environment, red and underlined when it does not — so a typo is visible before you send. 3 sends. 4 saves the request to its file, and 5 re-reads that file, dropping whatever is unsaved.

Tip

While a request is in flight the **Send** button becomes a square. Pressing it stops the request — not just the waiting: the connection is dropped, or the browser's fetch is aborted. A request against something that never answers otherwise runs until the workspace timeout, 30 seconds by default.

5.2 What the tabs hold



- 1 Query string rows
- 2 Request headers
- 3 Authorization, or inherit it
- 4 Body and its kind
- 5 Pre-request and test scripts

Figure 13: Everything about the request, one tab each.

1 is the query string, as rows you can switch off individually rather than editing by hand. 2 is request headers, the same way. 3 is authorization, covered in Authorization 8. 4 is the body and its kind. 5 is the pre-request and test scripts. A count in the tab label tells you a tab has something in it without opening it.

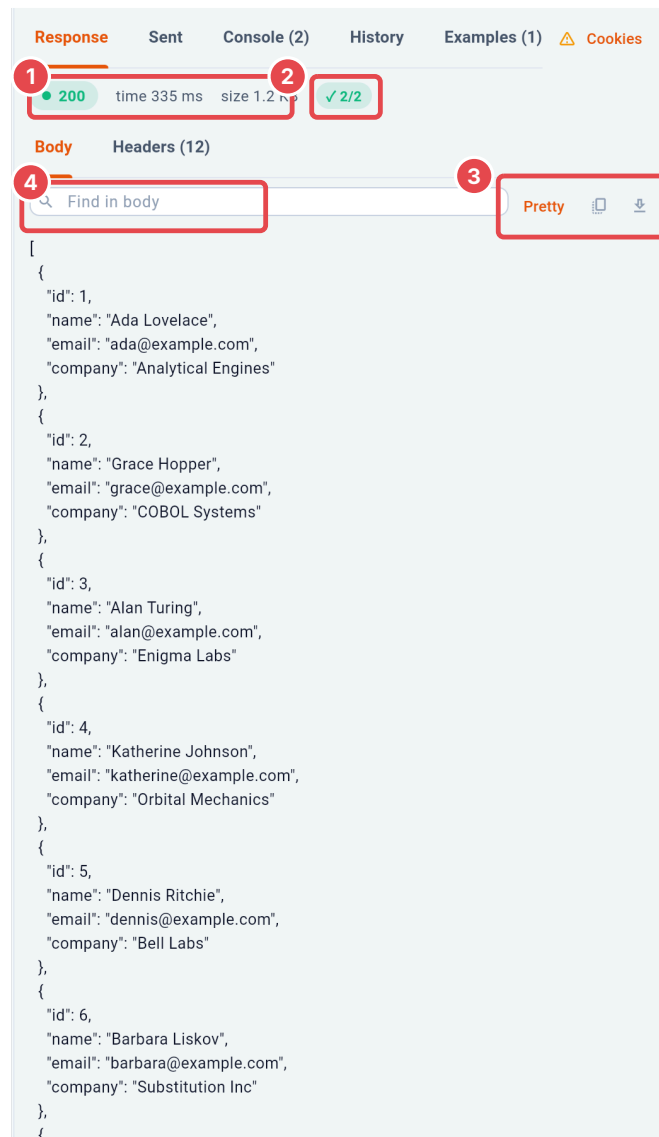
5.3 The execution pipeline

Pressing **Send** runs five steps in a fixed order, and the same order runs on the worker when a schedule fires.



Figure 14: One request, five steps. The worker does the same, so a scheduled run cannot behave differently.

5.4 Reading the response

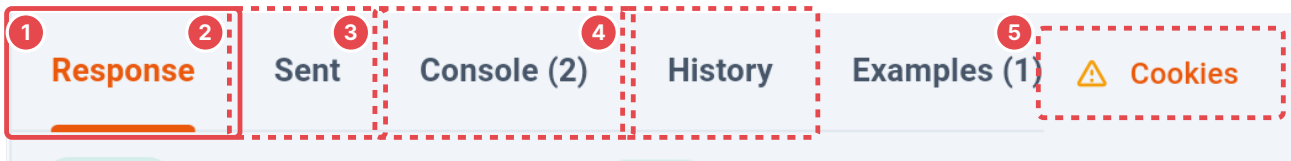


- 1 Status, time and size
- 2 Tests that ran after the response
- 3 Pretty-print, copy, save
- 4 Find in the body

Figure 15: A response that came back, with tests that ran on it.

1 is the status, how long it took and how big it was. 2 counts the tests that ran afterwards — green when they all passed. 3 pretty-prints the body, copies it, or saves it to a file. 4 searches inside the body, which is the fastest way through a long one.

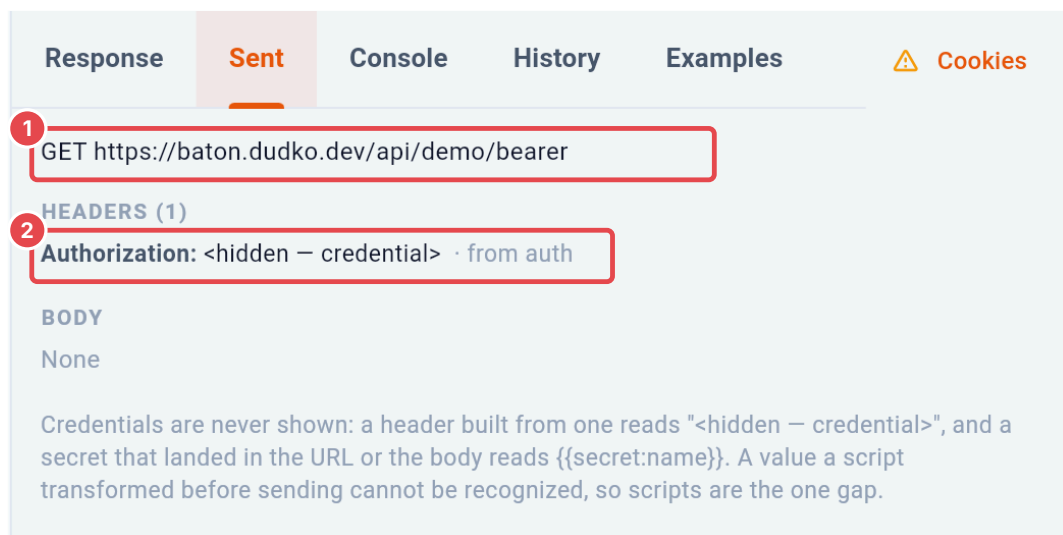
5.5 The other views of a run



- 1 What came back
- 2 What actually went out
- 3 Script output and test results
- 4 Earlier runs
- 5 The session cookie jar

Figure 16: The response panel's tabs.

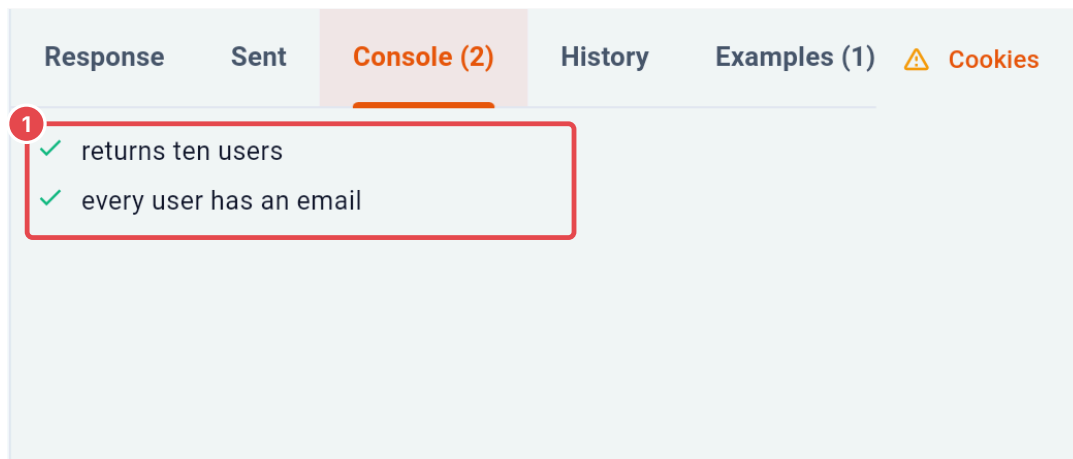
1 is what came back. 2 is what actually went out — headers included, each labelled with where it came from, which settles arguments about whether a header was really sent. 3 is script output and test results. 4 is earlier runs of this request. The tab after it, **Examples**, holds saved responses — see below. 5 is the session cookie jar.



- 1 The URL as it actually went out
- 2 A credential, hidden — and where it came from

Figure 17: What really went out — including what was hidden.

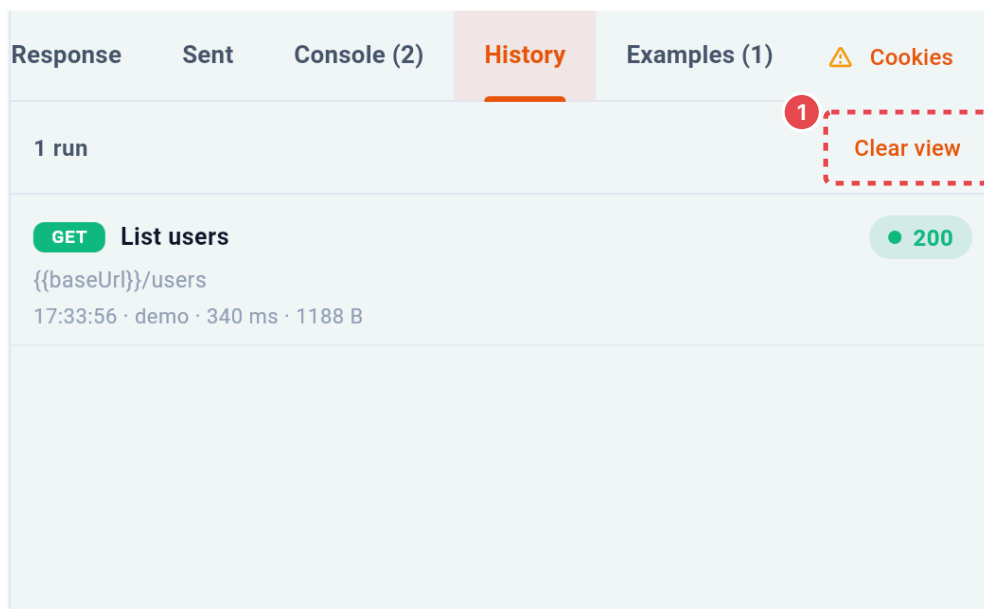
1 is the URL after every `{{variable}}` was resolved. 2 is a header Baton added from the request's auth: the name and its origin are shown, the value is not. A secret that landed in a URL or a body reads back as `{{secret:name}}` for the same reason — this view exists to be pasted into an issue.



- 1 One line per test, with its name

Figure 18: Each test by name, pass or fail.

- 1 lists one line per test. A failing test names the expectation that did not hold, so the console is usually enough to see what changed.



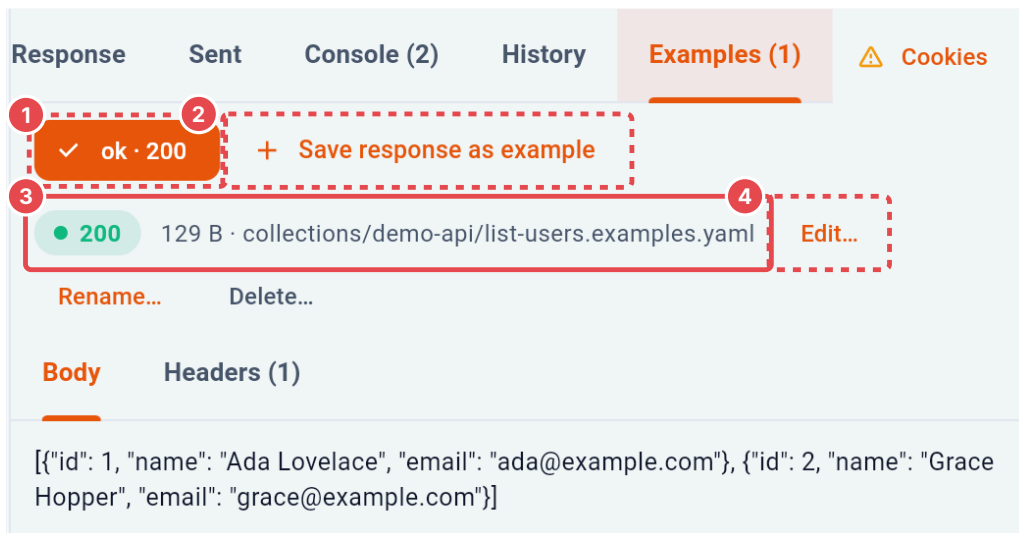
- 1 Clear the list

Figure 19: Earlier runs of this request.

