

How to build an MCP

Adding an MCP server on top of an API you already have.

Why build an MCP if you have an API?

Getting one number out of the API

1. Find the docs
2. Get a credential
3. Work out the endpoint and its parameters
4. Parse the JSON
5. Put it in a spreadsheet

Getting the same number out of an MCP

1. Paste a URL and sign in
2. Ask

... in the tool they already had open all day.

The API works fine. Most people still can't use it.

In practice, people don't learn to curl. They ask someone and wait three days, or they use a figure from a deck that is eighteen months old.

And it is an on-ramp rather than a walled garden — once someone finds what they want, the server can hand back the exact API call that produced it.

Preparing to build the MCP: audit what the API can actually do.

- **Read the source, not the docs.** Routes, controllers, serializers, enums. Documentation drifts; the code is what runs.
- **Ask three things of every endpoint.** Which parameters actually filter? What paginates? Is any total returned?
- **Every gap is work you just took on.** An endpoint that cannot filter server-side means paging and filtering locally, which drags in row caps and having to report how much you actually scanned.

That table is the design document. Note the upstream change that would remove each workaround next to the workaround — some turn out to be one-line fixes nobody had noticed.

WHERE THIS COMES FROM

Two real-world examples.

ESPEN

WHO AFRO neglected-tropical-disease data

5 tools. Hosted, with per-user sign-in through a corporate identity provider.

Rails app. Years old. Real users.

NTDeliver

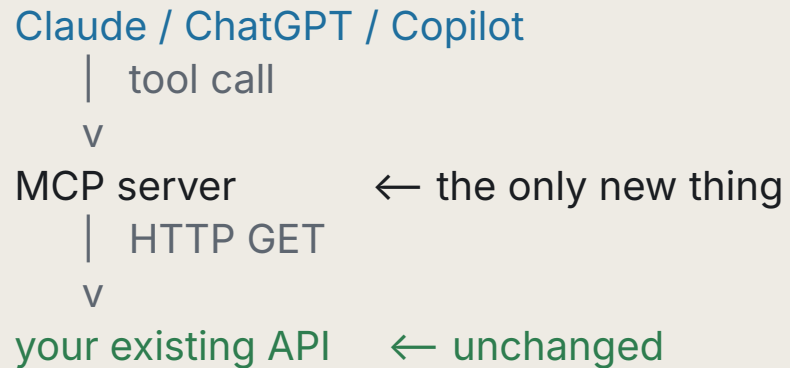
WHO NTD medicine delivery tracker

3 tools. Runs locally over stdio. Built in a day, a month after the first one.

Also Rails. Also already in production.

Neither project wrote a line of new data access. That is the normal case — you are almost never starting from scratch.

The MCP server sits between chat and your API.



- **No database connection.** It has none. Every answer comes back through the API you already expose.
- **No second permission model.** Your API keeps enforcing exactly what it enforced yesterday.
- **Read-only by construction.** One client, a single `get()` method, and a test that fails if a write ever appears.

The whole server is nine source files.

```
src/  
index.ts      entrypoint  
server.ts     create the server, register the tools  
config.ts     read env, fail fast on bad config  
logger.ts     structured logs, redaction  
<api>/client.ts the ONLY outbound HTTP path  
tools/shared.ts row caps, error translation, encoding  
tools/<tool>.ts one file per tool  
read_only.test.ts the guarantee, asserted end to end
```

Two of these carry the design. One client, so read-only is something you can actually enforce. One shared module, so every tool caps rows and translates errors the same way instead of drifting apart.

TypeScript on Node, the official MCP SDK, Zod for the tool schemas, Vitest. Nothing exotic — nobody's MCP server is hard because of the SDK.

Let Azure AD or Google handle the login.

Azure AD / Google --who are you?→ issues a token

MCP server

validates it

- publishes protected-resource metadata
- returns 401 + WWW-Authenticate
- issues nothing, stores nothing

Your server is a resource server and nothing more. That part is genuinely small — an afternoon.

What is not small: the identity provider will only accept an application identifier on a domain your organisation has verified. So you need a real hostname, so you need a DNS record from a team that has never heard of you. Start that conversation on day one — it outlasted our build by three weeks.

No identity provider? Add OAuth to Rails.

```
# Gemfile
gem 'devise'    # you almost certainly already have this
gem 'doorkeeper' # this is the new part
```

- **Devise already knows who the user is.** Doorkeeper turns that into /authorize and /token — it makes your app an OAuth2 provider.
- **The MCP server does not change.** It is still just validating a token. Only the issuer is different.
- **And it skips the DNS problem.** No external provider means no verified-domain requirement, which was the slowest part of option 1.
- **The trade:** you are now running an OAuth server attached to the app that holds your data. Worth it for a small internal tool; think harder for anything bigger.

What a new user does to get started.

1. Settings → Connectors → Add custom connector
2. Paste the URL
3. Sign in with the account you already use
4. Ask a question

No token to store. Nothing to install. No new tool to learn, because it is the tool they already had open.

Step 3 is why the last two slides mattered. A server behind one shared token cannot offer this — it stays a developer-only setup until per-user sign-in exists.

The translation layer goes stale when the API improves.

The upstream team fixed a dropped-filter bug and started returning a proper error. Our guardrail then began refusing those filters from a cached copy of their configuration — which they edit in production.

We nearly swapped a confidently wrong answer for a confidently wrong refusal. It encodes assumptions about someone else's system; you own it at their cadence.

It is your API's assumptions, written down.

Your API's contract, restated so a model does not have to guess at it — warnings, error translation, row caps and truncation flags, and filters it knows the endpoint cannot apply.

Without it, a 200 with zero rows reads as “this never happened”, and the model writes that down as fact.

99 lines

the Rails controller it wraps

Look up the endpoint. Check roles. Validate filters.
Select columns. Render.

390 lines

the MCP tool doing the wrapping

Almost none of the difference is MCP plumbing. The SDK part is about forty lines.

WHAT WE DID WITH ALL OF THIS

We packaged all of this as a reusable skill.

A SKILL IN CLAUDE-STANDARDS, OUR SHARED CONFIG REPO — IN REVIEW

`/mcp-server create`

`/mcp-server audit`

create

Starts by surveying what the upstream API can actually filter, page and count — read from its source, not its docs. That table is the design document: every gap is work you just took on.

audit

Nine checks, each one a defect we shipped or found. The trigger isn't a code review — it's an upstream API change.

Because most of them describe a server that was correct when written and stopped being correct when something upstream moved — with no line of our code changing.

TO SUM UP

An MCP server is just an adapter.

No database. No copy of anyone's data. No second permission model. It cannot write. The API it sits on is unchanged.

Almost none of the work is the MCP part. It's writing down everything your API assumes its callers already know.

The dividend: doing that turned up four real defects in the API itself, fixed within a day of being described clearly. Making a system legible to a model is a decent way of making it legible to its own maintainers.