

Three tabs, one command.

How to build a command-line tool that reads the systems your business actually runs on, with the worked example I built against HubSpot, QuickBooks and Gmail, the seven prompts that produce it, and the part everyone gets stuck on: getting the tokens.

BY DAVE BROWN

Before a customer call I used to open three tabs. HubSpot for the deals, QuickBooks for what they owe, Gmail for what was actually said. Then I built a small thing that turns all three into one line of typing:

TERMINAL

```
recon "Acme Glass"
```

It is about 1,400 lines of TypeScript with three runtime dependencies. No database, no server, no web app, nothing deployed anywhere. The code was the quick part. Most of the work was authentication.

This guide is in two halves. The first half is that specific tool: what it does, how it is put together, and the decisions I would make again. The second half is the general case: what a CLI is, when it is the right shape for a problem, how to get OAuth tokens out of Google and HubSpot without losing a morning, and seven prompts you can hand to an AI agent to build your own version against whatever systems you happen to live in.

~1,400

Lines of TypeScript in the source. Tests add a few hundred more.

3

Runtime dependencies. Everything else is the standard library.

0

Servers, databases, containers, or things that can be down at 8am.

1

Word you have to remember to use it.

WHAT'S IN HERE

- 01 What a CLI actually is. Why this shape beats an app for a certain class of problem, and when it doesn't.
 - 02 The worked example. Three systems, one screen, and the design decisions worth copying.
 - 03 OAuth without the fog. What it is, why it's awkward for a CLI, and step-by-step token setup for Google and HubSpot.
 - 04 The seven prompts. Copy-ready, with the technical decisions already made so the agent doesn't improvise.
 - 05 Make it yours. The adaptation prompt, and a list of tools worth building for other jobs.
 - 06 What breaks first. The things I got wrong, and the two features I'd add next.
-

PART ONE · THE SHAPE

A CLI is a program you talk to by typing its name.

No window, no login screen, no URL. You type a word and some arguments, it does one job, it prints text, it exits. That constraint is the whole point.

If you have ever typed `git status` or `npm install`, you have used one. The category covers everything from one-line shell scripts to tools with thirty subcommands, but the good ones share a shape: a verb, some input, text out, done. They don't run in the background. They don't hold state between runs beyond a config file. They don't have users, in the plural. They have you.

That sounds like a limitation. It is mostly a gift. A tool with no interface has no interface to design, no frontend to build, no hosting bill, no session handling, no responsive breakpoints, no uptime. The entire surface area you have to get right is: what does it take in, and what does it print.

Three things can type your command.

This is the argument that actually matters in 2026, and it's the reason I keep building these instead of little web apps.

- **You can.** Before a call, in a terminal you already have open. No tab switching, no waiting for a dashboard to load the thing you needed one number from.
- **A script can.** Loop it over a list of accounts, pipe the output to a file, run it from cron on Monday morning. Anything that speaks text can drive it.
- **An AI agent can.** Give Claude Code a tool called recon and one line explaining it, and it can pull a customer brief without you handing it your OAuth tokens, your API docs, or any knowledge of HubSpot's object model. The CLI is the permission boundary and the vocabulary at the same time.

That third one changes the economics of building small tools. A year ago a script that only you could run was worth exactly what it saved you. Now every script you write is also a capability you can hand to an agent, which means the same afternoon of work compounds.

A tiny surface over messy systems is worth more than a big one. One word plus a company name beats a dashboard with forty filters.

When a CLI is the wrong answer.

I would rather name the trade-off than pretend this shape wins everywhere. Don't build a CLI when:

- **Non-technical people need to use it daily.** A terminal is a real barrier. If your sales team is the audience, you are building an internal web app or a Slack command, not this.
- **It needs to be always on.** Anything that reacts to a webhook, watches a queue, or sends the 8am digest without someone typing is a scheduled job or a service. A CLI can be the thing cron calls, but the schedule lives elsewhere.
- **The output is genuinely visual.** Charts, images, anything you want to point at in a meeting. Terminals are a text medium and fighting that is a waste of a week.
- **Multiple people need their own logins.** The moment you need per-user accounts and a token store that isn't "a file in my home directory," you have crossed into building a product.

Everything else is CLI-shaped: "I keep looking the same thing up in three places," "I need this data in a format I can paste," "I want an agent to be able to check this."

The five rules I hold myself to.

1. **Read-only until proven otherwise.** A tool that only reads can be run by anyone, scripted by anything, and handed to an agent without a second thought. The blast radius is zero. Add writes later, deliberately, behind a flag.
2. **One command, one job.** If you need a second job, that's a second subcommand, and if you need eight subcommands you probably needed a different tool.
3. **Never die halfway.** Three integrations means three things that can be down. Two working sections plus one honest error beats a stack trace every time.
4. **Errors name the fix.** Not "401 Unauthorized." "Gmail authorization has expired. Run: recon auth google."
5. **Text out, no surprises.** Colour and layout when a human is watching, plain text when the output is piped. The tool should behave when something else is reading it.

What recon does, and how it is put together.

One command finds the same company in three systems that have never heard of each other, and prints a single brief. Every line of it is a link back to the source.