History is per request and kept in `.baton/` inside the workspace — so it is yours, and it records the request as it was **written**, never as it was resolved. 1 empties the list.

5.6 Saved responses

A response worth keeping can be saved as an **example**: its status, the headers worth replaying and its body, in a file of its own beside the request. Examples are what a mock server answers with (Section 13), what an AI agent reads to learn what an endpoint returns (Section 14), and what an OpenAPI import fills in from the spec's own examples.



- ❶ The saved examples — pick one to see it
- ❷ Keep the response you just got
- ❸ A file of its own, beside the request
- ❹ Edit, rename or delete it

Figure 20: The Examples tab: what is saved, and a way to save what just came back.

❶ lists the saved examples by name and status; pressing one shows it. ❷ saves the response you just received as a new one, with a name suggested from its status — `ok` for a `200`. ❸ names the file it lives in, `list-users.examples.yaml` next to the request's own file, so saving an example never touches the request and shows up as a diff of its own. ❹ edits the example on screen, and renaming and deleting sit beside it.

Note

A saved example leaves out the headers that describe one particular connection — `date`, `content-length` and their kind — and `set-cookie`, which is a credential. Secret values are masked in it the way the **Sent** view masks them.

Note

In the browser build the cookie jar stays empty, and the tab says so. A page may neither set a `Cookie` header nor read `Set-Cookie` — that boundary is what stops a web page forging another site's session, so there is nothing to work around. Pair a worker, or use the desktop app, when cookies matter.

6 Variables and environments

A request that names a host once and is edited for staging is a request that will be sent at production by mistake. Variables exist so the request never mentions the host at all.

6.1 Writing a reference

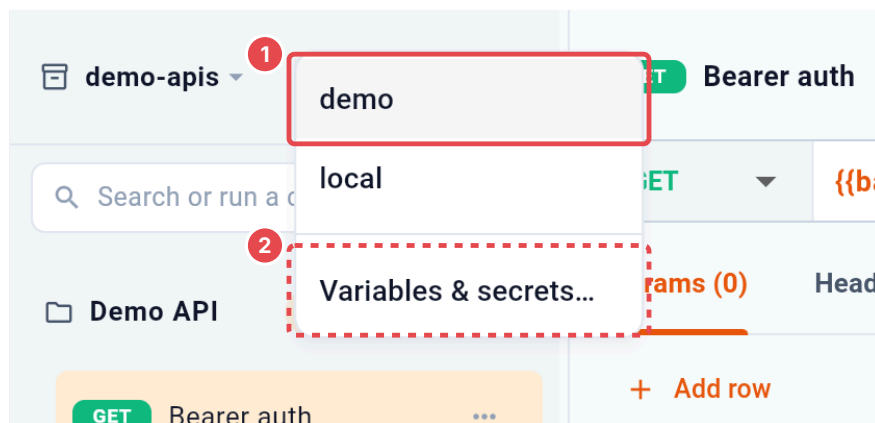
Anywhere a value is typed — URL, header, body, script — `{{name}}` stands in for a value. The editor colours it: orange when it resolves, red when it does not.

Form	Means
<code>{{baseUrl}}</code>	A variable, or a secret of that name when one is declared
<code>{{secret:apiToken}}</code>	A secret, and only a secret — fails rather than falling back
<code>{{timestamp}}</code>	A built-in, computed at send time

Note

The plain form is the one to reach for. `{{apiToken}}` already takes the secret wherever one is declared; the strict `{{secret:...}}` form means “fail rather than quietly use a variable of this name”, which is a decision worth making deliberately rather than by default.

6.2 Switching environments

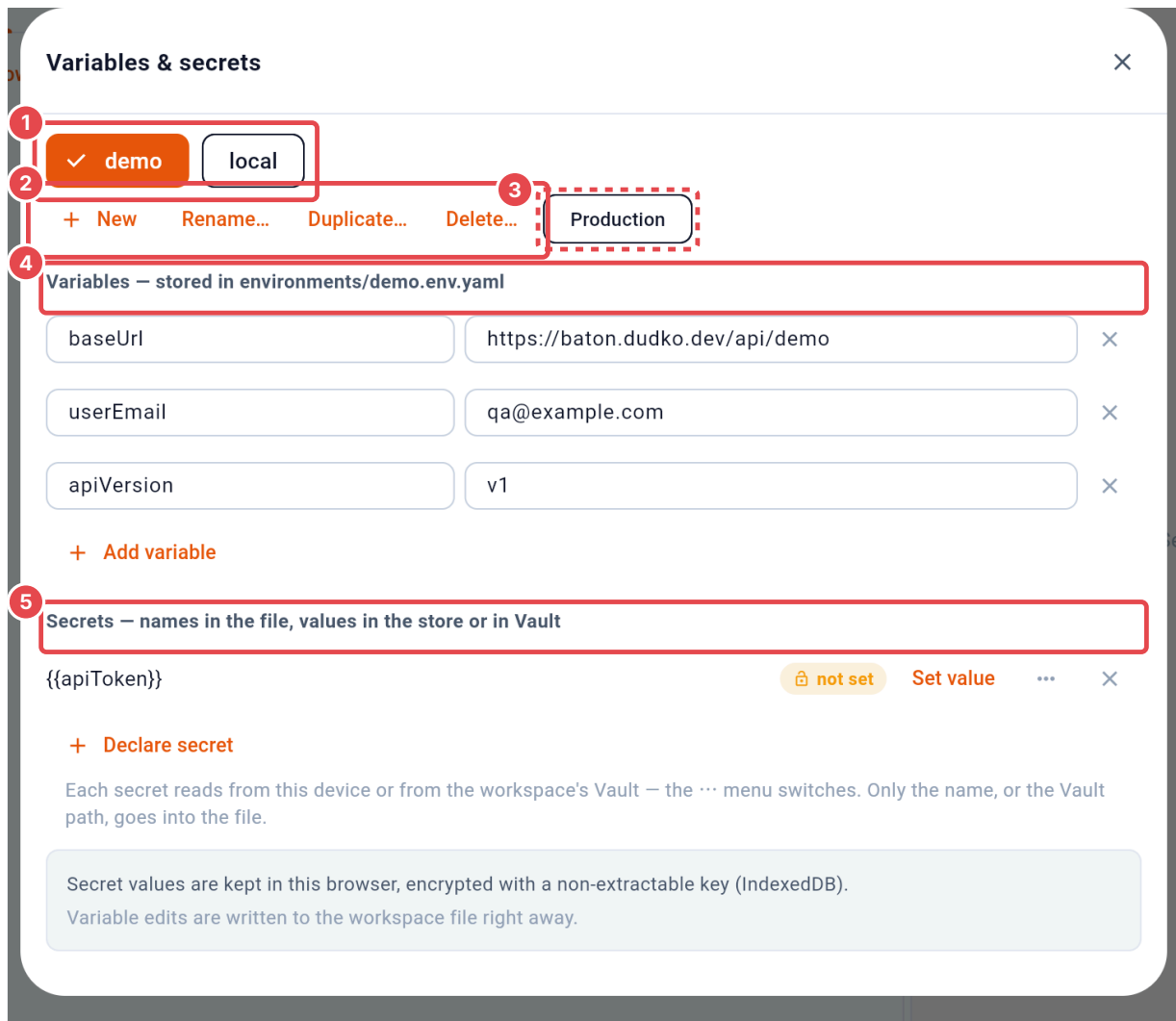


- 1 Environments defined in the workspace
- 2 Edit values and secret names

Figure 21: The environment chip lists what the workspace defines.

1 lists the environments in the workspace — here `demo` and `local`. Switching changes what every reference resolves to, with nothing else to edit. 2 opens the editor for values and secret names.

6.3 Editing values



- 1 Pick the environment to edit
- 2 Create, rename, duplicate, delete
- 3 Production: an agent's send asks you first
- 4 Plain values, written to the file
- 5 Secrets: only the name is stored

Figure 22: Variables and secrets, per environment.

1 picks which environment is being edited, independently of which one is active. 2 creates, renames, duplicates or deletes one — duplicating is how a staging is made from a production without retyping every row. 3 marks it as production, below. 4 is the plain variables, and the heading names the file they are written to. 5 is the secrets, which work differently enough to have their own chapter.

6.4 Marking production

3 changes nothing for you: requests into a production environment send as they always do. It changes things for an AI agent connected to the app (Section 14). A send from an agent into a production environment waits until you confirm it in the app, and if you refuse, the agent is told it

was your decision, not a failure to retry. The mark is `production: true` in the environment's file, so it travels with the repository to everyone who clones it.

Tip

Variables also exist at the collection level, from the collection's own menu. A base URL that only one collection uses belongs there rather than in every environment.

7 Secrets

The workspace is a git repository, and a repository gets cloned, shared and occasionally leaked. So Baton splits a secret in two: the **name** is part of the workspace, the **value** never is.

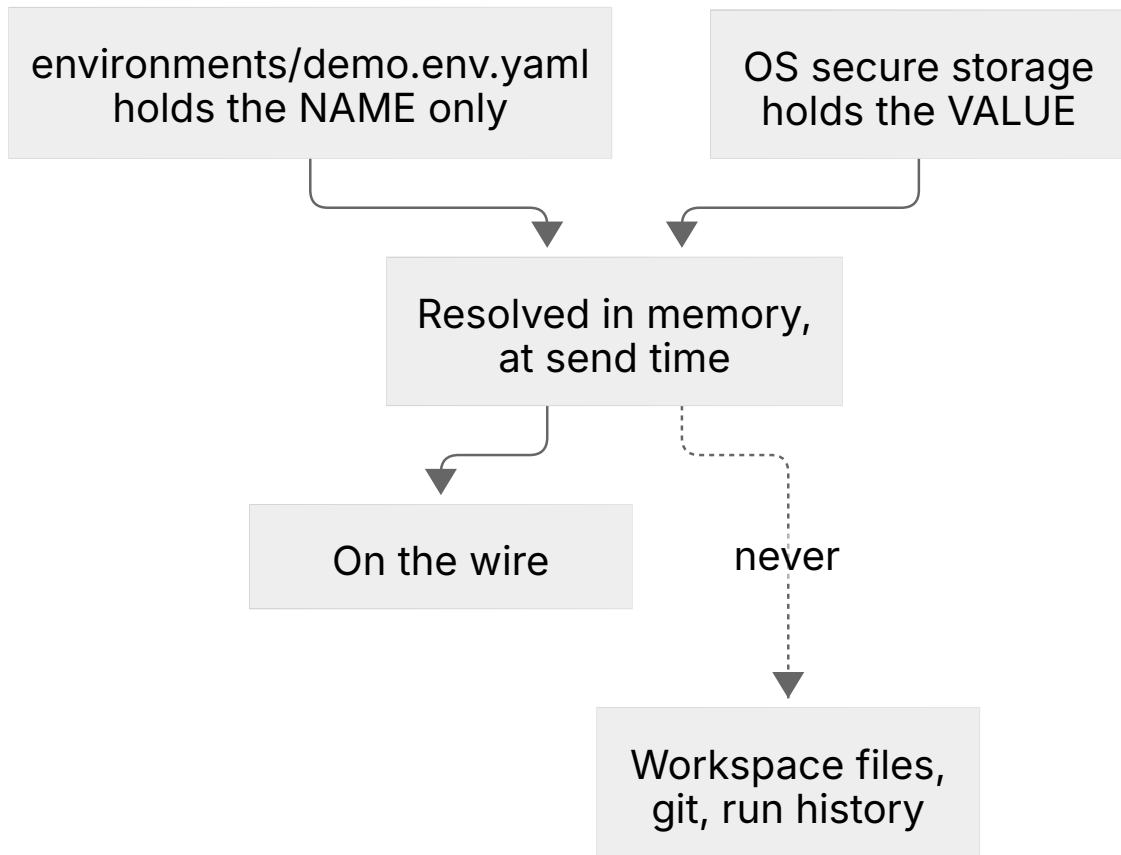


Figure 23: The name travels with the repository. The value does not travel at all.

