Sascha Grunert
Jul 8 · 9 min read

A web application completely in Rust

My latest software architectural experiment is to write a complete real-world web application in Rust with as less as boilerplate as possible. Within this post I want to share my findings with you to answer the question on how much web Rust actually is.

The related project to this post can be found on GitHub. I put both, the client-side frontend and the server-side backend, into one repository for maintainability. This means Cargo needs to compile a frontend and a backend binary of the whole application with different dependencies.

> *Please be aware that the project is currently fastly architectural evolving and everey related source code of this article can be found within the `rev1` branch. You can read the second part of this blog series here.*

The Application itself is a simple authentication demonstration. It allows you to login with a chosen username and password (must be the same) and fails when they are not equal. After the successful authentication a JSON Web Token (JWT) is stored on both the client and server side. Storing the token on the server side is usually not needed but I've done that for demonstration purposes. It could be used to track how much users are actually logged in for example. The whole application can be configured via a single Config.toml, for example to set the database credentials or server host and port.

```toml
[server]
ip = "127.0.0.1"
port = "30080"
tls = false

[log]
actix_web = "debug"
webapp = "trace"

[postgres]
host = "127.0.0.1"
username = "username"
password = "password"
database = "database"
```

Config.toml hosted with ❤ by GitHub view raw

The default Config.toml for the webapp

The Frontend — Client Side

I decided to use yew for the client side of the application. Yew is a modern Rust framework inspired by Elm, Angular and ReactJS for creating multi-threaded frontend apps with WebAssembly (Wasm). The project is under highly active development and there are not that many stable releases yet.

The tool cargo-web is a direct dependency of yew, which makes cross compilation to Wasm straight forward. There are actually three major Wasm targets available within the Rust compiler:

- *asmjs-unknown-emscripten*—using asm.js via Emscripten
- *wasm32-unknown-emscripten*—using WebAssembly via Emscripten
- *wasm32-unknown-unknown*—using WebAssembly with Rust's native WebAssembly backend

I decided to use the last one which requires a nightly Rust compiler, but demonstrates Rust native Wasm possiblities as its best.

> *WebAssembly is currently one of the hottest 🔥 topics when it comes to Rust. There is a lots of ongoing work in relation to cross compiling Rust to Wasm and integrating it in the nodejs (npm packaging) world. I decided to go the direct way, without any JavaScript dependencies.*

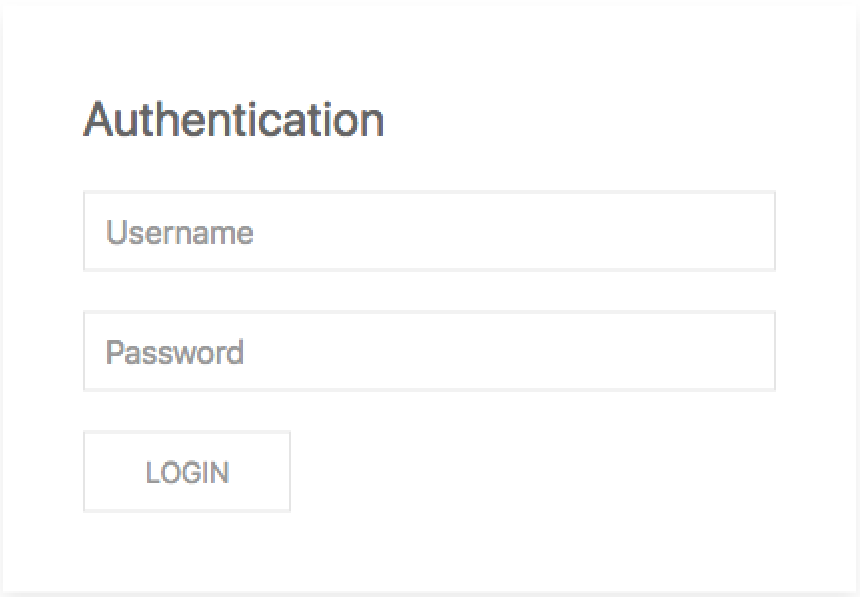
When starting the frontend of the web application (in my project via `make frontend`), cargo-web cross compiles the application to Wasm and packages it together with some static content. Then cargo-web starts a local web server which serves the application for development purposes.

```
> make frontend
   Compiling webapp v0.3.0 (file:///home/sascha/webapp.rs)
    Finished release [optimized] target(s) in 11.86s
    Garbage collecting "app.wasm"...
    Processing "app.wasm"...
    Finished processing of "app.wasm"!

If you need to serve any extra files put them in the 'static' directory
in the root of your crate; they will be served alongside your application.
You can also put a 'static' directory in your 'src' directory.

Your application is being served at '/app.js'. It will be automatically
rebuilt if you make any changes in your code.

You can access the web server at `http://0.0.0.0:8000`.
```