RECON "ACME GLASS"

🔍 Looking up Acme Glass...

- ✓ HubSpot
- ✓ QuickBooks
- ✓ Gmail

🏢 ACME GLASS
🌐 acmeglass.com

🇺🇸 HUBSPOT

Acme Glass ↗

👤 Owner: Sarah Miller

📈 Open deals: 2

💰 Pipeline: \$84,000

👉 Enterprise Renewal ↗

\$64,000

Contract Sent

📅 Closes Sep 30

💰 QUICKBOOKS

Acme Glass ↗

💵 Balance: \$14,820

📄 Open invoices: 3

🔴 #10492 \$5,100 Overdue Aug 1 ↗

📧 RECENT EMAIL

📅 Sep 16, 2:14 PM

👤 Sarah Jones <sarah@acmeglass.com>

📧 Re: Expansion proposal ↗

"Looks good. I have one question about the renewal term before I send it to our CFO..."

Every ↗ is a real hyperlink. Click the invoice, land on that invoice in QuickBooks. Click the subject, land on that thread in Gmail. Nothing is written to any system. The tool has no capability to write at all.

What happens when you press enter.

1. HubSpot and QuickBooks are searched **in parallel**, by name. They only need the string you typed, so there is no reason to do them in sequence.
2. Gmail waits, because it searches far better with a domain than with a company name. It uses the domain HubSpot returned, falls back to an email address from QuickBooks, falls back to the quoted name. If the domain search comes back empty, it retries by name rather than showing you an empty inbox.
3. Each of the three returns a small tagged value: *data*, *no match*, or *failure*. The renderer looks at the tag and prints the right thing, including, for a failure, the exact command that reconnects that system.

The stack, and why each piece.

These are the decisions I would hand to an agent up front, which is exactly what the prompts in part four do. Unmade decisions are where agents improvise, and improvisation is where you get a Postgres dependency in a tool that reads three APIs.

DECISION	WHY
TypeScript, strict	API payloads are the messiest input you'll handle. Types at the boundary mean the renderer can trust what it's given. <code>strict: true</code> from line one. Retrofitting it is miserable.
Node 20+, ESM, NodeNext	Global fetch without a polyfill. <code>"type": "module"</code> with <code>module</code> and <code>moduleResolution</code> both set to <code>NodeNext</code> , which means relative imports end in <code>.js</code> even though the files are <code>.ts</code> . Odd the first time, correct forever after.
Commander	Arguments, subcommands, <code>--help</code> , <code>--version</code> . Thirty lines of wiring instead of a hand-rolled parser you'll get wrong.

DECISION	WHY
Plain fetch, no SDKs	Vendor SDKs are large, opinionated, and hide the request you're actually making. Three APIs, three files, every call visible. The one exception is Google's auth library, because token refresh is the one place I don't want a clever homegrown version.
dotenv	Config from <code>.env</code> in the project and <code>~/recon/.env</code> once installed globally. Real environment variables always win.
Vitest	Fast, TypeScript-native, and stubbing fetch is one line. Tests run against canned API payloads, never the live APIs.
tsc to dist/, bin + npm link	No bundler. <code>dist/index.js</code> starts with a shebang, <code>package.json</code> declares <code>"bin": { "recon": "dist/index.js" }</code> , and <code>npm link</code> puts the word on your PATH. <code>npm run dev</code> runs the TypeScript directly while you're building.
No database	The tool has no state worth keeping. Credentials go in <code>~/recon/</code> as JSON at mode 600. That is the entire persistence layer.

Four design choices worth stealing.

01

One normalized shape, defined before any integration exists.

`src/types/company.ts` describes the handful of fields the report prints (a company, a deal, a customer, an invoice, an email) and nothing else. It is deliberately not a faithful mirror of anybody's API. Each integration file is responsible for turning its own vendor's payload into that shape, so the renderer never learns that HubSpot calls it `dealname` and QuickBooks calls it `DocNumber`.

Write this file first. It is the contract that lets you add a fourth system later without touching the output code.

02

A tagged result type, so nothing can die halfway.

Every section is one of three things, and TypeScript makes you handle all three:

```
SRC/TYPES/COMPANY.TS
```

```
export type SectionResult<T> =  
  | { status: "ok"; data: T }  
  | { status: "empty"; message: string }  
  | { status: "error"; message: string; fix?: string };
```

A broken integration renders as its own section with the command that fixes it, while the other two print normally. This is the single highest-value pattern in the whole tool, because a customer brief that is two-thirds right at 8:55am is worth a great deal and a stack trace is worth nothing.

03

Errors you are happy to show a person.

One custom error class, `FriendlyError`, carrying a short sentence and an optional fix command. Anything that isn't a `FriendlyError` is a bug, and bugs only print in full behind `--debug`. The user-facing failure looks like this:

```
OUTPUT
```

```
 RECENT EMAIL
```

```
 Gmail authorization has expired.
```

```
Run:  
  recon auth google
```

04

Build links from the API, not from memory.

HubSpot record URLs need the portal id *and* the region-specific UI domain, both of which come back from `/account-info/v3/details`. My portal is on `na2`. A hardcoded `app.hubspot.com` would have produced a dead link on every single record, and I would have found out in front of someone.