7.1 What is stored where

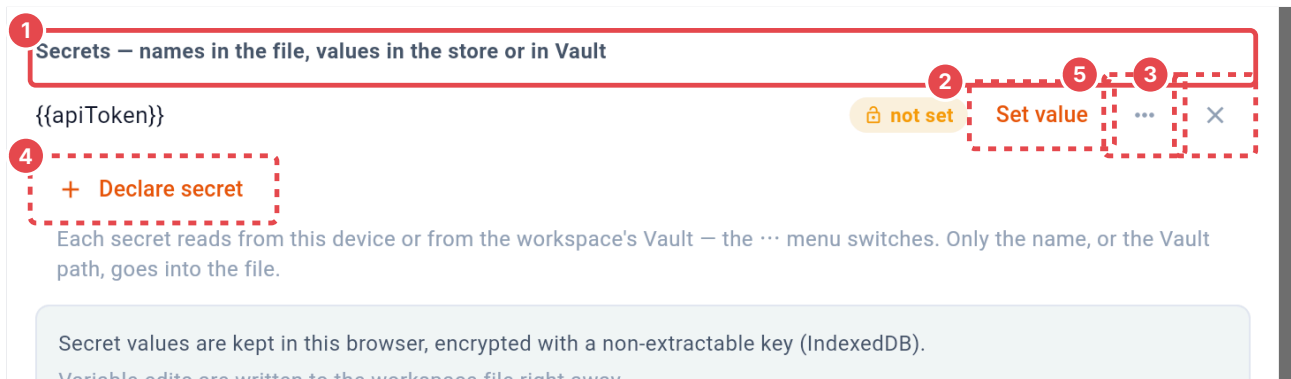
The environment file records that a secret called `apiToken` exists. The value sits in the operating system's secure storage — the Keychain on macOS, DPAPI on Windows, libsecret on Linux — and in the browser build, in storage encrypted with a key the page itself cannot extract — or in HashiCorp Vault, below. The two meet only in memory, at send time.

Warning

It follows that a secret value is **not** in the repository, so a colleague who clones it has the names and not the values, and must set their own. This is the intended behaviour, not a gap.

7.2 Declaring one and giving it a value

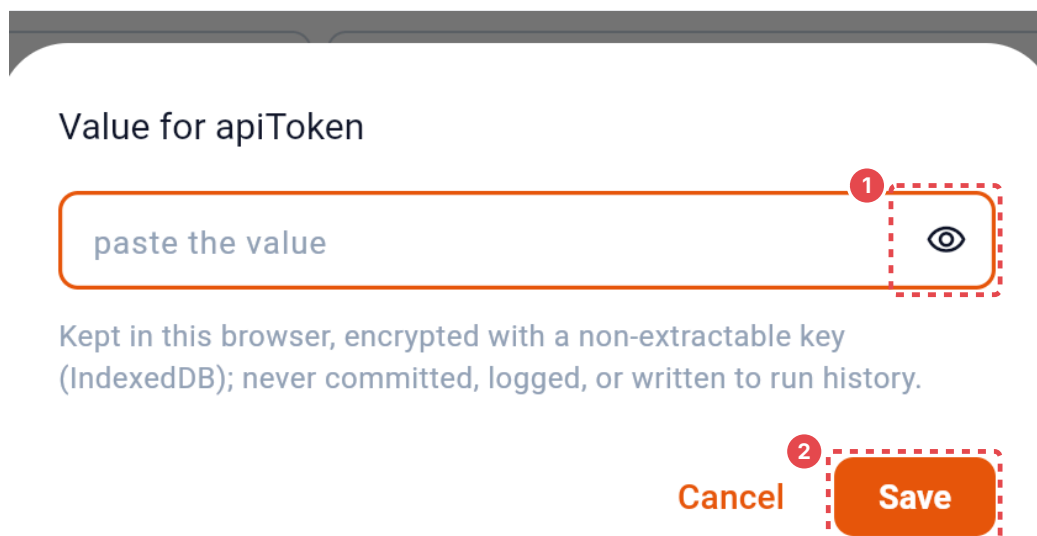
In `Variables & secrets` the lower half lists secrets.



- 1 Only the name is written to the file
- 2 Put the value in this machine's store
- 3 Forget both halves
- 4 Declare a new one
- 5 This device, or a path in Vault

Figure 24: The secrets half: a name, and whether this machine has a value.

1 says where the names are written — the environment file, which is in git. 2 sets the value on this machine. 3 forgets both halves at once. 4 declares a new one. 5 chooses where the value comes from: this device, or Vault.



- 1 Reveal what you typed
- 2 Store it — the file is untouched

Figure 25: Setting a value. It goes to the store, not to the file.

1 reveals what you typed, for checking a paste. 2 stores it.

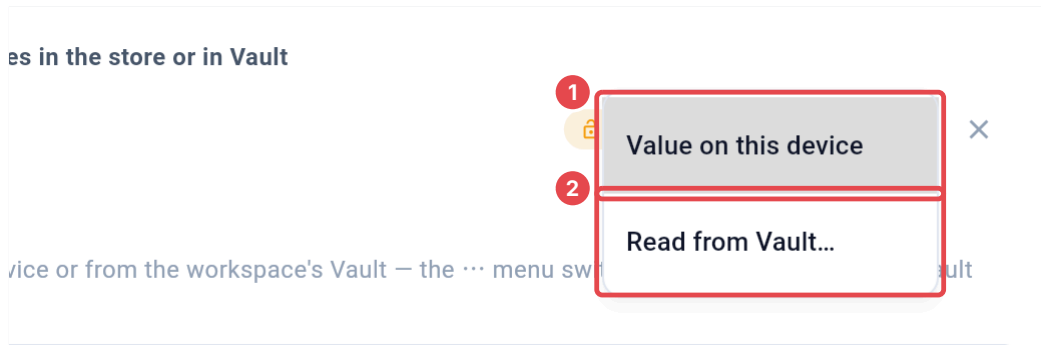
1. Press **Declare secret** and name it. The name goes into the environment file.
2. Press **Set value** on its row and paste the value. It goes to the store.
3. Use it as `{{apiToken}}` anywhere a value is typed.

The result is visible in **Sent** after the next run: the header is there, its origin is named, and the value is not shown.

Undeclare and forget the value removes both halves: the name leaves the file and the value leaves the store.

7.3 Values from HashiCorp Vault

A team that already keeps its credentials in HashiCorp Vault does not have to copy them into every machine's store. A secret can read its value from a Vault path instead: the path goes into the environment file, and the value is fetched when a request is sent and kept nowhere.

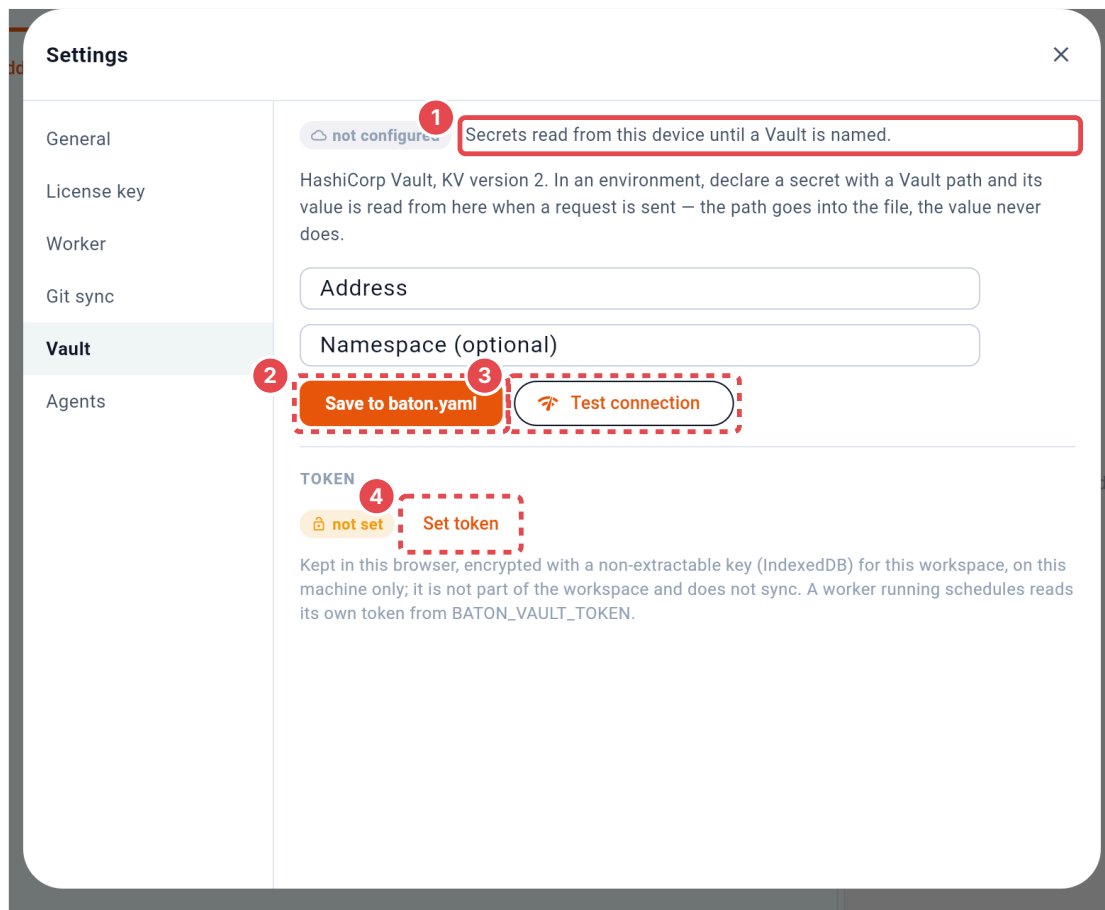


- 1 The OS secure store on this machine
- 2 A path in the workspace's Vault

Figure 26: The ... menu on a secret's row: where its value comes from.

1 keeps the value in this device's secure store, as above. 2 asks for a path in Vault instead — a KV version 2 path and the field in it, such as kv/billing/prod#password.

The Vault itself is set once per workspace, under **Settings** → **Vault** :



- 1 Whether a Vault is named
- 2 Address and namespace go into baton.yaml
- 3 Check the address and the token
- 4 The token stays on this device

Figure 27: The workspace's Vault: the address in the file, the token on this device.

1 says whether a Vault is named; until one is, every secret reads from this device. The address and the optional namespace belong to the workspace, and 2 writes them to baton.yaml, so a colleague's checkout and the worker read the same server. 3 looks the token up against the address as typed, so a wrong address or namespace shows before anything is saved. 4 sets the token — which, unlike the address, stays on this machine, in the same store as any other secret value. A worker running schedules takes its own token from BATON_VAULT_TOKEN.

7.4 When a secret is missing

A request that references a secret with no value does not get sent. The response panel says so plainly:

```
ResolveError: unresolved secrets: apiToken
```

That is the whole failure — nothing left the machine. Set the value and send again.

7.5 What never contains a secret

Baton goes out of its way to keep resolved values out of anything written down. Run history records the request as it was **written**, not as it was resolved. The **Sent** view masks credential headers and shows a placeholder in their place, so it can be screenshotted. Logs record sizes and hashes, never bodies or authorization headers. A snippet copied with **Copy as code** carries `{{secret:name}}` rather than the value, because a snippet exists to be pasted into a chat or an issue — exactly where a credential must not follow it.

