

# Overview

This is the architecture documentation of the software project done for MSE TSM\_SoftwEng 2023.

If you are reading this as a PDF, note that there is a much prettier, interactive version embedded in the application itself.

# Architecture Decision Records

We document our design decisions as [architecture decision records](#). We follow the simple [format proposed by Michael Nygard](#).

# Rust everywhere

## Status

Accepted.

## Context

The requirements for this project include:

- collecting data on a microcontroller
- storing and serving the data with a web service
- displaying the data in an SPA

These are vastly different domains, for which different programming languages are well suited. For an SPA, JavaScript is completely dominant. For the embedded part, C has the necessary performance and ecosystem. In the web service space, competition is lively. Java, Python and Go and more are common choices.

The only language that's a rock-solid option for every one of these domains is Rust. It has the performance and control of C with memory-safety on top and a fast-paced embedded ecosystem. Its expressive type system makes it a breeze to build reliable and correct web services. Finally, its best-in-class support for compiling to webassembly and GUI-libraries utilizing state-of-the-art rendering patterns make it well-suited for developing an SPA.

## Decision

The entire software system will be written in Rust.

## Consequences

Developing both the SPA and the microcontroller will be more challenging, as Rust doesn't have as mature of an ecosystem as JavaScript and C respectively.

It will be significantly easier to share code between the three software components. Developing cross-cutting features will require less context-switching. Testing and CI/CD will be significantly easier to setup, as there is only one toolchain to worry about.

# HTTP routing with axum

## Status

Accepted.

## Context

The Rust standard library does not include an HTTP router. A common task like that is best solved by using a library. There are several suitable ones available for Rust. [Actix](#) and [axum](#) are the most popular at the time of writing. For our simple requirements, neither is better or worse. There is existing experience with axum in the development team.

## Decision

HTTP routing in the web service will be handled using axum.

## Consequences

HTTP routing will not have to be done by hand.

# JSON with serde

## Status

Accepted.

## Context

Web services communicate using well-specified formats, typically JSON. The best way to do de-/serialization of JSON or another format in Rust is to use [serde](#). Since our tech stack is pure Rust, there is no need for OpenAPI or similar contract enforcement mechanisms. Shared definitions of the data types sent across process boundaries ensure compatibility. JSON, being a text-based format, is not the most efficient, but easily inspectable. For more efficient communication, binary formats are better suited.

## Decision

The web service communicates via JSON, powered by serde.

## Consequences

De-/serialization of inter-process messages will not have to be done by hand. A potential future migration to a more performant format than JSON is trivial.

# SPA with Leptos

## Status

Accepted.

## Context

Modern single-page-applications are written using component-based UI libraries. This is because updating the DOM globally in an imperative manner quickly becomes unmanageable. In the JavaScript ecosystem, [React](#) is the most popular, but there are other ones using different reactivity systems. Rust also has several such UI libraries, despite the ecosystem being much younger. The Rust-equivalent for React is [Yew](#), but another notable library is [Leptos](#). It has comparable popularity to Yew and uses a rendering system most similar to JavaScript's [Solid](#). For our requirement of implementing the flux/redux/store pattern, both are equally suitable. Both provide an API for making a reactive singleton object accessible to the entire component tree. There is existing experience with Leptos in the development team.

## Decision

The SPA will be developed using the Leptos UI library.

## Consequences

Developing the SPA will be a smooth experience.





# Architecture Diagrams

## C4 Container Diagram



User

## SoftwEng Project

Web App [Rust, Leptos SPA, Tailwind]

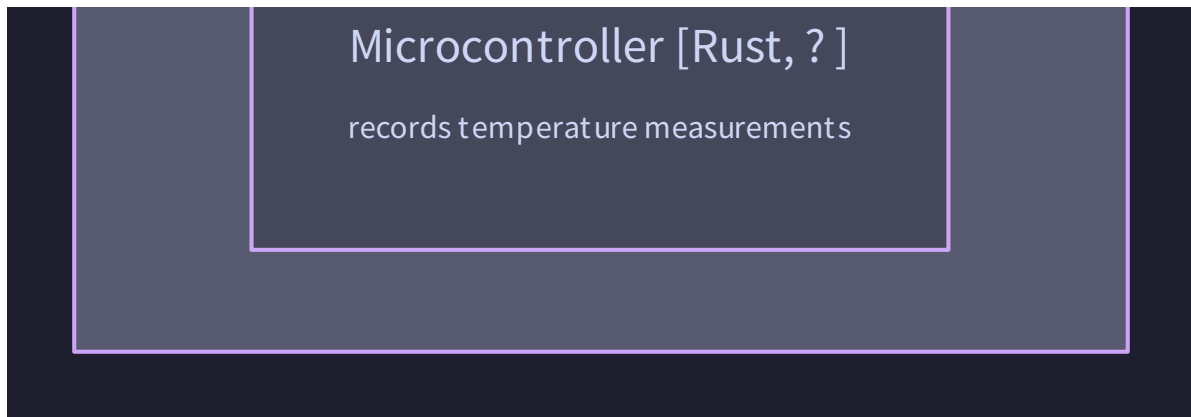
displays temperature measurements

*fetches data*  
*[REST API]*

Web Service [Rust, Axum]

stores temperature measurements

*sends measurements*  
*[?]*



## 4+1 model

TODO

