A QUESTION WORTH ASKING FIRST

Do we need a frontend at all?

Before rebuilding a legacy interface,
I wanted to test a different idea:
connect the old system to an AI assistant
and see what is left to build.

A research note

darko.martic.net

WHERE THIS STARTED

The obvious plan is to rebuild the screens.

Every organisation has a legacy application that people depend on, that badly needs a refresh, and whose ageing technology makes that refresh hard to do. The instinct is to rebuild the interface on something modern.

Before committing to that, I wanted to check an assumption hiding inside it — that the answer to a bad interface is a better interface.

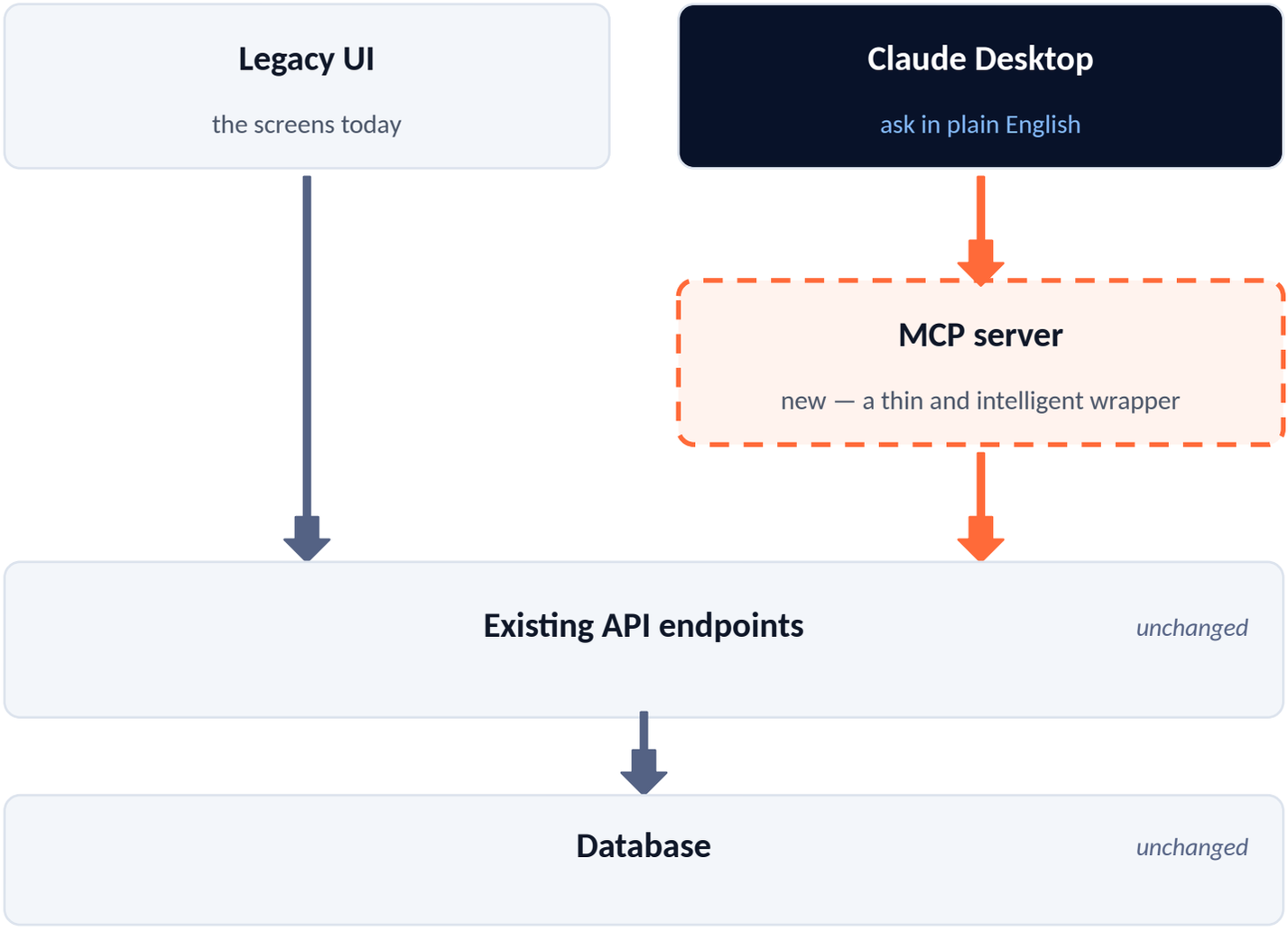
Because when you watch how these systems are actually used, much of it is looking things up. Which record is this. What changed. What is ending soon. Editing is a smaller share than it feels.