8 Authorization

Auth is a property of the request, but most requests should not carry their own.

Token
`{{apiToken}}`

Reference a stored secret by its plain name — `{{apiToken}}` takes the secret wherever one is declared. Write `{{secret:apiToken}}` only where it must fail rather than fall back to a variable. Either way it is resolved in memory at send time and masked in logs.

- 1 Take the collection's auth
- 2 Basic, Digest, AWS SigV4
- 3 Bearer, API key, OAuth 2.0

Figure 28: Eight ways to authorize, and one that defers.

1 is the default and usually the right answer: the request takes whatever the collection, or the folder above it, defines. Set a bearer token once on the collection and every request in it is authorized; change it in one place when it rotates. 2 and 3 are the methods themselves.

Method	Notes
Inherit	Take the enclosing collection's or folder's auth
None	Send nothing, even if the collection defines auth
Basic	Username and password
Digest	Challenge-response; the first 401 is part of the exchange
AWS SigV4	Signed with access key, secret, region and service
Bearer	A token in Authorization
API key	A named value, in a header or the query string
OAuth 2.0	A flow that fetches a token, then sends it as a bearer

Tip

Put the token in a secret and reference it: `{{apiToken}}` in the Bearer field. The field then holds a name, which is safe to commit, and the value stays on the machine.

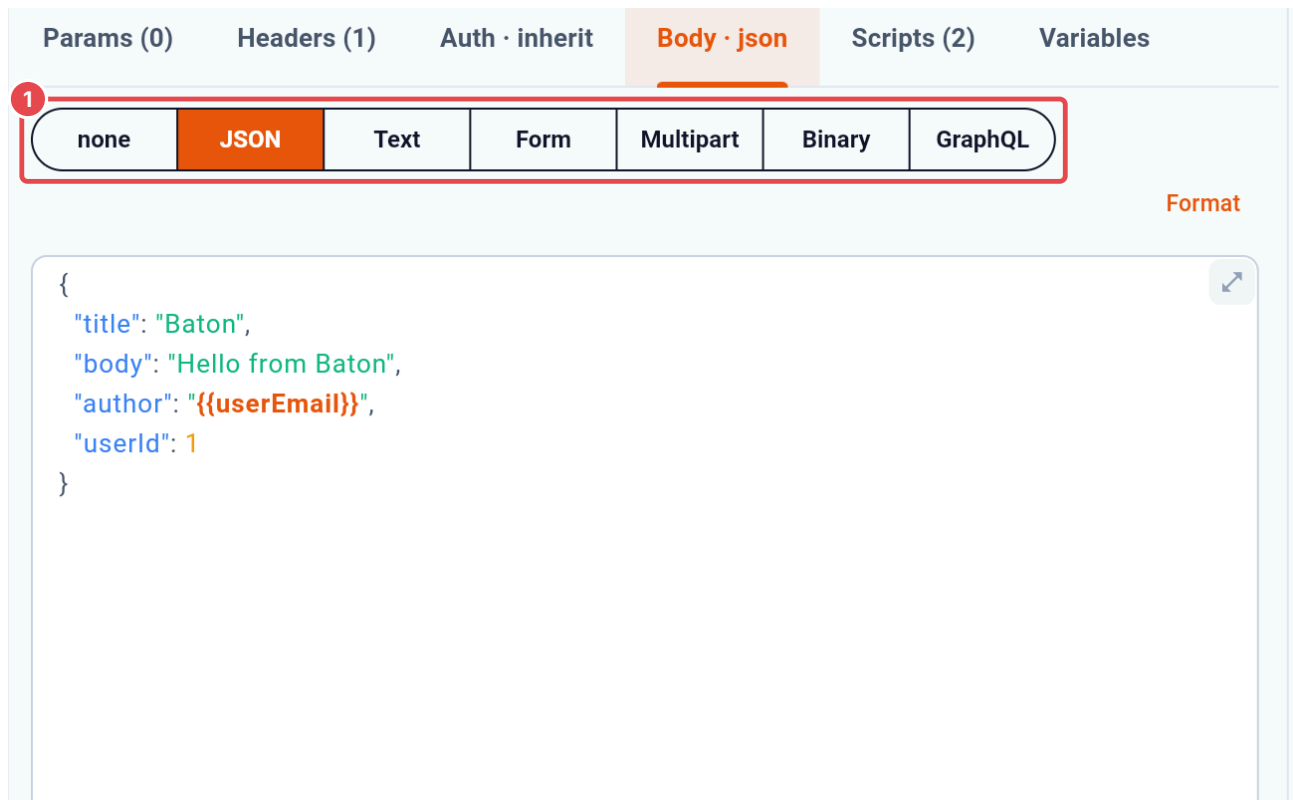
8.1 OAuth 2.0

OAuth is a token *acquisition* flow, so Baton resolves it before the request is built: a cached or refreshable token keeps it silent, and anything else asks you to authorize from the **Auth** tab. The authorization page opens in the system browser on desktop, and in the platform's own authentication session on iOS and Android. Discovery (RFC 8414) and dynamic client registration (RFC 7591) are supported, so an issuer URL is often all that is needed.

Note

A request whose OAuth flow has not produced a token yet fails with `oauth2 must be resolved to a bearer token before sending` rather than going out unauthorized. Authorize from the Auth tab and send again.

9 Request bodies



1 Body kind — the editor and the Content-Type follow it

Figure 29: The body kind decides the editor and the Content-Type.

1 picks the kind. The editor below changes with it, and so does the Content-Type Baton sends unless a header overrides it.

Kind	For
none	GET and anything else with nothing to send
JSON	The common case; the editor formats and folds it
Text	Anything you want sent verbatim
Form	application/x-www-form-urlencoded rows
Multipart	Rows that may be files — the boundary is Baton's
Binary	A file sent as-is
GraphQL	Query and variables as two panes — see GraphQL and WebSocket 11

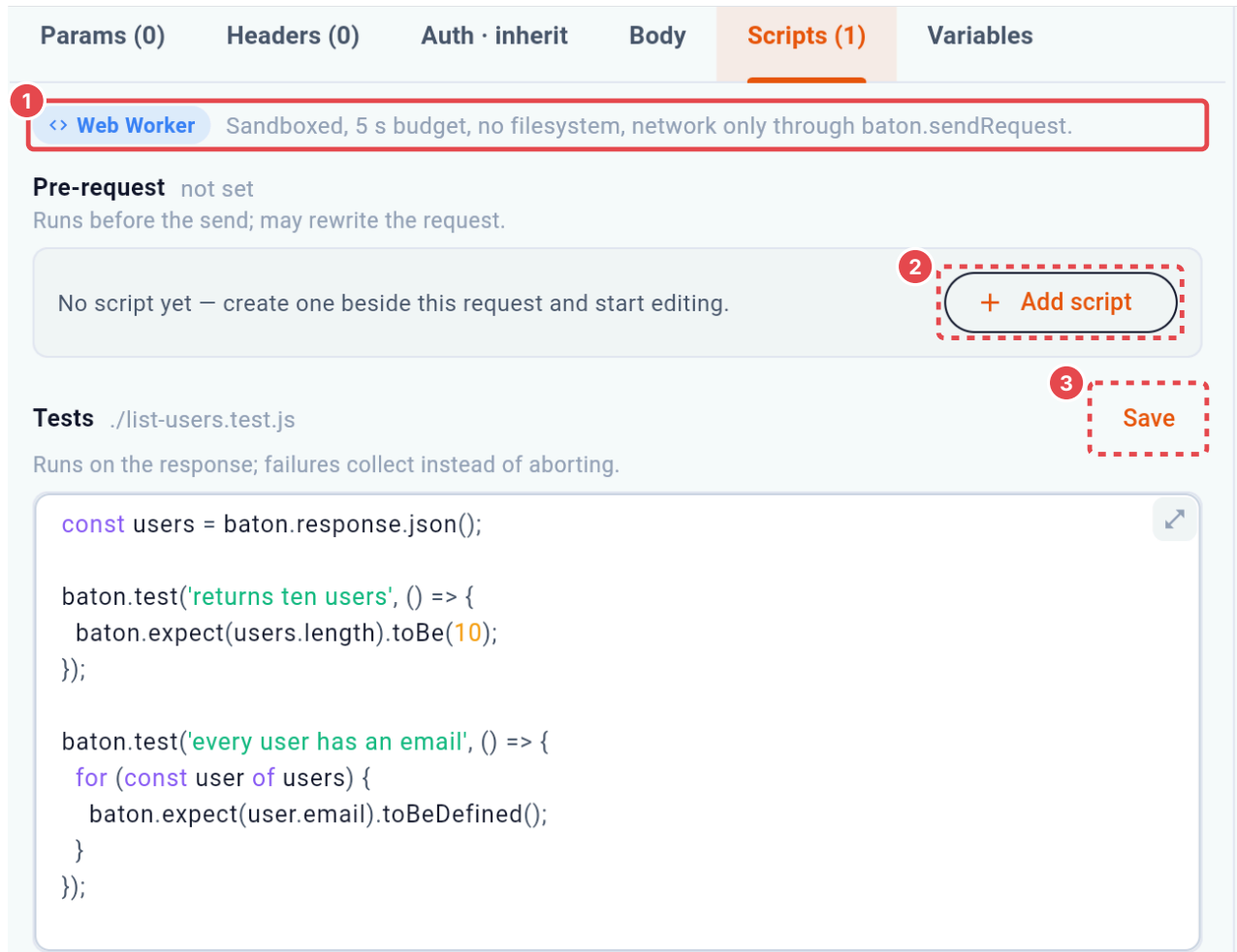
A JSON body takes `{{variables}}` like anywhere else, so a payload that differs only by an id between environments stays one request.

Note

Multipart and binary bodies are encoded before they leave the resolver, and sent exactly as encoded. The browser build sends them through `fetch` as a single envelope rather than letting the browser rebuild a `FormData`, so the boundary you see in **Sent** is the boundary that went out.

10 Scripts and tests

Two scripts hang off every request: one runs before it is sent, one after the response arrives. Both are JavaScript, and both are sandboxed — a timeout, no filesystem, no ambient network. The only way out is the `baton.*` API.



- 1 The engine, and what it is allowed to do
- 2 Add a pre-request or test script
- 3 Save the script with the request

Figure 30: The Scripts tab: pre-request above, tests below.

1 names the engine that runs them and what it may do. 2 starts a script, and 3 saves one with the request.

10.1 What a script can do

Call	Does
<code>baton.vars.get(name)</code>	Read a variable — never returns a secret
<code>baton.vars.set(name, v)</code>	Set one for the rest of the session
<code>baton.secrets.get(name)</code>	Read a secret — never returns a variable
<code>baton.request.*</code>	Inspect or change what is about to be sent

Call	Does
<code>baton.response.*</code>	Status, headers and body, in the test script
<code>baton.test(name, fn)</code>	Declare a test; the name appears in the console
<code>baton.sendRequest(...)</code>	Send another request from the script
<code>console.log(...)</code>	Write a line to the console

The split between `vars` and `secrets` is deliberate: a script cannot read a secret by guessing it is a variable, and the completion list offers each call only the names it can actually resolve.

10.2 Tests

A test is a named assertion that runs against the response:

```
baton.test("returns ten users", () => {  
  const users = baton.response.json();  
  if (users.length !== 10) throw new Error(`got ${users.length}`);  
});
```

Its name is what appears in the console, and in the collection runner's per-request count. Write the name as the thing that should be true — the console then reads as a list of statements about the API rather than a list of functions.

Note

The same script runs identically in the app and on the worker, which is what makes a scheduled run trustworthy: shared conformance fixtures keep the two engines in step, so "works here, fails on schedule" is not a thing that happens quietly.

11 GraphQL and WebSocket

11.1 GraphQL

A GraphQL request is an HTTP POST with a particular body, and Baton treats it as exactly that — there is no separate client underneath, so everything in the other chapters still applies.