Same instinct applies to QuickBooks, where sandbox and production are different hostnames for both the API and the UI, and to anything else with a region in it. If the vendor will tell you the base URL, ask them for it.

THE PATTERN

Fetch the account/tenant details once, derive every URL from it.

THE FALLBACK

If that call fails, drop the links and print the report anyway.

The layout, in full.

SRC/	
src/	
index.ts	CLI wiring (Commander), top-level error handling
env.ts	.env loading, config dir, redirect URI
errors.ts	FriendlyError + describeError
match.ts	deliberately dumb company-name matching
commands/company.ts	orchestrates the three lookups
commands/auth.ts	auth google quickbooks status
integrations/hubspot.ts	one file per API, returns normalized data
integrations/quickbooks.ts	
integrations/gmail.ts	
auth/google.ts	OAuth flow + token refresh
auth/quickbooks.ts	OAuth flow + realm id + token refresh
auth/local-server.ts	the throwaway localhost callback server
auth/token-store.ts	~/recon/*.json at mode 600
formatting/company-report.ts	renders the brief
formatting/style.ts	colour, emoji widths, OSC 8 hyperlinks
types/company.ts	the normalized shapes
tests/	22 tests, all against stubbed payloads

No repositories, no dependency injection, no plugin architecture. Each file is one you can read top to bottom in under a minute, which is the only architectural goal I had.

OAuth without the fog.

This is the part that stops people. Not the code. It's the twenty minutes in a console you've never opened, clicking through screens that assume you're shipping a product to strangers.

What OAuth is, in one paragraph.

OAuth is how you let a program act on your behalf **without giving it your password**. You go to the provider yourself, in your own browser, log in as you always do, and approve a specific, limited request: *this program may read my Gmail*. The provider hands your program a token. The token is scoped, it expires, and you can revoke it from a settings page without changing your password or affecting anything else.

The valet key analogy is old and still the best one. Your password is the key to everything. An OAuth token is the key that drives the car but doesn't open the trunk, stops working on Tuesday, and can be cancelled from your phone.

The vocabulary, once.

TERM	WHAT IT ACTUALLY IS
Client ID / secret	Your program's username and password with the provider, not yours. Created once in their developer console. The secret goes in <code>.env</code> and never in git.
Scope	The specific permission being asked for, as a string. <code>gmail.readonly</code> is a different thing from full Gmail access, and users see the difference on the consent screen.
Redirect URI	Where the provider sends the browser after you approve. It must be registered in advance and matched character for character. This is the bit that makes CLIs awkward.
Authorization code	A short-lived string that arrives on the redirect. Useless on its own: it has to be exchanged, server-side, using the client secret.
Access token	The thing you actually put in the <code>Authorization: Bearer</code> header. Typically good for about an hour.

TERM	WHAT IT ACTUALLY IS
Refresh token	The long-lived one that buys you new access tokens without another browser trip. This is the token that matters, and the one providers are stingiest about issuing.
State	A random string you send out and check on the way back. It proves the response belongs to the request you made. Sixteen random bytes, compared on return, rejected if it doesn't match.
Tenant id	Whatever the provider calls the account you just connected: <code>realmId</code> at Intuit, <code>portalId</code> at HubSpot. Store it alongside the tokens; most API calls need it.

Why it's awkward for a CLI specifically.

OAuth was designed for websites. A website has an address to redirect back to and a server to keep secrets on. A command-line tool has neither. Three problems follow, and each has a standard answer.

PROBLEM 01

There is nowhere to redirect to.

The answer is to briefly become a website. The tool starts an HTTP server on `127.0.0.1:8787`, opens your browser at the provider's consent URL, waits for the redirect to land on `/callback`, reads the query parameters, serves a small "you can close this tab" page, and shuts the server down. It exists for about fifteen seconds.

One function handles this for both providers, with a five-minute timeout, a state check, and a clear error if the port is already in use. It is roughly eighty lines and it is the same eighty lines in every CLI that does OAuth.

PROBLEM 02

There is nowhere safe to put the tokens.

Not the repo. Not a `.env` you might commit. Credentials go in `~/.recon/google.json` and `~/.recon/quickbooks.json`, with the directory at mode 700 and the files at mode 600, readable by you and nobody else on the machine. Two lines of `fs`, and it is the right amount of sophistication for a personal tool.

The client secret is a different thing from the tokens: it lives in `.env`, which is in `.gitignore`, with a committed `.env.example` listing the variable names and nothing else. If you are ever unsure whether you committed a secret, assume you did and rotate it.

PROBLEM 03

Everything expires, on different schedules.

Access tokens last about an hour, so the tool refreshes before every run rather than discovering the expiry mid-report. It refreshes a minute early, because a token that expires between your check and your request produces a 401 that looks like a bug.

Refresh tokens expire too, and the rules differ per provider. Google kills them after seven days while your app is in testing, Intuit rotates them on every exchange and expires them after about a hundred days. Whatever comes back from a refresh, write it down. The recovery path for all of it is the same: a `FriendlyError` that says authorization expired and names the reconnect command.

STEP BY STEP

Getting a Google token for Gmail.