frontend_build.shell hosted with ❤ by **GitHub** **view raw**

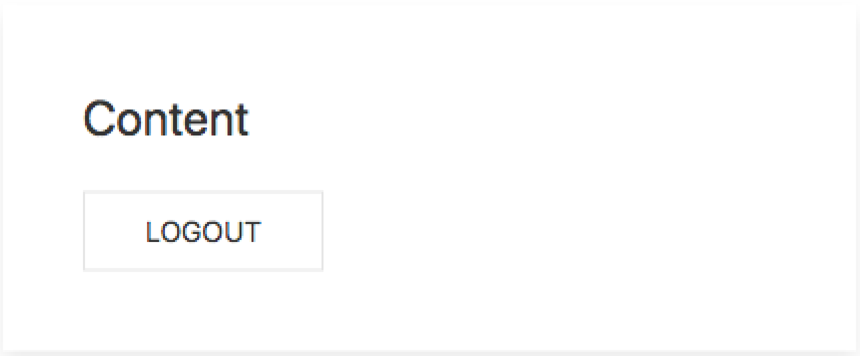
Yew has some great features, like the reusable component architecture, which made it easy to split my application into three major components:

- *RootComponent*: Directly mounted on the `<body>` tag of the website and decides which child component should be loaded next. If a JWT is found on initial entering of the page, it tries to renew the token with a backend communication. If this fails, it routes to the *LoginComponent*.
- *LoginComponent*: A child of the *RootComponent* and contains the login form field. It also communicates with the backend for a basic username and password authentication and saves the JWT within a cookie on successful authentication. Routes to the *ContentComponent* on successful authentication.



The LoginComponent

- *ContentComponent*: Another child of the *RootComponent* and contains the main page content (for now only a header and logout button). It can be reached via the *RootComponent* (if a valid session token is already available) or via the *LoginComponent* (on successful authentication). This component communicates with the backend when the user pushed the logout button.



The ContentComponent

- *RouterComponent*: Holds all possible routes between the components which hold content. Also contains an initial “loading” state and an “error” state of the application. Is directly attached to the *RootComponent*.

Services are one of the next key concepts of yew. They allow reusing the same logic between components like logging facades or cookie handling. Services are stateless between components and will be created on component initialization. Beside services yew contains the concepts of Agents. They can be used for sharing data between components and provide an overall application state, like needed for a routing agent. To accomplish the routing for the demonstration application between all components a custom routing agent and service was implemented. Yew actually ships no stand-alone router, but their examples contain a reference implementation which supports all kinds of URL modifications.

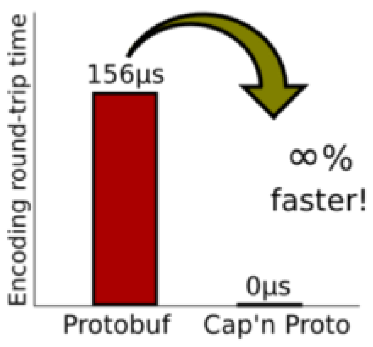
> *Amazingly, yew uses the Web Workers API to spawn agents in separate threads and uses a local scheduler attached to a thread for concurrent tasks. This enables high concurrency applications within the browser written in Rust.*