The screenshot shows the Baton GraphQL interface. At the top, there are tabs for different request types: none, JSON, Text, Form, Multipart, Binary, and GraphQL (which is selected). Below the tabs, there is a pink button labeled 'GraphQL' and a text description: 'Introspect to list the fields this endpoint exposes.' To the right of this, there are two red dashed boxes with numbered circles. Box 1 contains an 'Introspect' button with a schema icon. Box 2 contains a 'Format' button. Below this, there is a 'Query' section with a large text area containing a GraphQL query:

```
query User($id: ID!) {  
  user(id: $id) {  
    name  
    email  
    posts(options: { paginate: { page: 1, limit: 3 } }) {  
      data {  
        title  
      }  
    }  
  }  
}
```

 To the right of the query text area is a 'Format' button. Below the query section, there is a 'Variables (JSON)' section with a text area containing:

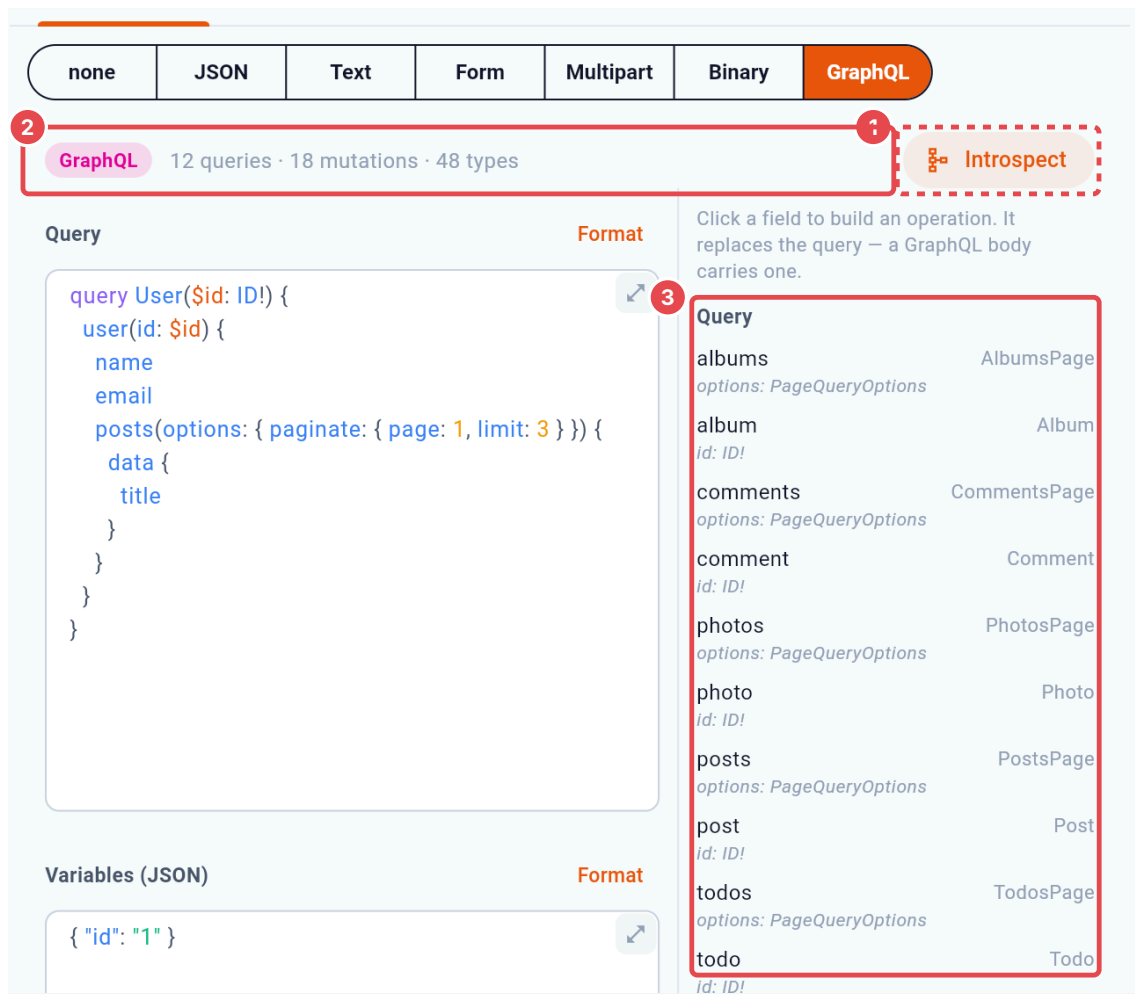
```
{ "id": "1" }
```

 To the right of the variables text area is another 'Format' button.

- 1 Ask the endpoint for its schema
- 2 Reformat the query

Figure 31: Query above, variables below, schema on request.

- 1 asks the endpoint to describe itself. Once it has, field names complete as you type the query.
- 2 reformats the query or the variables.



- 1 One press; the schema stays until the request changes
- 2 What came back
- 3 Every query and mutation, with its arguments and type — click one to build it

Figure 32: After introspection: the schema, next to the query it is for.

1 is that same button, after it has been pressed. 2 sums up what came back — here 12 queries, 18 mutations and 48 types — and 3 lists every query and mutation the endpoint has, with its arguments and its return type. Clicking a field writes that operation into the query pane. A GraphQL body carries one operation, so the new one replaces what was there, as the line above the list warns.

The schema belongs to the request, not to the window: choosing another request drops it, and 1 fetches it again.

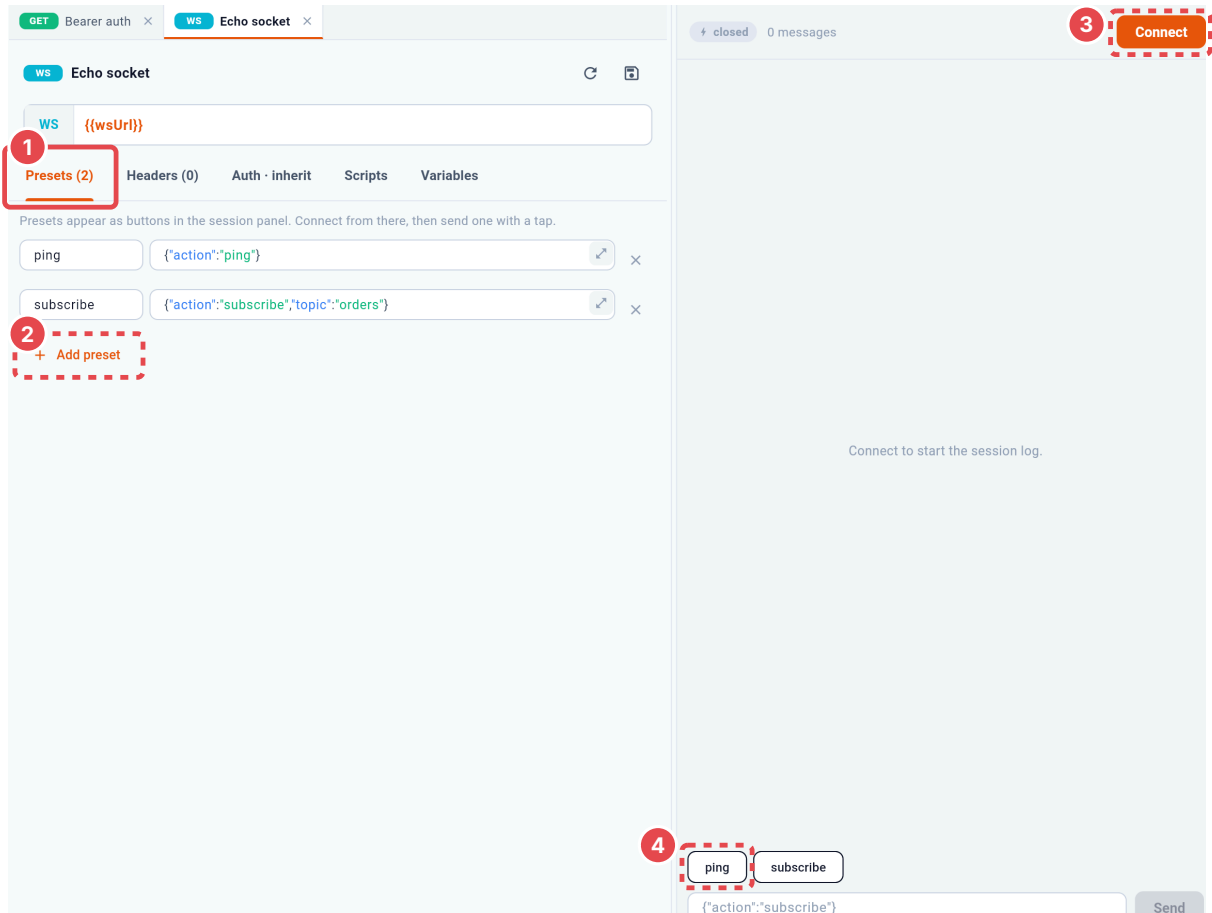
Note

Introspection is an ordinary query, and a server is free to refuse it. The usual refusal comes from a gateway that caps query depth — it answers the introspection query with an error instead of a schema, and Baton shows the status it got back. Nothing else about the request is affected; only completion and the field list go missing.

The query and the variables are two panes because variables are what change between runs; `{{variables}}` work inside both, so an id that differs per environment stays out of the query text.

Tip

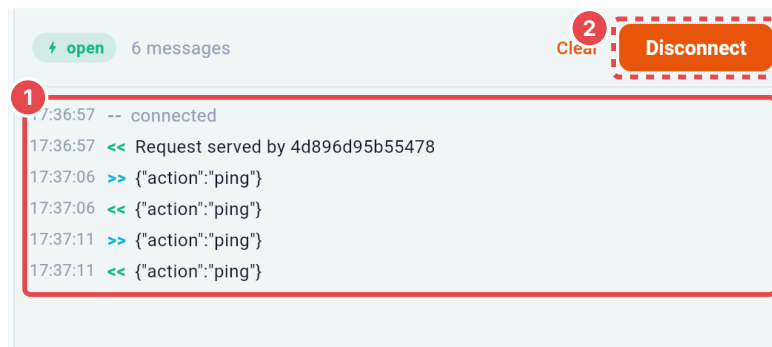
On a narrow window the two panes stack and the whole tab scrolls, rather than splitting a phone screen into two unusable halves.

11.2 WebSocket

- 1 Named messages to send over and over
- 2 Add one
- 3 Open the socket — the log fills below
- 4 A preset: one press sends it

Figure 33: A socket request: presets to send, and the session log.

A WS request connects rather than sends, and stays connected. **Presets** are named messages you send repeatedly — a subscribe frame, a ping — so a session is driven by pressing names rather than retyping JSON. The log below records both directions with timestamps, and survives until you disconnect or clear it.



- 1 Both directions, with the time of each frame
- 2 Close the socket

Figure 34: The same panel once it is connected.

1 is the log. >> is a frame Baton sent, << one that arrived, and a line with neither is the session talking about itself: connected here, a close code when it ends. Every line carries the time it happened, so a silent minute is visible as a silent minute. 2 closes the socket; the log stays.

Pressing a preset sends it at once and leaves a copy in the box at the foot of the panel, so the next frame can be an edit of the last one rather than a retype.