Twenty minutes the first time, five minutes every time after. The console has been renamed and rearranged more than once, so trust the nouns below more than the exact menu names.

1. **Create a project.** In the Google Cloud console, use the project picker in the top bar and create a new one. Name it after your tool. Everything below happens inside it.
2. **Enable the Gmail API.** APIs & Services → Library → search "Gmail API" → Enable. Skip this and every call fails with a long message about the API not having been used in your project, which reads like an outage and isn't one.
3. **Configure the consent screen.** This lives under the Google Auth Platform section: branding, audience, data access, clients. If you have Google Workspace, choose **Internal**: no verification, no seven-day token expiry, and only people in your organisation can consent. If you're on a personal Gmail account, your only option is **External**, left in **Testing**.
4. **Add yourself as a test user.** On an External app in Testing, only listed test users can get through the consent screen. Add the Gmail address whose mail you want to read.

5. **Add the scope.** Under data access, add `https://www.googleapis.com/auth/gmail.readonly`. Read-only is genuinely all a brief needs, and it is the difference between a tool that can embarrass you and one that can't.
6. **Create the OAuth client.** Clients → Create client → application type **Web application** → under authorized redirect URIs add `http://localhost:8787/callback` exactly. Save, then copy the client ID and client secret.
7. **Put them in .env.** `GOOGLE_CLIENT_ID` and `GOOGLE_CLIENT_SECRET`. Confirm `.env` is in `.gitignore` before you paste anything.
8. **Run the auth command.** `recon auth google` opens your browser, you approve, the browser hits your localhost server, the tool exchanges the code and writes `~/.recon/google.json`. Verify with `recon auth status`.

TWO FLAGS THAT DECIDE WHETHER THIS WORKS AT ALL

The authorization URL needs `access_type=offline` to get a refresh token in the first place, and `prompt=consent` to get one again on a second authorization. Without the second flag, Google issues a refresh token on your first consent and silently withholds it on every subsequent one, which produces the single most common OAuth bug there is: it worked when you built it, and it stopped working the first time you re-authorized.

THE OTHER FOUR THINGS THAT WILL TRIP YOU

"Google hasn't verified this app." Expected on an unverified app. Advanced → Go to (unsafe). You are the developer and the user; you know exactly what you approved.

Seven-day refresh tokens. While an External app sits in Testing, Google expires refresh tokens after seven days and you get `invalid_grant`. For a personal tool, re-running `recon auth google` once a week is the honest answer. Publishing to production removes the limit, but `gmail.readonly` is a *restricted* scope, so publishing means Google's verification process and, for restricted scopes, a third-party security assessment. That is a real project, not a checkbox. Workspace users should just use an Internal app and skip the whole thing.

redirect_uri_mismatch. The registered URI and the one your code sends must be identical strings. `http://localhost:8787/callback` is not `http://localhost:8787/callback/`, and it is not `http://127.0.0.1:8787/callback`. Localhost is the one place plain `http` is allowed.

Adding a scope later doesn't upgrade an existing token. Change the scopes, and you have to re-consent to get a token that carries them.

A note on client type: Google recommends the **Desktop app** type for installed applications, which lets you use a loopback address on any available port. That's the better choice if you want the tool to grab whatever port is free. I pinned the port and registered it against a Web

application client because an exact, visible redirect URI is easier to debug. When it fails, the error tells you precisely which string didn't match. Either is defensible; pick one and be consistent.

STEP BY STEP

Getting a HubSpot token.

Much easier, because for a tool that only ever touches your own portal, HubSpot lets you skip OAuth entirely.

HubSpot has two ways in. A **public app** is the full OAuth dance, and it's what you build if other companies will install your integration into their portals. A **private app** is a token scoped to one portal (yours) that you generate in the settings UI and paste into your environment. No browser flow, no refresh, no expiry until you rotate it. For a personal CLI, private app every time.

1. **Check you're a Super Admin** on the portal. Private apps are an admin-level setting and the menu simply won't be there otherwise.
2. **Settings (the gear) → Integrations → Private apps → Create a private app.** Name it after your tool.
3. **On the Scopes tab, add exactly what you need.** For this brief:
`crm.objects.companies.read`, `crm.objects.deals.read`, and `crm.objects.owners.read`.
That last one is what turns an opaque owner id into "Sarah Miller." Search by name; the list is long and almost everything in it is a write scope you don't want.
4. **Create the app and copy the token.** It's behind a "show token" click and looks like `pat-na1-...` or `pat-na2-...` depending on your region. Treat it exactly like a password: it is a bearer token with no second factor in front of it.
5. **Paste it into .env** as `HUBSPOT_ACCESS_TOKEN` and run the tool. That's the whole setup.

Missing scopes fail loudly and specifically. HubSpot returns a message naming the scope it wanted. Add it to the private app's scope list and the existing token picks it up. You don't need to generate a new one.

Your portal may not live on `app.hubspot.com`. `/account-info/v3/details` returns both the portal id and the region-specific UI domain, and that endpoint needs the `oauth` scope, which private-app tokens carry. Build record URLs from what it returns. This is the bug I actually shipped and had to fix.

Batch your reads. HubSpot meters requests in short windows, so fetching twenty-five deals means one batch-read call, not twenty-five calls in a loop. Association endpoints give you the ids; the batch endpoint turns them into records.

