I mapped an undocumented legacy system, built a working prototype, and wrote the spec to rebuild it.

In one day. With AI.

Here is exactly how — and what I would do next.

A FRONTEND-FIRST PROPOSAL

Should we modernise only the frontend of the legacy app?

I think so — as a first step, at least.

darko.martic.net · September 2026

MY ASSUMPTION

The backend is fine. It is the surface that has aged.

The business process behind this app has not changed in years. The same operations are supported, and every endpoint the interface calls is still needed — that is clear from the documentation produced during this research. The engine is doing its job. The part people touch is the part that dated.

A quick win for users

The daily frustration lives in the interface, not the engine. Replacing it delivers a visibly better experience in a short time.

The ageing technology goes

The frontend is where the technical debt is most acute and hardest to hire for. It can be retired on its own.

A foundation built to last

Documentation and a way of working designed for change — so upgrades are routine rather than archaeology.

And the backend?

Its stack is current, so security patching is routine upkeep. An obsolete backend stack is usually what forces a rebuild — not the case here. Out of scope of this research.

THE GOAL

Three questions

The app people use every day runs on an ageing frontend. Replacing it had been discussed and never scoped, because nobody could say what it would take. So I set out to answer that.

1

Is it possible without touching the backend?

Rebuilding the backend as well would be a different order of investment.

2

What would it actually involve?

No documentation existed. Scope was unknowable, so any estimate was a guess.

3

Would it genuinely be better for users?

A rewrite that reproduces today's experience is money spent for nothing.

I scoped the work to answer these three cheaply, before asking anyone to commit to a build.

THE WORK

What I did, in three steps

1

Mapped the system

Recovered how the existing app talks to its backend — every screen, every data call — by observing the running system rather than reading source.

I had no real expectations going in. I did not know whether this would produce anything usable.

2

Built a working prototype

A new interface over the real data, used to test whether the findings held and whether the result was actually better.

Because the alternative is pretending you know what you need before you have seen it.

3

Wrote it up as a specification

Turned everything learned into a documented “contract”, a plan and a task breakdown that a team could pick up cold.

Not meant to be perfect documentation — meant to scope everything, so a BA or PO can review and fill in detail.

I did all of it in a single day.

No backend changes. No development team engagement. No time from anyone else.

THE METHOD

I did it with AI, in conversation

Not a tool I ran, and not automation. It was a working session — I described what needed answering and steered as the answers arrived.

1

I described the goal, not the steps

I started with the question rather than a method. Claude proposed how to investigate, and explained what each finding meant.

2

It inspected the running app

Working inside the browser, Claude observed what the app actually does — then checked every conclusion against the live system.

3

Build, notice, correct

The prototype was built and reworked in the same session. Whenever I spotted something wrong, it revealed something new about the old system.

4

Write it down while learning it

The documentation came from what had been observed and verified — not reconstructed afterwards from memory.

The judgement stayed mine.

Claude did the volume — reading, probing, building, writing up.

FINDINGS

What I found — the three questions, answered

Yes — it is possible

The backend can stay exactly as it is. This is a frontend project, not a platform project.

The scope is now known

The system is mapped in full: which screens are genuinely complex, and which are the same pattern repeated.

It can be meaningfully better

The prototype already does things the current app cannot — e.g. instant search across every field.

I also surfaced several issues in the existing system that need an owner whether or not anything gets rebuilt.

ASSETS

What exists now, even if this goes no further

Documentation

A full written “contract” for a system that had none. Useful to anyone touching it, for any reason.

A working prototype

Runs on real data. Can be put in front of users right away to test whether the direction is right.

A build-ready plan

Specification, architecture and a task breakdown — but it needs review, and detail added where needed.

The downside case is not zero

If nothing gets rebuilt, the documentation still removes a real risk: the current system was understood by very few people and written down by none. That is fixed either way.

ROADMAP

If this goes ahead

1. **Validate concept**
   Confirm the approach holds at real scale

2. **Foundation**
   Build the shared groundwork every screen uses

3. **Build screens**
   Viewing and editing, screen by screen

4. **QA**
   Test against the spec, not against memory

5. **Migrate in parallel**
   New app runs alongside the old one

First, though — read the documentation

What Claude produced in a day is an impressive foundation, but it was recovered by observation, not handed over. Someone who knows the domain needs to read it end to end and correct the details. That is reading and editing, not writing from scratch.

HOW FUTURE UPGRADES AND MAINTENANCE WOULD WORK

By “future” I mean the ordinary work that follows a build — the changes, fixes and upgrades that never stop.

1

A change starts in the documentation, not the code

I describe what should change and why, and update the spec first. The code follows the document.

2

Claude Code does the work against the documented change

My job becomes describing the change clearly and reviewing what comes back.

3

The documentation is the single source of truth

Nobody reconstructs context from a trail of Jira tickets, chat threads and memory.

4

Onboarding is reading, not reverse-engineering

Someone new starts from documents instead of a decade-old codebase.

5

The backend can modernise later without a second rewrite

One translation layer means a later backend change lands in one place.

The point is not just this rebuild. It is also that the one after it will be far cheaper.

ONE LAST THOUGHT

If I got all of this done in a single day, imagine what someone who actually knows what they are doing could do with it.

☺️

IF THIS SOUNDS FAMILIAR

Most companies have one of these. An app everyone depends on, that nobody has documented.

If you are staring at one, the approach in these slides is worth a day of your time. Happy to talk through how it worked.

Found this useful? Repost it for someone who needs it.

darko.martic.net