Every component implements its own `Renderable` trait which enables us to include HTML directly within the rust source via the `html!{}` macro, this is pretty great and for sure checked by the compilers internal borrow checker!

```
impl Renderable<LoginComponent> for LoginComponent {
    fn view(&self) -> Html<Self> {
        html! {
            <div class="uk-card uk-card-default uk-card-body uk-width-1-3@s uk-positior
                <form onsubmit="return false",>
                    <fieldset class="uk-fieldset",>
                        <legend class="uk-legend",>{"Authentication"}</legend>
                        <div class="uk-margin",>
                            <input class="uk-input",
                                   placeholder="Username",
                                   value=&self.username,
                                   oninput=|e| Message::UpdateUsername(e.value), />
                        </div>
                        <div class="uk-margin",>
                            <input class="uk-input",
                                   type="password",
                                   placeholder="Password",
                                   value=&self.password,
                                   oninput=|e| Message::UpdatePassword(e.value), />
                        </div>
                        <button class="uk-button uk-button-default",
                                type="submit",
                                disabled=self.button_disabled,
                                onclick=|_| Message::LoginRequest,>{"Login"}</button>
                        <span class="uk-margin-small-left uk-text-warning uk-text-right
                            {&self.error}
                        </span>
                    </fieldset>
                </form>
            </div>
        }
    }
}
```

The `Renderable` implementation for the LoginComponent

The communication from the frontend to the backend and vice versa is implemented via a WebSocket connection for every client. The WebSocket has the benefit that it is usable for binary messages and the server is able to push notifications to the client too if needed. Yew already ships a WebSocket service, but I decided to create a custom version for the demonstration application mainly reasoned by the lazy initialized connection directly within the service. If the WebSocket service would be created during component initialization I would had to track multiple socket connections.



I decided to use the binary protocol Cap'n Proto as application data communication layer (instead of something like JSON, MessagePack or CBOR) for speed and compactness reasons. One little side note worth to mention is that I did not use the interface RPC Protocol of Cap'n Proto, because the Rust implementation does not compile for WebAssembly (because of tokio-rs' unix dependencies). This makes it a little bit harder to distinguish between the right request and response types, but a cleanly structured API could solve the problem here:

```
@0x998efb67a0d7453f;

struct Request {
    union {
        login :union {
            credentials :group {
                username @0 :Text;
                password @1 :Text;
            }
            token @2 :Text;
        }
        logout @3 :Text; # The session token
    }
}

struct Response {
    union {
        login :union {
            token @0 :Text;
            error @1 :Text;
        }
        logout: union {
            success @2 :Void;
            error @3 :Text;
        }
    }
}
```

Cap'n Proto protocol definition for the application

You can see that we have two different login request variants here: One for the *LoginComponent* (credential request with username and password) and another for the *RootComponent* (already available token renewal request). All needed protocol related implementations are packed within a protocol service, which makes it easily reusable within the whole frontend.

UIkit — A lightweight and modular front-end framework for developing fast and powerful web interfaces.

The user interface of the frontend is powered by UIkit, where version `3.0.0` will be released in the near future. A custom build.rs script automatically downloads all needed UIkit dependencies and compiles the overall stylesheet. This means custom styles can be inserted within a single style.scss file and are application wide applied. Neat!

Frontend testing

Testing is a little bit a problem in my opionion: The separate services can be tested pretty easily, but yew does not provide a convenient way how to test single components or agents yet. Integration and end-to-end testing of the frontend is also not possible within plain Rust for now. It could be possible to use projects like Cypress or Protractor but this would include too much JavaScript/TypeScript boilerplate so I skipped this option.

> *But hey, maybe this is a good starting point for a new project: An end-to-end testing framework written in Rust! What do you think?*

The Backend — Server Side

My chosen framework for the backend is actix-web: A small, pragmatic, and extremely fast Rust actor framework. It supports all needed technologies like WebSockets, TLS and HTTP/2.0. Actix-web supports different handlers and resources, but within the demonstration application are just two main routes used:

- `/ws`: The main websocket communication resource
- `/`: The main application handler which routes to the statically deployed frontend application

By default, actix-web spawns as much workers as CPU cores are available on the local machine. This means a possible application state has to be shared safely between all threads, but this is really no problem with Rusts fearless concurrency patterns. Nevertheless, the overall backend should be stateless, because it could be deployed with multiple replicas in parallel within an cloud based (like Kubernetes) environment. So the applications state should be outside of the backend within a separate Docker container instance for example.