A private app token only works in the portal that made it. The day you want to run this against a client's portal, you are building a public app with a real OAuth install flow: the same dance as Google, with an install URL in place of a consent screen.

QuickBooks, in one paragraph, because it's the third pattern.

Intuit is full OAuth like Google, plus one wrinkle: the callback carries a `realmId`, the id of the company file you picked during consent, and every API call needs it, so it gets stored alongside the tokens rather than derived later. Sandbox and production are separate hostnames for both the API and the UI, so the environment is config, not a constant. And Intuit *rotates* the refresh token on every exchange: whatever comes back from a refresh is what you must save, or you'll be re-authorizing far sooner than you expected. Between them, those three systems cover most of what you'll meet: a static token, a standard OAuth flow, and an OAuth flow with a tenant id attached.

The security floor.

- **Read-only scopes** unless you have a specific reason not to.
- **.env in .gitignore**, with a committed `.env.example` that names variables and holds no values.
- **Tokens at mode 600** in a mode 700 directory under your home folder, never in the project.
- **A state parameter** on every authorization request, checked on the way back.
- **Know the revoke path** for each provider before you need it: Google account permissions, HubSpot private app settings, Intuit connected apps.

- **Never print a token**, not even under `--debug`. Log the status code and the URL, not the header.

PART FOUR · THE PROMPTS

Seven prompts, in order, with the decisions already made.

Run them one at a time in an empty folder with Claude Code or the agent of your choice, reading what comes back between each. They are long and specific on purpose. Every decision you don't make is one the model will make for you, differently each time.

Before you start: make a directory, run `git init`, and open your agent there. Anything in square brackets and teal is yours to fill in. The examples name HubSpot, QuickBooks and Gmail because that's the tool I built; part five has the version where you swap in your own systems.

One habit that matters more than any individual prompt: **stop at the end of each one and actually read the code**. Seven prompts run back-to-back gives you something that compiles and that you don't understand. Seven prompts read one at a time gives you a tool you can fix at 8:55am.

PROMPT 01

The scaffold and the contract.

Sets every technical decision, defines the normalized shapes, and explicitly forbids writing any integration yet. Ending a prompt with "stop and show me" is what keeps an agent from building four files you didn't ask for.

COPY THIS

Build me the skeleton of a TypeScript CLI called `recon`.

WHAT IT WILL EVENTUALLY DO

One command, `recon "Acme Glass"`, looks the same company up in `HubSpot`, `QuickBooks Online` and `Gmail` and prints a single brief to the terminal. Read-only. It never writes to any of those systems.

STACK (these are decided, do not substitute anything)

- TypeScript 5.7+, "strict": true, target ES2022
- Node 20+, ESM ("type": "module"), with "module" and "moduleResolution" both set to "NodeNext", so relative imports end in .js even though files are .ts
- Commander for argument parsing and subcommands
- dotenv for config
- Plain global fetch for every HTTP call. No axios, no vendor SDKs. The only allowed exception is an official auth library where token refresh would otherwise be hand-rolled
- Vitest for tests
- tsc compiles src/ to dist/. package.json declares "bin": { "recon": "dist/index.js" } and src/index.ts starts with `#!/usr/bin/env node`. npm link is how it gets on my PATH
- No database, no server, no framework, no DI container, no plugin system, no class where a function will do

ARCHITECTURE RULES

- Each external system gets exactly one file in src/integrations/ that returns a normalized shape. The renderer must never see a raw API payload.
- src/types/company.ts defines those shapes and this generic:

```
export type SectionResult<T> =  
  | { status: "ok"; data: T }  
  | { status: "empty"; message: string }  
  | { status: "error"; message: string; fix?: string };
```
- src/errors.ts defines a FriendlyError carrying a short message and an optional `fix` string (the command that resolves it). Anything that is not a FriendlyError is a bug and only prints in full behind `--debug`.
- src/env.ts loads .env from the working directory and from `~/.recon/.env`, with real environment variables winning, and exports helpers for reading required and optional variables.

WHAT TO BUILD NOW

1. package.json, tsconfig.json, .gitignore (.env ignored, dist ignored),

- and `.env.example` listing variable names with no values.
- 2. `src/types/company.ts` with `SectionResult` plus the normalized shapes for a company with deals, a customer with invoices, and an email message. Only the fields the report will actually print.
- 3. `src/errors.ts`, `src/env.ts`.
- 4. `src/index.ts`: Commander wiring for the main command, a `--debug` flag, `--version`, help text, and a top-level catch that prints a `FriendlyError` as one red sentence plus its fix command.
- 5. A stub command that builds one of these briefs from hardcoded fake data and `console.log`s it as JSON.

DO NOT write any integration or any authentication yet. When you are done, show me the file tree and the contents of `src/types/company.ts`, and stop.

PROMPT 02

The easy integration: a static token.

Always start with the system that authenticates with a single environment variable. You get a real API call working before you've introduced OAuth, which means when OAuth breaks you know it's OAuth.

COPY THIS

Add the first integration: `HubSpot`, in `src/integrations/hubspot.ts`.