Those are questions. Not screens.

HOW THE PIECES FIT

An addition, not a rewrite.

It sits beside the app and calls the endpoints it already exposes.



Both paths end at the same endpoints — the wrapper only calls them.

Building an MCP server means building an AI layer. Any other system that needs this data (reporting, another service, a different tool) can connect through this standardized protocol. It makes the existing API future-proof.

STEP 0 — PREREQUISITES

What you need before starting

Less than you would expect. No new infrastructure, and nothing asked of another team.

- An account with your normal access to the application
- Its API endpoints — documented, or recovered by watching the app run
- Python 3.10 or newer
- Claude Desktop, or another MCP-capable client

What you do NOT need

No backend changes. No new endpoints. No elevated permissions.
The server reads exactly what your own login can already see.

STEP 1 — THE SETUP

A few minutes of plumbing

In Claude Desktop: Settings → Developer → Edit Config. Point it at your script, restart, done.

```
{
  "mcpServers": {
    "legacy-app": {
      "command": "/path/.venv/bin/python",
      "args": ["-m", "legacy_mcp.server"],
      "env": {
        "BASE_URL": "https://app/context",
        "SESSION":  "<session cookie>"
      }
    }
  }
}
```

A pasted session cookie is fine for research and unacceptable for anything permanent — a real deployment needs a service account.

STEP 2 — WHERE TO BEGIN

Start with one endpoint. Not the whole API.

The temptation is to map everything first. Resist it. Pick a single endpoint and get one honest answer out of it — that teaches you more than a week of planning.

- Read-only, so nothing can go wrong while you learn
- Returns a list people already ask questions about
- Small enough that you can eyeball the result and spot errors

```
# start here. one call, one shape.
rows = await api.get("records/list")
print(len(rows), rows[0].keys())
```

Once one endpoint works end to end, the rest is repetition.

STEP 3 — TURNING IT INTO A TOOL

A tool is a function plus a plain explanation

You write an ordinary function. Above it, in plain English, you explain what it does and when to use it. That text is what the AI assistant reads to decide whether to call it.

Write it for a new colleague, not for a compiler. Vague wording there is the most common reason a tool never gets used.

```
@mcp.tool()
def search_records(owner=None, status=None,
                   changed_since=None):
    """Search records by owner, status or
    when they last changed.

    The application shows one record at a
    time, so questions spanning many records
    cannot be answered there.
    """
    return store.query(owner, status, changed_since)
```

Most tools look like this. The hard part is the wording, not the code.

THE ADVANTAGE I DID NOT EXPECT

A screen can only answer questions someone predicted.

Every screen in every application is a guess about what people would want to know. The guesses that were right became features. The rest became exports to a spreadsheet.

A handful of tools does not need the guess. The AI assistant combines them to answer things nobody designed for.

Questions nobody built a screen for

“Which of these has no owner, and what are they worth?”
“What changed after the review, and by whom?”
“What data are we storing that nobody ever looks at?”

WHAT IT ACTUALLY LOOKS LIKE

No interface. Just a question.

Claude

What changed last week, and who changed it?

`search_records`

Twenty-three changes across nine records. Three people:

Who	Changes	Area
A. Smith	11 changes	Scheduling
R. Chen	8 changes	Budget
B. Jones	4 changes	Status

Most of it was scheduling. Two records were changed and then changed back the following day.

Illustrative example.

A question like that usually means exporting to a spreadsheet, because no screen was built to answer it.

SO — DO WE NEED A UI?

It turns out to be a split, not a replacement.

Where asking wins

Anything occasional · questions spanning many records · combining things no screen puts together · one-off analysis · checking whether something is set up correctly · finding data nobody maintains · questions you only ask once · the long tail nobody would fund a feature for

Where screens win

Work done the same way every day · entering and editing data · anything needing layout to make sense · tasks where being wrong is costly and you want the whole form in front of you

THIS IS NOT A CONCLUSION

The research continues.

I am not finished, and I would not trust anyone who said they were. What I have is a cheap way to test an expensive assumption before committing to it.

Next: what changes when the AI assistant can write and not only read. Where the trust boundary sits. Whether any of this holds up with people who did not build it.

If you have a legacy application that is badly documented, this is worth trying before you scope the rewrite. You may be scoping the wrong thing.

Following along? I will post what I find.

darko.martic.net