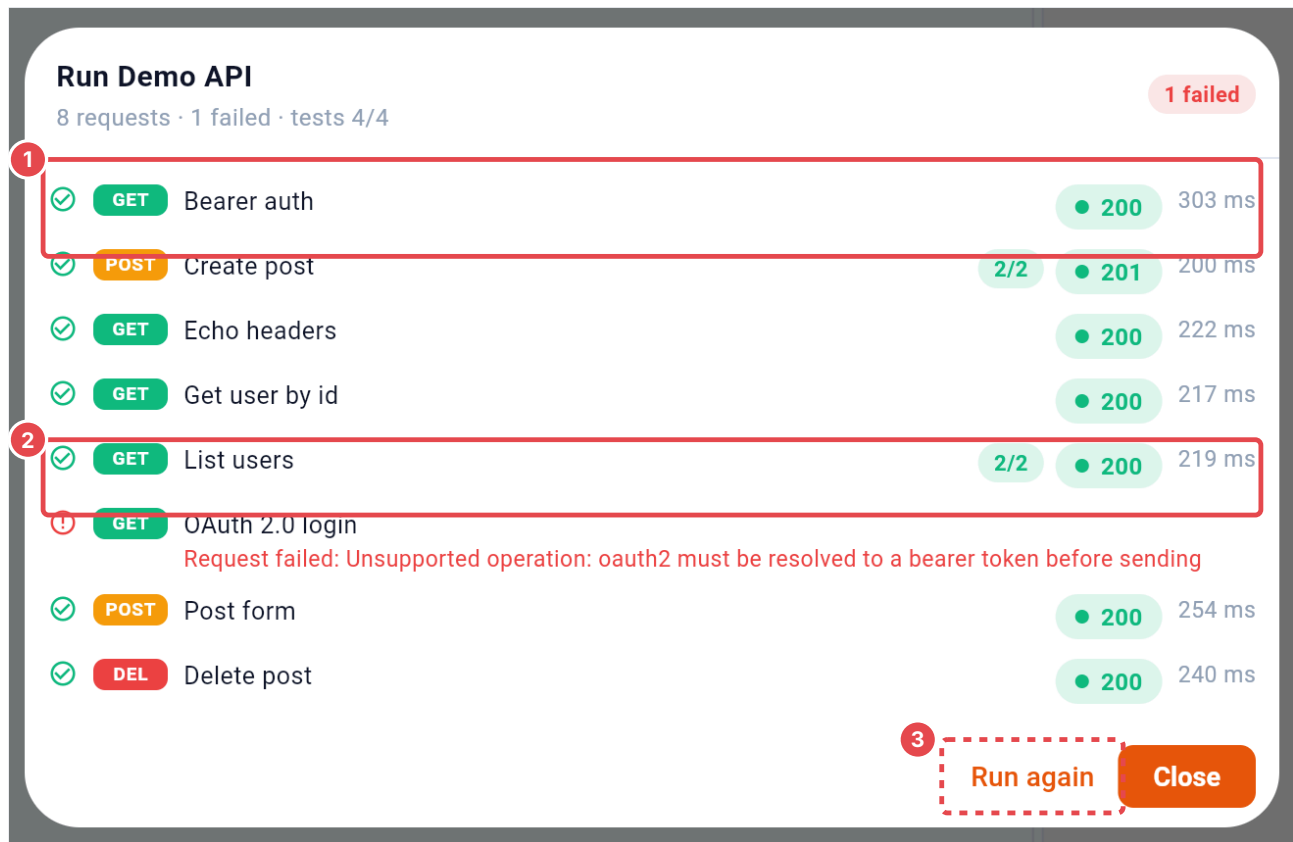
Auth and headers work as they do for HTTP, which is how a token reaches a socket that expects one at the handshake.

Note

One thing differs by build. A browser cannot put headers on a WebSocket handshake — the platform forbids it — so the web build names the headers it had to drop as the first line of the log and connects without them. The desktop builds use a socket that takes them, which is what makes an authenticated WS request work there.

12 Running a collection

A collection can be run end to end: every request in tree order, with its scripts and tests, in one pass. It is reached from the collection's menu or from the command palette.



- 1 A step that failed, with the reason in place
- 2 A step that passed: tests, status, time
- 3 Repeat the whole run

Figure 35: Eight requests, two of which failed — and the reason is in the row.

The heading counts the whole run: how many requests, how many failed, and the total tests passed. 1 is a step that failed, with the reason in place rather than hidden behind a click. 2 is one that passed, showing its tests, status and timing. 3 runs the lot again.

12.1 Reading a failure

The two failures above are worth keeping, because they are the two kinds you will meet:

Row says	Means
ResolveError: unresolved secrets: apiToken	The secret has no value on this machine. Nothing was sent. Set it in Variables & secrets .
oauth2 must be resolved to a bearer token before sending	The request uses OAuth and no token has been obtained yet. Authorize from its Auth tab.

A run does not stop at the first failure — every request is attempted, so one missing secret does not hide the state of the other seven.

Note

Scheduled runs are the same pipeline on the worker, described in Git sync and the worker 13. A collection that runs clean here is the one worth putting on a schedule.

13 Git sync and the worker

13.1 How sync works

Sync is commits in your repository. There is no sync server: Baton writes the same YAML files a human would, commits them and pushes. The transport differs by platform — a real git client on desktop, a host's REST API on a phone, either that or a paired worker in a browser — but the destination is always the repository you own.

GitHub sync

not connected sync straight to a GitHub repository — no git, no worker

Repository

GitHub GitLab Bitbucket

Branch
main

AUTHORIZE · HOST API

Only the host-API transport uses this token. A git checkout authenticates with its own credentials, set below.

OAuth App client id

from your GitHub OAuth App with device flow enabled

Sign in with GitHub

or paste a token

Using the host API — the browser build has no git engine — pair a worker to use its git bridge, or sync over the host API, so this build syncs through the host API — enter a repository and a token.

Through GitHub there is no local history and no commit signing — a git checkout has both. Either way the commits land in your repository.

Cancel Connect & sync

- 1 Connection state
- 2 GitHub, GitLab or Bitbucket
- 3 Authorize by device flow

Figure 36: Connecting a workspace to a repository.

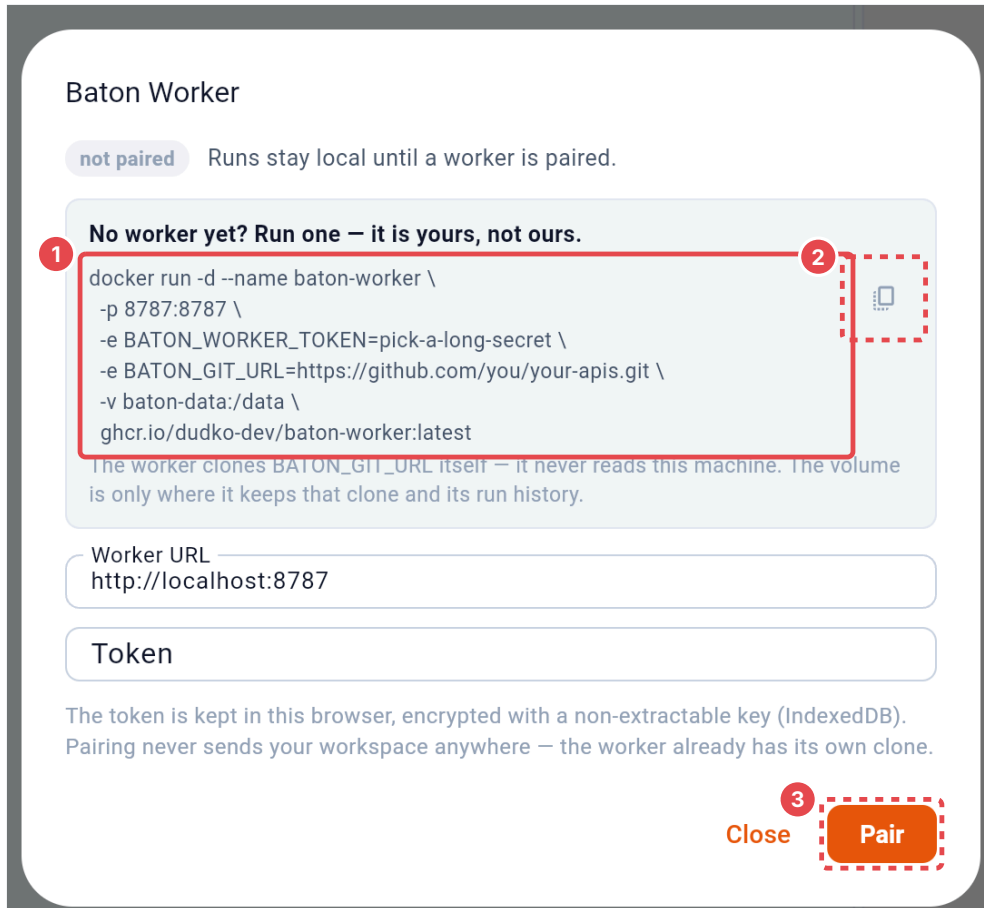
- 1 shows whether this workspace is connected.
- 2 picks the host — GitHub, GitLab or Bitbucket.
- 3 authorizes by device flow, so no password is typed into Baton.

Warning

Through a host's API there is no local commit history and no commit signing: the API creates the commit. For signed commits, or to keep history locally, use the desktop build or pair a worker.

Because the files are deterministic — fixed key order, two-space indent, stable quoting — a change to one request is a small diff, and two people editing different requests merge without conflict. When the same request changes on both sides, Baton resolves it field by field rather than line by line, and asks about anything it cannot settle.

13.2 The worker



- 1** Run your own worker with this
- 2** Copy the command
- 3** Pair once it is running

Figure 37: The worker is yours: a container you run.

A worker adds three things: scheduled runs, a git bridge for the browser build, and a relay for APIs a browser may not call directly. **1** is the command that starts one, **2** copies it, **3** pairs once it is up.

It is not a service of ours. The container clones your repository itself and never reads the machine Baton is running on; the pairing token is kept in the browser's encrypted storage.

13.3 In CI

The worker's runner is also a one-shot command, `baton-worker run`, for a CI job: it runs a checkout's collections once, request after request, and exits with a verdict. No server and no pairing token — the checkout, and secrets in environment variables, are all it needs.

```
BATON_SECRET_APITOKEN=... npx -y @dudko.dev/baton-worker run \
  --workspace . --environment staging \
  --collection collections/users-api \
  --reporter junit --output baton-report.xml
```

A request fails when it cannot be sent, answers with a 4xx or 5xx status, or fails a test. The exit code is 0 when every request passed and 1 otherwise, so the job fails with them. `--reporter junit` writes the report a CI shows in its test tab, json one for a script, and secrets are masked by value in all of them. The worker's container image runs the same command:

```
docker run --rm -v "$PWD:/data/workspace" -e BATON_SECRET_APITOKEN \
  ghcr.io/dudko-dev/baton-worker run --workspace /data/workspace --environment staging
```

13.4 Mock servers

Saved examples (see the chapter on sending) are enough to stand an API up without the API. A mock is a file in the workspace, `mocks/<name>.yaml`, that names a port and the collections to answer for. The worker starts it through its API and answers every request in those collections that has an example. The route is the request's address without its host, so `{{baseUrl}}/users/{{id}}` answers `GET /users/42`, and the answer is the example as saved — status, headers and body. The MCP server can start the same mock, so an agent writing a client can work against it before the API exists. A mock listens on `127.0.0.1` only.

13.5 Why a browser needs one

A page may only call an API that allows it by answering with CORS headers. Nothing a page can do changes that — not a Web Worker, not a Service Worker, not `mode: 'no-cors'`, which returns a response with no status, headers or body. So the browser build either calls APIs that opt in, or relays through a worker, which sends from a server where the rule does not apply.