AUTH

A single environment variable, `HUBSPOT_ACCESS_TOKEN`, sent as ``Authorization: Bearer <token>``. No OAuth for this one. Export an ``isHubSpotConfigured()`` that just checks whether the variable is set.

WHAT IT DOES

Given a company name, return the normalized shape from `src/types/company.ts`, or undefined if nothing matches. Specifically:

1. Search companies by name and pick the best match.
2. Fetch that company's associated deals, batched into a single request, not one request per deal.
3. Resolve opaque ids into labels: the deal stage id into a stage name via the pipelines endpoint, the owner id into a person's name.
4. Sort deals open-first, then largest first.

MATCHING

Put name matching in its own file, `src/match.ts`, and keep it deliberately dumb: lowercase, strip punctuation, drop trailing company suffixes (inc, llc, ltd, co, corp, gmbh, plc), then score exact > starts-with > contains. Export a `bestMatch()` that takes candidates and a name accessor. Write unit tests for this file. It is pure and it is where subtle bugs live.

LINKS

Every record in the output needs a URL back to the source system. Fetch the account/tenant details from the API and build URLs from what it returns. `HubSpot's /account-info/v3/details` returns both the portal id and a region-specific UI domain; my portal is not on `app.hubspot.com`. If that call fails, return records without URLs rather than failing the lookup.

ERRORS

- Missing token, 401 or 403 → `FriendlyError` naming the variable to set.
- Any other non-2xx → `FriendlyError` with the status code.
- Never log or print the token itself, including under `--debug`.

Wire it into the main command so the real data replaces the stub, and show me one real run before you write anything else.

PROMPT 03

The OAuth flow and the callback server.

The prompt that saves the most time. It specifies the loopback server, the state check, the storage location, the file permissions, and the two Google flags: all the things an agent will otherwise get three-quarters right.

Now add OAuth 2.0 for [Google \(Gmail, read-only\)](#), in `src/auth/`.

THREE FILES

1. `src/auth/local-server.ts`, one reusable localhost callback server used by every provider I add later:
 - starts an http server on `127.0.0.1` at `OAUTH_PORT` (default `8787`)
 - resolves with the callback's query parameters, rejects on an ``error`` parameter, on a ``state`` mismatch, or after a 5 minute timeout
 - always closes the server, on every path
 - serves a small styled HTML page saying the tab can be closed
 - if the port is in use, a `FriendlyError` naming `OAUTH_PORT`
 - also exports `openBrowser(url)`, best-effort per platform, because the URL is always printed to the terminal as a fallback
2. `src/auth/token-store.ts`, with `saveTokens(name, data)` and `readTokens(name)`: JSON at `~/.recon/<name>.json`, directory created at mode `0o700`, files written at mode `0o600`. `readTokens` returns undefined on a missing or unparseable file rather than throwing.
3. `src/auth/google.ts`, the provider flow:
 - scopes: <https://www.googleapis.com/auth/gmail.readonly> only
 - redirect URI `http://localhost:8787/callback`, read from env so it can be overridden, and used identically in the auth URL and the token exchange
 - generate a random 16-byte hex ``state`` and verify it on return
 - the authorization URL must include `access_type=offline` AND `prompt=consent`, or a second authorization returns no refresh token
 - exchange the code for tokens, persist them via token-store
 - export an `accessToken()` that returns a live token, refreshing and re-saving when the stored one is within a minute of expiry
 - if there is no refresh token or the refresh fails, throw a `FriendlyError`: "[Google](#) authorization has expired." with fix "`recon auth google`"

THEN

Add an ``auth`` subcommand group: ``recon auth google`` runs the flow and confirms where the credentials were written; ``recon auth status`` lists every system and whether it is connected, with the exact command to fix each one that isn't.

Do not print, log, or write tokens anywhere except the token store. Do not add a second provider yet.

PROMPT 04

The second data source, and the search order.

The interesting logic in a multi-system tool isn't the API calls, it's deciding what to search with. This prompt makes that decision explicit instead of leaving it to whatever the model thinks is reasonable.

COPY THIS

Add `src/integrations/gmail.ts`, using the OAuth token from the previous step, and then fix the orchestration in the main command.

THE INTEGRATION

Search messages, take the most recent 5, and for each one fetch only the metadata headers (From, To, Subject, Date) plus the snippet, not the full body. Decode HTML entities in snippets. Build a URL to the thread.

Return the normalized shape. Map 401/403 to a `FriendlyError` with the reconnect command.

THE SEARCH ORDER (this is the part that matters)

Gmail searches badly on a company name and well on a domain, so:

1. Run the two name-only lookups (`HubSpot` and `QuickBooks`) in parallel with `Promise.all`. They need nothing but the string I typed.
2. Build the Gmail query from the best identifier available, in this order: the domain returned by `HubSpot`, then an email address from `QuickBooks`, then the quoted company name. Strip a leading `www`.
3. If a domain-based search returns zero results, retry once with the name-based query before reporting an empty section.

Put the query construction in its own exported pure function and unit test it: domain only, email only, both, neither.

ORCHESTRATION

Every lookup goes through one helper that turns success, no-match, and failure into a `SectionResult`. One integration throwing must never stop the others. I want two good sections and one honest error, never a stack trace. Print a one-line status per system as each resolves, then the report.