I decided to use a PostgreSQL database as main data storage. Why? Because the awesome Diesel project already supports PostgreSQL and provides a safe, extensible Object-relational mapping (ORM) and query builder for it. This is pretty great since actix-web already supports Diesel. In result, a custom idiomatic Rust domain specific language can be used to create, read, update or delete (CRUD) the sessions within the database like this:

```rust
impl Handler<UpdateSession> for DatabaseExecutor {
    type Result = Result<Session, Error>;

    fn handle(&mut self, msg: UpdateSession, _: &mut Self::Context) -> Self::Result {
        // Update the session
        debug!("Updating session: {}", msg.old_id);
        update(sessions.filter(id.eq(&msg.old_id)))
            .set(id.eq(&msg.new_id))
            .get_result::<Session>(&self.0.get()?)
            .map_err(|_| ServerError::UpdateToken.into())
    }
}
```

diesel.rs hosted with ❤ by **GitHub** **view raw**

UpdateSession Handler for actix-web powered by Diesel.rs

For the connection handling between actix-web and Diesel the r2d2 project is used. This means we have (beside the application with its workers) an shared application state which holds multiple connections to the database as a single connection pool. This makes the whole backend very easily large scaling and flexible. The whole server instantiation can be found here.

Backend testing

The integration testing of the backend is done by setting up a test instance and connecting to an already running database. Then a standard WebSocket client (I used tungstenite) can be used to send the protocol related Cap'n Proto data to the server and evaluate the expected results. This worked pretty well! I did not use the actix-web specific test servers because setting up a real server was not much more work. Unit testing of the other parts of the backend worked as simple as expected and produced no real pitfalls.

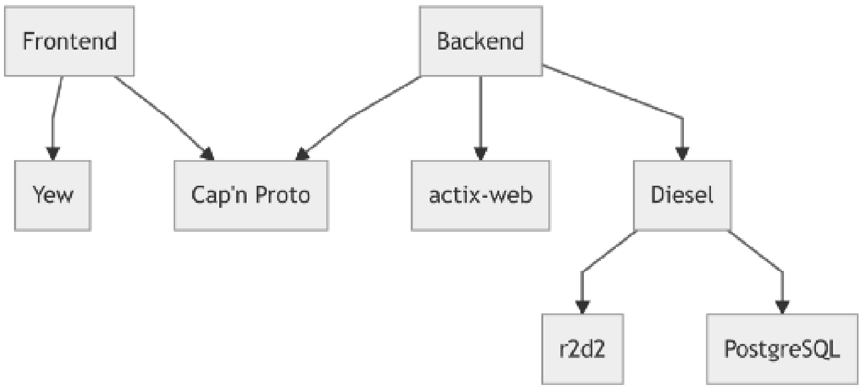
The Deployment

Deploying the application can be done easily via an Docker image.

The Makefile command `make deploy` creates a Docker image called `webapp`, which contains the statically linked backend executable, the current `Config.toml`, TLS certificates and the static content for the frontend. Building a fully statically linked executable in Rust is achieved with a modified variant of the rust-musl-builder docker image. The resulting webapp can be tested with `make run`, which starts the container with enabled host networking. The PostgreSQL container should now run in parallel. In general, the overall deployment is not that big part of the deal and should be flexible enough for future adaptions.

Summary

As a summary, the basic dependency stack of the application looks like this:



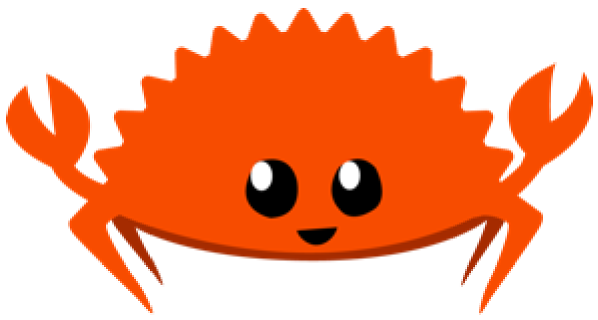
The only shared component between the frontend and backend is the Cap'n Proto generated Rust source, which needs a locally installed Cap'n Proto compiler.

So, are we web yet (in production)?

That is the big question, my personal opinion on that is:

> *On the backend side I would tend to say "yes", because Rust has beside actix-web a very mature HTTP stack and various different frameworks for building APIs and backend services quickly.*

> *On the frontend side is also a lots of work ongoing because of the WebAssembly hype, but the projects needs to have the same matureness as the backend ones, especially when it comes to stable APIs and testing possibilities. So there is a "no" for the frontend, but we're on a pretty good track.*



Thank you very much for reading until here. ❤️

I will continue my work on the demonstration application to continuously find out where we are in Rust in relation to web applications. Keep on rusting!