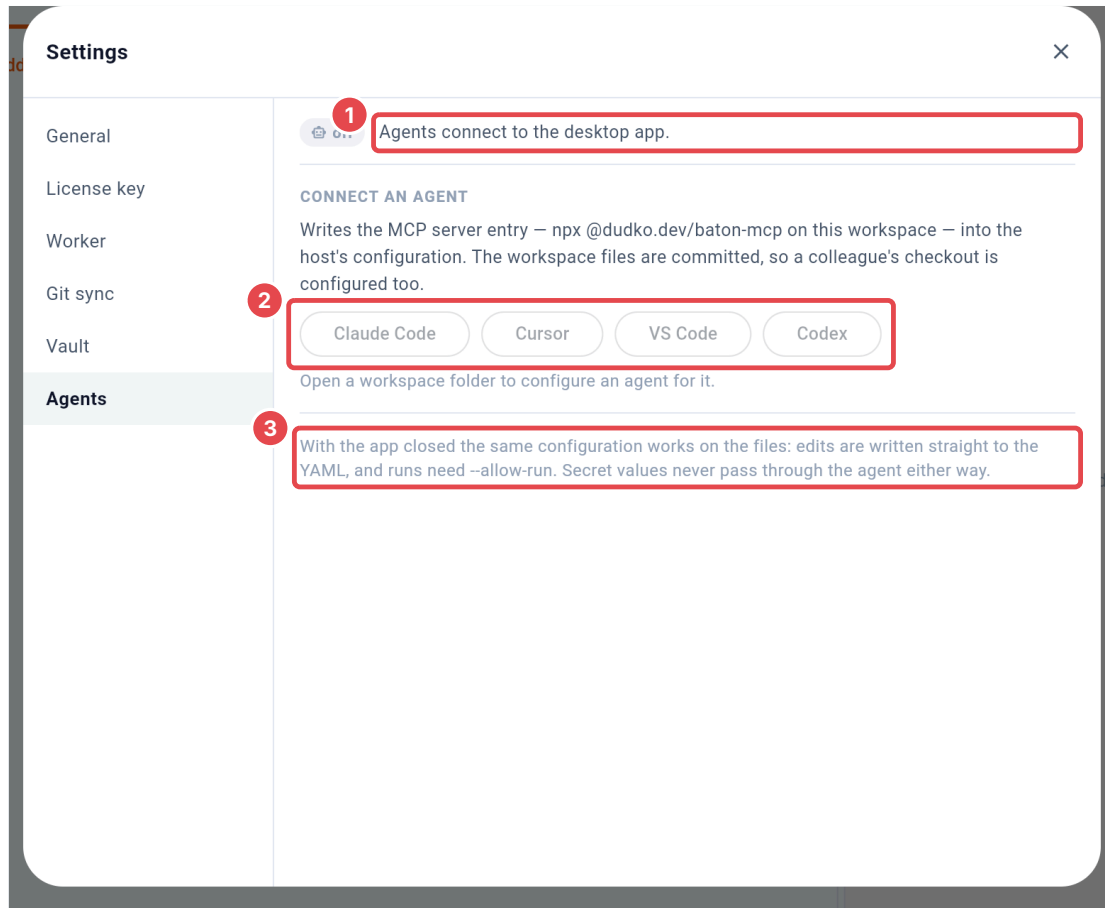
Note

When a request is refused this way Baton says so, and says the request may still have reached the API: a preflight is refused before anything is sent, but an ordinary cross-origin response is fetched, processed by the API, and only then withheld from the page. On a POST, treat it as possibly done.

14 AI agents

Baton ships an MCP server, `@dudko.dev/baton-mcp`, so an AI agent — Claude Code, Cursor, VS Code, Codex — can list, read, edit and run the requests in a workspace. It is a program on your own machine, started by the agent's host like any other MCP server; nothing about it passes through a service of ours.

14.1 Connecting an agent



- 1 Whether the app is listening for agents
- 2 Write the MCP entry for one host
- 3 Without the app: the same entry works on the files

Figure 38: Settings → Agents, here in the browser build, where there is no workspace folder to configure.

1 says whether the app is listening for agents. 2 writes the MCP entry for one host — `npx @dudko.dev/baton-mcp` on this workspace — into that host's configuration inside the workspace. The configuration is committed with the rest, so a colleague's checkout is set up too. The buttons need a workspace folder, which is why they are greyed out in a browser. 3 is the other way an agent works, below: with the app closed, the same entry still works on the files.

14.2 Through the app, or on the files

When the desktop app has the workspace open, the agent works **through the app**:

- **Edits are drafts.** An agent's change to a request lands on that request's tab as unsaved edits, exactly as if it had been typed, with a banner naming the agent. **Accept** saves it, **Accept all** saves every agent draft waiting, and **Discard** puts the file's content back.
- **Sends run in the app,** with the app's secrets, cookies and OAuth tokens and through the same steps as **Send**, so the response panel shows the result. The agent gets the status, headers, timing, tests and up to 16 KB of the body.
- **Production asks first.** A send from an agent into an environment marked production (see the chapter on variables), and every deletion, waits for you to confirm it in the app. A refusal reaches the agent as your decision.

The app listens on 127.0.0.1 only, with a token made for that launch, and answers nothing else — not another machine, not a web page. The switch in **Settings** → **Agents** turns it off.

With the app closed — and always in a browser or on a phone, neither of which can listen for an agent — the agent works **on the files**: its edits are written straight to the YAML, and it cannot send anything unless the server was started with `--allow-run`.

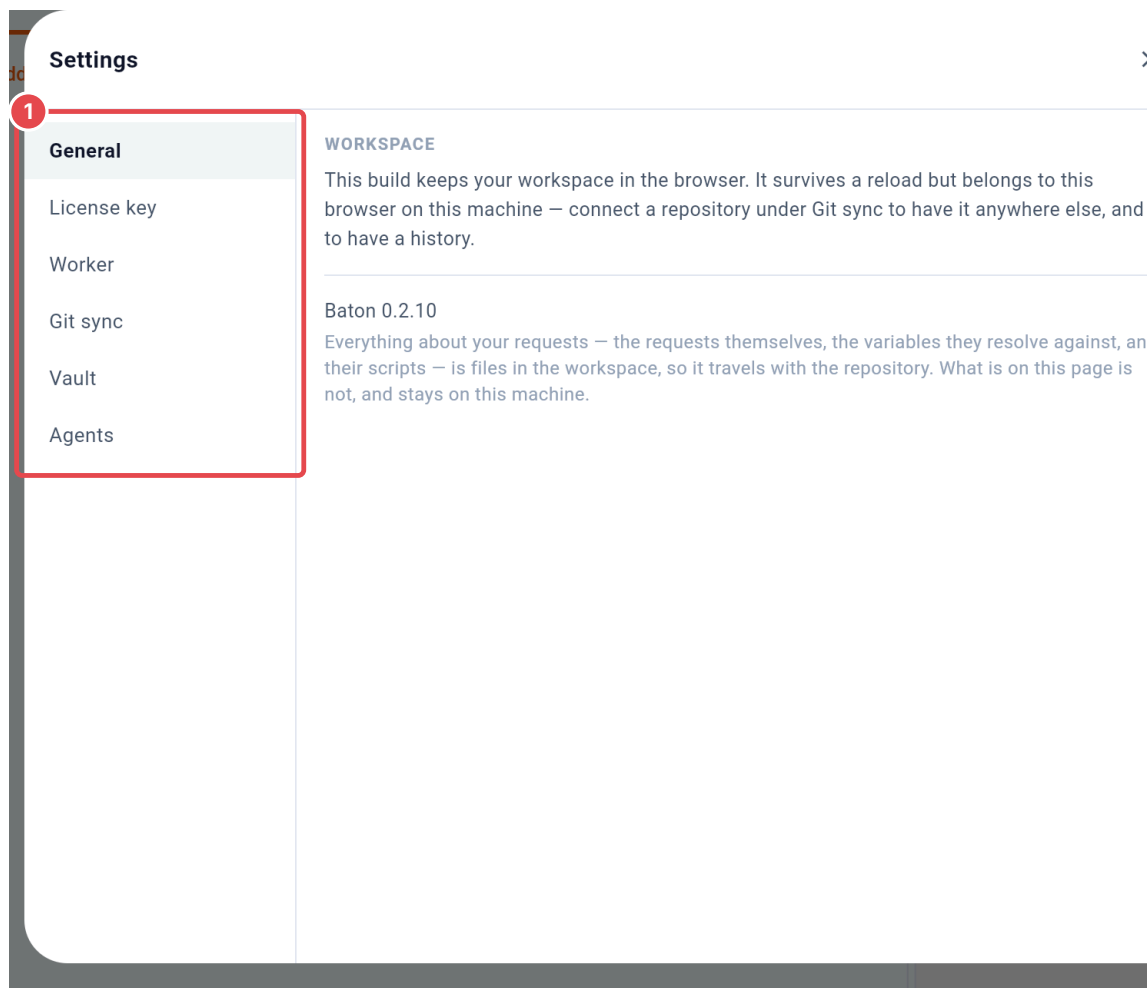
Warning

Secret values never reach the agent, either way. An auth field it writes must be a `{{name}}` reference; a literal credential is refused. A response it reads has every secret value the send used masked out, and the `Authorization`, `Cookie`, `Set-Cookie` and `X-API-Key` headers are replaced by name.

The agent does not commit. Whatever it changed is in the working tree, and committing it is yours to do — or to tell it to do — like any other change.

15 Settings, licensing and trouble

15.1 Settings

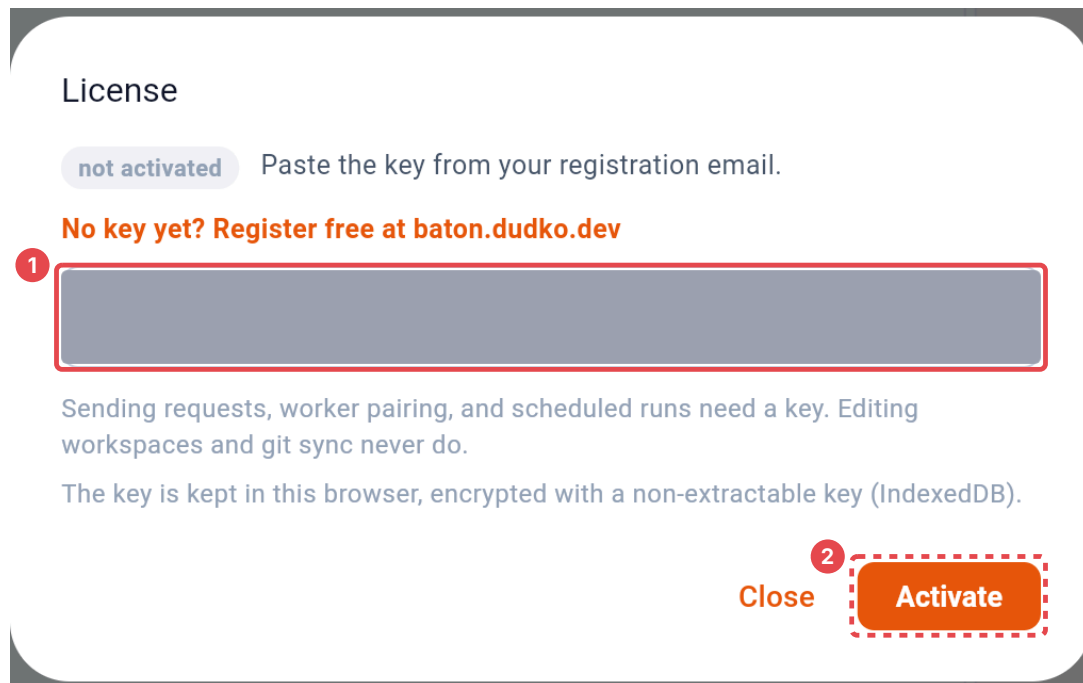


1 General, License key, Worker, Git sync, Vault, Agents

Figure 39: Six groups, and the workspace note that explains the browser build.

1 is the six groups: **General** for the workspace and updates, **License key**, **Worker** pairing, **Git sync**, **Vault** (Section 7) and **Agents** (Section 14). The workspace settings here — timeout, redirects, TLS validation, extra CA, client certificate, proxy — are written to `baton.yaml` and so travel with the repository.

15.2 Licensing



- 1 Paste the key from the registration email
- 2 Activate

Figure 40: The license dialog. The field is masked in this manual, not in the app.

Paste the key from the registration email into 1 and press 2. A free key comes from baton.dudko.dev and is valid for six months; registering again with the same address renews it. Verification is offline. The key is checked against a public key built into the app, so Baton never calls home to validate it and works air-gapped.

Needs a key	Never needs one
Sending a request	Creating and editing workspaces
Pairing a worker	Collections, requests, variables, secrets
Scheduled runs	Git sync — commit, push, pull, conflict resolution

Note

The App Store and Google Play builds are not gated at all: the store handles purchase, and a second check beside it would be both redundant and a review risk. The key dialog does not appear there.

15.3 When something does not work

What you see	What it means
ResolveError: unresolved secrets: ...	A secret has no value on this machine. Nothing was sent. Set it in Variables & secrets .
A {{reference}} is red in the URL	It does not resolve against the active environment — wrong environment, or a typo in the name.