PROMPT 05

The output layer.

Terminal formatting is where agents produce the most generic work, so this prompt is the most prescriptive. The emoji-width rule and the Apple Terminal fallback are both things I only learned by shipping them broken.

COPY THIS

Write the rendering layer: `src/formatting/style.ts` and `src/formatting/company-report.ts`. It takes the brief and returns a string. It must be pure: no `console.log` inside the renderer.

`style.ts`

- Small ANSI helpers (bold, dim, red, green, cyan, plus a few 256-colour ones roughly matching each product's brand colour).
- Colour is disabled entirely when `process.stdout.isTTY` is false or `NO_COLOR` is set. Piping this tool must produce clean text.
- OSC 8 hyperlinks so a label itself is clickable, with a hard rule: Apple Terminal ignores OSC 8, so when `TERM_PROGRAM` is `Apple_Terminal`, or when output is piped, print the raw URL on its own line instead. Never show a label whose link silently does nothing.
- Emoji occupy two terminal columns. One `icon()` helper appends exactly two spaces after every glyph, and a `displayWidth()` counts wide characters as two so padded columns actually line up. Every icon in the report goes through `icon()`; nothing hand-spaces.
- A greedy word wrap with a max line count and an ellipsis, for long snippets.
- Currency and date formatters: whole dollars when the value is whole, "Sep 30" for dates, "Sep 16, 2:14 PM" for timestamps.

`the report`

- A boxed header with the company name and domain.
- One section per system, each with a heading, in a fixed order.
- For a status of "empty", one muted line. For "error", the message plus an indented "Run: <fix>" block. Never a stack trace.
- Overdue invoices print in red, decided by comparing the due date to today. Colour carries meaning here. It is not decoration.
- Every record (`company`, `deal`, `customer`, `invoice`, `email thread`) is a link back to its source system.

Then show me the output for a brief where one system is connected, one returns no match, and one has failed.

PROMPT 06

Tests that don't touch the network.

Twenty-odd tests is the right number for a tool this size. The one that earns its keep is the last one: proof that a dead integration still produces a full report.

COPY THIS

Write the test suite with Vitest. No test may make a real network call or read my real credentials.

APPROACH

Stub global fetch with a small table of [RegExp, payload] routes so each test declares the API responses it expects, and throw on any unmatched URL. An unexpected request should fail the test loudly, not return undefined. Mock the auth modules to hand back a fixed token.

COVER

1. match.ts: exact, prefix, substring, suffix-stripping, and no-match.
2. The [Gmail](#) query builder: domain only, email only, both, neither.
3. Each integration against a realistic canned payload, asserting the normalized shape, including the awkward cases: [a deal with no amount](#), [an invoice with a zero balance](#), [a message missing a Subject header](#).
4. The "not configured" path for each system: the right FriendlyError with the right fix command.
5. Argument parsing: no arguments prints help and exits non-zero; --debug is picked up.
6. The one that matters: a brief where one integration throws still renders a full report containing the other two sections and the reconnect command for the broken one.

Then add npm scripts: build (tsc), dev (tsx src/index.ts), test (vitest run), typecheck (tsc --noEmit). Make sure all four pass and paste the test output.

PROMPT 07

Ship it, and make it callable by an agent.

The --json flag is the step most people skip, and it's the one that turns a personal tool into something an agent or a script can use without parsing your box-drawing characters.

COPY THIS

Finish the tool.

1. A `--json` flag that prints the brief as JSON and nothing else: no status lines, no colour, no emoji, no spinner. Exit code 0 when at least one system returned data, 1 when every system failed. This is the mode a script or an AI agent will use, so the shape must be stable and documented.
2. A README with: what it does, a real sample of the output, `install` (`npm install`, `npm run build`, `npm link`), the environment variables in a table, the auth commands, and a short "layout" section listing each source file with a one-line description.
3. An AGENTS.md (and a CLAUDE.md that points at it) recording the rules an agent needs to not break this later: read-only, no database, no framework, one file per integration returning normalized shapes, FriendlyErrors with fix commands, stack traces only behind `--debug`.
4. Check `.gitignore` covers `.env`, `.env.*` except `.env.example`, `dist`, and `node_modules`. Confirm nothing in the working tree contains a real token. Grep for the token prefixes and for `"client_secret"`.
5. Tell me, in plain language, the three most likely ways this breaks in production and what the failure looks like from the terminal.

The same tool, pointed at whatever you happen to live in.

HubSpot, QuickBooks and Gmail are just the three tabs I had open. The shape (one word, several systems, one screen) transfers to almost any job where you look the same thing up in more than one place.

Three questions, answered before you prompt anything.

- 1. **What do you look up more than once a week, in more than one place?** Not what would be impressive to build. What you actually do. If nothing comes to mind, watch yourself for two days and count tabs.
- 2. **What is the one argument?** A company name, a ticket number, an order id, an email address. If you need three arguments to describe the job, the job isn't defined yet.
- 3. **What is on the one screen of output?** Write it out by hand first: literally type the terminal output you want into a text file. That fake output is the best specification you will ever hand an agent, and it takes four minutes.

IF YOU'RE A...	THE COMMAND	THE THREE SYSTEMS BEHIND IT
Support lead	who "jane@acme.com"	Helpdesk tickets, order history, shipping status: the three tabs every escalation opens.
Founder	monday	Bank balance, open invoices, pipeline changed since Friday, calendar for the week.
Recruiter	cand "Priya S"	ATS record, last email thread, interview feedback docs, calendar history.
Engineering manager	ship	Open PRs, failing builds, on-call incidents, issues that changed overnight.
Agency owner	client "Acme"	Time logged this month, invoice status, active project tasks, last client email.
Consultant	pipe	Proposals out, follow-ups overdue, contracts unsigned, money not yet collected.

The prompt below is prompt 01 with the specifics pulled out. Fill the teal parts, run it, then continue with prompts 02 through 07 substituting your own systems as you go. If one of your systems authenticates with a single API key, do that one second, before any OAuth, for the reason in part four.

THE ADAPTATION PROMPT

Build me the skeleton of a TypeScript CLI called `[one short word]`.

WHAT IT WILL EVENTUALLY DO

One command, `[tool] "[the one argument]"`, looks up `[the thing]` across `[system A]`, `[system B]` and `[system C]` and prints one brief to the terminal. Read-only. It never writes to any of those systems.

THE OUTPUT I WANT

`[Paste the terminal output you hand-wrote. Fake numbers are fine. This is the specification. Everything else serves it.]`

WHAT I ALREADY KNOW ABOUT THE APIS

`[System A]:` auth is `[an API key in an env var / OAuth 2.0 / OAuth plus a tenant id]`. The lookup I need is `[search by name, then fetch related records]`.
`[System B]:` ...
`[System C]:` ...
`[If you don't know, say so and ask the agent to check the current docs before writing any client code, and to tell you which endpoints and scopes it plans to use before it uses them.]`

STACK (these are decided, do not substitute anything)

- TypeScript 5.7+, "strict": true, target ES2022
- Node 20+, ESM ("type": "module"), "module" and "moduleResolution" both "NodeNext", so relative imports end in .js
- Commander for arguments, dotenv for config, plain global fetch for HTTP. No axios, no vendor SDKs except an official auth library where token refresh would otherwise be hand-rolled
- Vitest for tests. tsc builds src/ to dist/. package.json declares a "bin" entry; src/index.ts starts with `#!/usr/bin/env node`
- No database, no server, no framework, no DI container, no plugin system

ARCHITECTURE RULES

- One file per external system in src/integrations/, each returning a normalized shape defined in src/types/. The renderer never sees a raw payload.
- A `SectionResult<T>` union of ok / empty / error, so one dead system still produces a full report with a reconnect command in place of that section.
- A `FriendlyError` carrying a message and the command that fixes it. Raw errors only behind `--debug`.
- Credentials in `~/.[tool]/` at mode `0600`. Secrets in `.env`, which is gitignored.

WHAT TO BUILD NOW

The project files, the types, the errors, the env loader, the Commander wiring with `--debug`, and a stub command that prints the brief from hardcoded fake data. No integrations, no auth yet. Then show me the file tree and the types file, and stop.

PART SIX · AFTERWARDS

What breaks first, and what I'd add next.

Everything below is from running the thing, not from planning it. This is the list I'd want handed to me before I started.

BREAKS FIRST

Name matching. "Acme Glass Inc" in one system and "Acme Glass" in another is the normal case, not the edge case. My matcher strips suffixes and scores exact over prefix over substring, and it still guesses wrong occasionally. Keep it dumb, keep it in one tested file, and print the name you matched so a wrong guess is visible rather than silent.

BREAKS SECOND

Token expiry, on whatever schedule you weren't thinking about. Google's seven-day testing limit caught me on a Monday. The fix isn't clever refresh logic, it's making the failure legible: one sentence, one command, no stack trace, and the other two sections still printing.

BREAKS THIRD

Rate limits, the moment you loop. A tool run by hand makes a dozen calls. The same tool looped over four hundred accounts in a shell script hits the limit in under a minute. Batch reads where the API offers them, and if you plan to loop, add a delay flag before you need it.

WOULD ADD NEXT

A cache with a short life. Thirty seconds of caching on a per-company basis turns a four-hundred-company loop from cruel to routine, and costs nothing when you're typing the command by hand.

WOULD ADD NEXT

Writes, narrowly and behind a flag. Logging a call note back to the CRM is the obvious one. But I'd keep the read path and the write path visibly separate, and I would not let an agent call the writing version without me watching.

The honest summary: the code was the easy part and an agent can write most of it in an afternoon. The value was in three decisions (one normalized shape, one tagged result type, one error class that names its own fix) and in the twenty minutes of console clicking that nobody writes down. That's what this document is for.

I now type recon before calls instead of opening three tabs, and the same command sits in my agent's toolbox where it can answer questions I haven't thought to ask yet. My golden retriever remains unimpressed, which feels like the correct amount of impressed.

MORE LIKE THIS

**More guides like this one, for people trying to use AI
without embarrassing themselves.**

Weekly, plain-language breakdowns on Instagram.

[@davecto](#)