What you see	What it means
the browser blocked this request	The API sends no CORS headers for this page. Pair a worker, or use the desktop app. The request may still have reached the API.
Sending needs a license key	No key on this machine. Editing and git sync keep working without one.
The cookie jar stays empty in a browser	A page may not read or set cookies for another site. Not a bug, and not something Baton can route around.
Request canceled.	You pressed the square on Send . The connection was dropped.

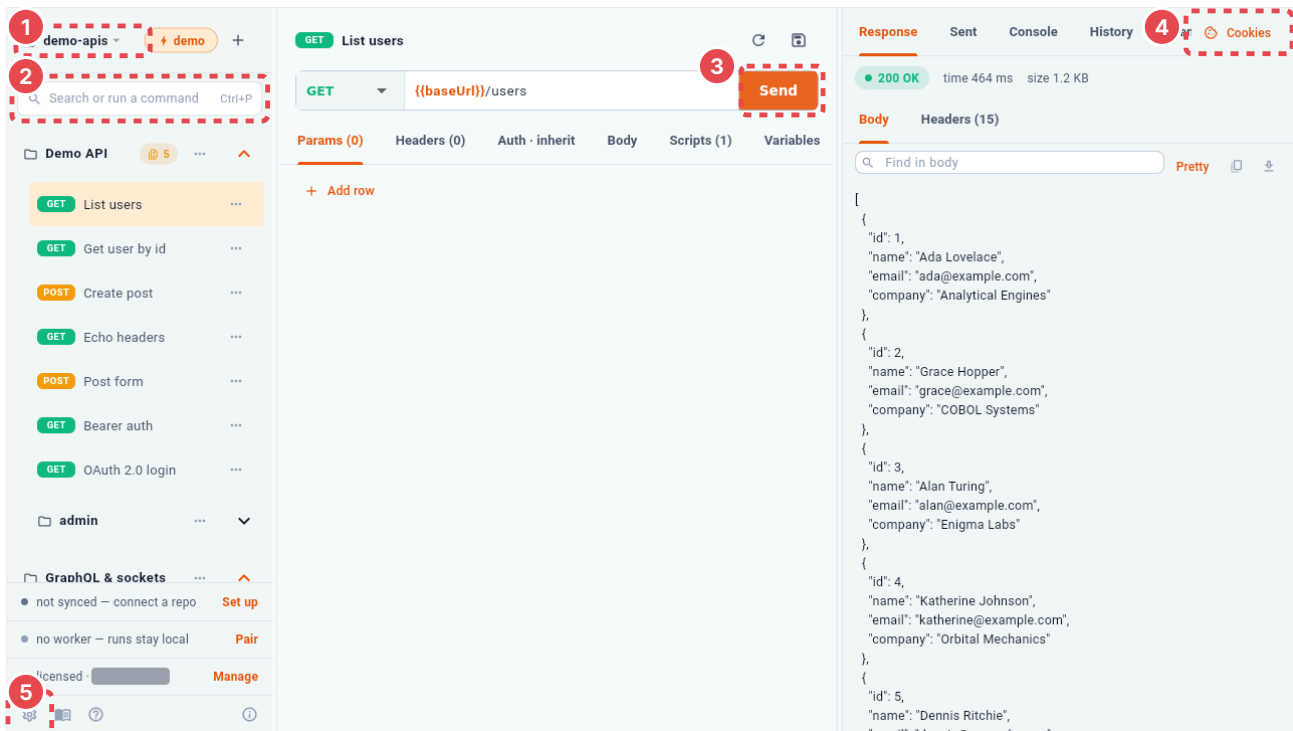
15.4 Keeping up to date

Desktop builds check for a new version against the site and offer to install it; the check carries no identifier and can be turned off in **Settings**. Store builds update through the store. The browser build is whatever the site is serving — reload to pick up a new one.

16 The desktop build

Everything so far was captured in a browser, because that is the build anyone can open without installing. The desktop builds — macOS, Windows, Linux — run the same Flutter code and look the same. The window below is the Linux build with the bundled demo, a key entered, and `{{baseUrl}}/users` answered in under half a second — nothing in the chapters before this one reads differently here.

Three things are genuinely different, and they are the reasons to install one.



- 1 Workspace — on the desktop, a folder you choose
- 2 Command palette
- 3 Send, and a square to stop it
- 4 The cookie jar — it works here
- 5 Settings, with a folder picker and updates

Figure 41: The Linux build: same demo, same request, a real answer.

16.1 A workspace is a real folder

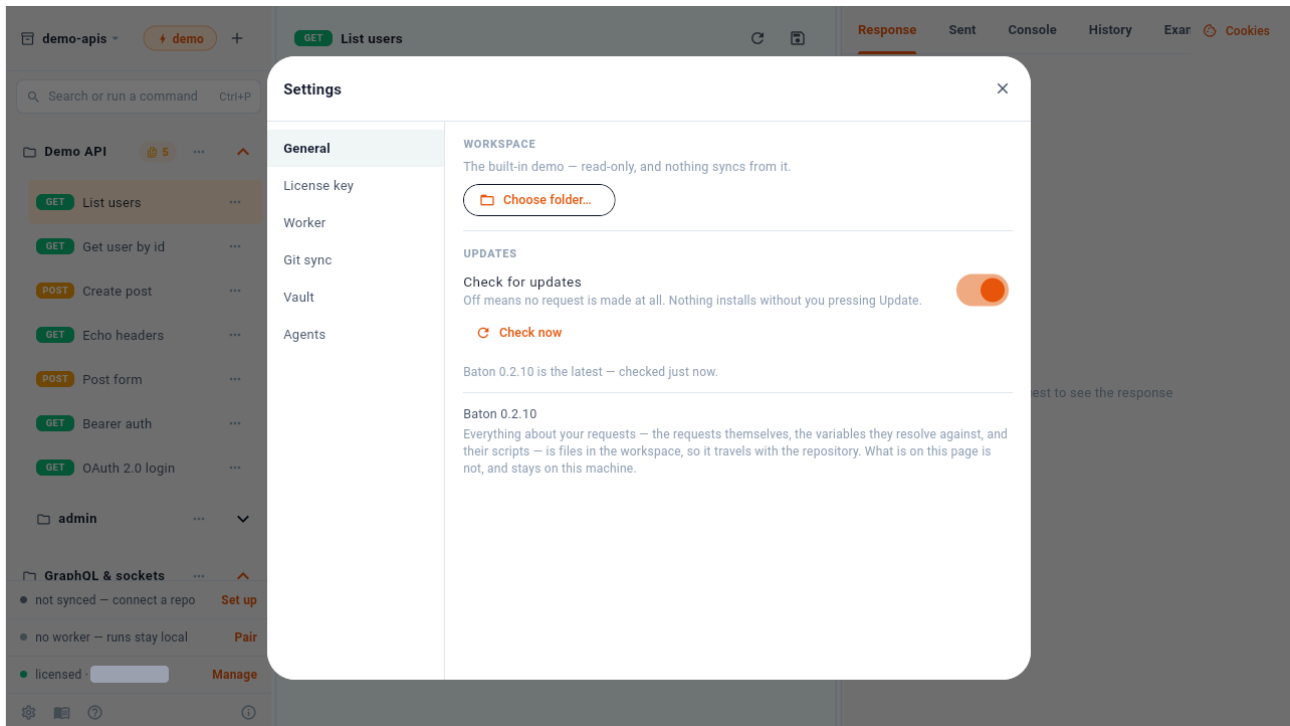


Figure 42: Settings on the desktop. Two things here do not exist in a browser.

Choose folder... points Baton at a directory, and that directory *is* the workspace: the YAML files are on disk where any editor, any script and git itself can reach them. A browser has no folder to point at, so it keeps a workspace in browser storage and reaches git through a host's API or a worker.

The **Updates** section is the other one. A desktop build checks the site for a new version and installs it in place when you press **Update** the check carries no identifier and the toggle turns it off entirely. Store builds update through the store, and the browser build is simply whatever the site is serving.

16.2 Cookies work

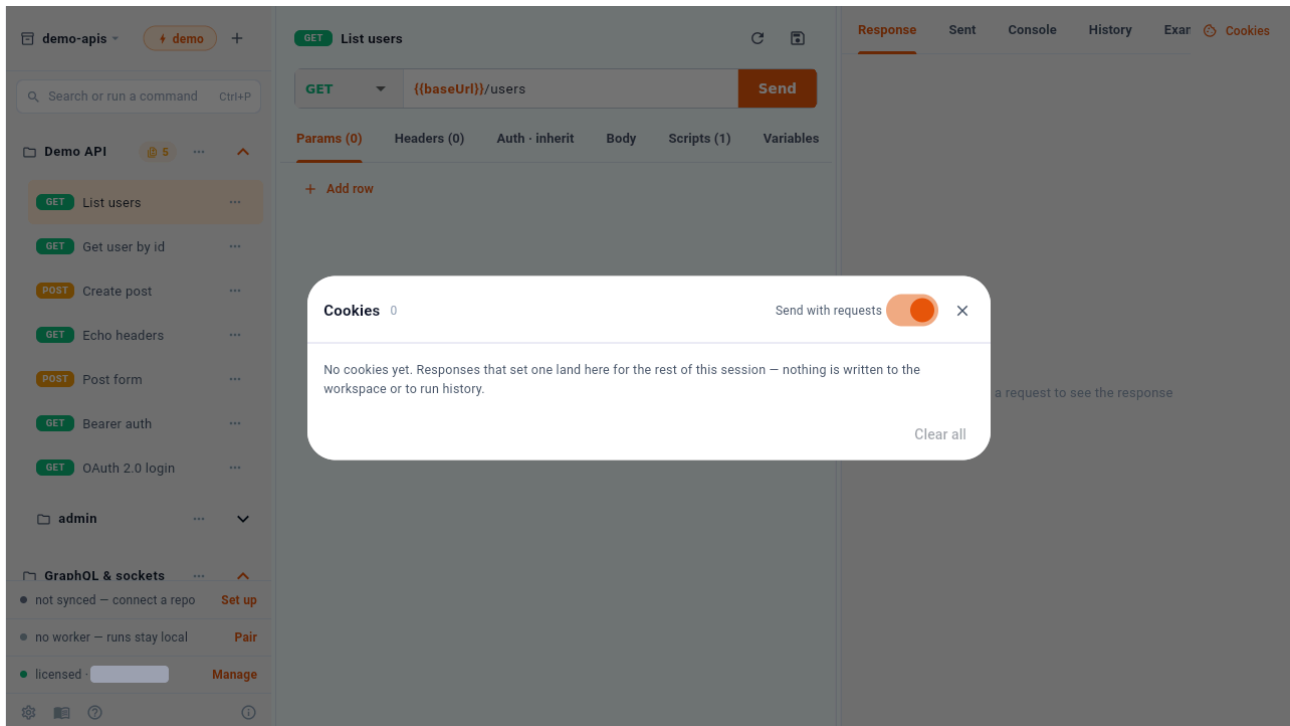


Figure 43: The jar on the desktop: empty, but real.

A response that sets a cookie lands here and is sent with later requests for the rest of the session — and nothing about it is written to the workspace or to run history, because a cookie is usually a session credential.

In a browser this dialog says instead that the build cannot use cookies at all. That is the platform's rule, not Baton's: a page may neither set a `Cookie` header nor read `Set-Cookie`, which is exactly what stops a web page forging another site's session.

16.3 Git without an intermediary

The desktop builds talk to git directly, so a commit is made and signed on your machine and the full history is local. Through a host's API — the browser's only option without a worker — the commit is created by the host, which means no local history and no commit signing. If signed commits matter, this is the build to use.

Note

The screenshots in this chapter are the Linux build of 0.2.10, run on a virtual display. Their frames are measured, not drawn by eye — but not in the same place as the rest of the manual. The desktop build registers on the AT-SPI accessibility bus and then exposes no tree through it, so there is nothing to measure inside it. The frames therefore come from the browser build at the same window size, which runs the same Flutter code and puts the widgets on the same coordinates: **Send**, found in both screenshots by its colour rather than by eye, occupies the same rows and ends on the same column. What differs between the two is content — this build lists the demo's requests in the order the demo declares them, the browser sorts them by name. Widgets that exist only on the desktop — the folder picker, the updates section — have no counterpart to measure, so those two figures carry no frames at all.