Sascha Grunert
Oct 3 · 4 min read

Lessons learned on writing web applications completely in Rust

> *This blog post is an update to the preceeding article "A web application completely written in Rust" and summarizes the projects' progress over the last months. Before continuing, please consider reading the initial blog post first.*

The initial idea when starting with the project four months ago was to evaluate the frontend and backend capabilities of Rust, a programming language mainly designed for systems programming. Iteratively finding out a good software architecture is one of the major goals and leads into continuous change within the project.

> *I'm now happy to say that the current state of the project provides a good starting point for users who want to start on full stack development with Rust.*

Major changes

Lots of things have changed under the hood in the last three months and new features (like HTTP redirects) have been added to the application. I will focus here on the major software-architectural parts. Let's start with the first big change, the project structure: I decided to use cargo workspaces for frontend and backend separation instead of a single crate. This opens the possibility of producing a clearly separated internal and external API between the backend, frontend and core. This also implicates that the dedicated *Cargo.toml* manifests are now better separating the runtime and development dependencies.

The core protocol

Another part which has changed was the removal of the Cap'n Proto protocol for backend to frontend (and vice versa) communication. The application never used the Remote Procedure Call (RPC) capabilities of Cap'n Proto, because this part does not build for WebAssembly right now. This means Cap'n Proto was mainly used for data structure generation, which does not fit that nicely into the overall software architecture.

The main advantage of the removal is that the source code generation is not necessary any more which drastically reduced the complexity of the application. The protocol is now packed directly in Rust structures, which means that we don't have any code generation to overcome:

```rust
//! Basic models

#[cfg(feature = "backend")]
use schema::sessions;
use std::convert::From;

#[cfg_attr(feature = "backend", derive(Insertable, Queryable))]
#[cfg_attr(feature = "backend", table_name = "sessions")]
#[derive(Debug, PartialEq, Deserialize, Serialize)]
/// A session representation
pub struct Session {
    /// The actual session token
    pub token: String,
}

impl Session {
    /// Create a new session from a given token
    pub fn new<T>(token: T) -> Self
    where
        String: From<T>,
    {
        Self {
            token: token.into(),
        }
    }
}
```

The *Session,* including the authentication token, is now generically usable by the frontend, backend and the database driver which is a huge improvement to the previous de-/serialisation approach. This means we do not need further abstractions to database models: They can be used directly in backend and frontend. Hooray! 👏🏼

Let’s have a look at the request and response messages:

```rust
//! Request messages

use protocol::model::Session;

#[derive(Debug, PartialEq, Deserialize, Serialize)]
/// The credentials based login request
pub struct LoginCredentials {
    /// The username
    pub username: String,

    /// The password
    pub password: String,
}

#[derive(Debug, PartialEq, Deserialize, Serialize)]
/// The session based login request
pub struct LoginSession(pub Session);

#[derive(Debug, PartialEq, Deserialize, Serialize)]
/// The logout request
pub struct Logout(pub Session);
```

```rust
//! Response specific implementations

use protocol::model::Session;

#[derive(Debug, PartialEq, Deserialize, Serialize)]
/// The login response
pub struct Login(pub Session);

#[derive(Debug, PartialEq, Deserialize, Serialize)]
/// The logout response
pub struct Logout;
```

Nothing big has changed here, but as already mentioned we are able to use the *Session* in both requests and responses. Another fun fact is that the *Logout* response is represented by an empty struct type, because there is simply nothing to be done after a successful logout from the web application.

Another new feature of the protocol is that the de-/serialisation will now be done using CBOR, whereas the data is transmitted via REST (within the body of HTTP messages) instead of a WebSocket connection. You might think that the additional overhead would have performance implications, but there were two major reasons to go for the more classic REST approach:

1. Load balancing and resource sharing of the different API paths is easier (especially with actix-web) when they terminate in dedicated handler functions instead of a single WebSocket.
2. The application is now ready for providing a third-party API interface to other users. The API is defined within the core part which will be used by the backend and frontend:

```rust
/// A simple macro for API definitions
macro_rules! apis {
    ($($name:ident => $content:expr,)*) => (
        $(#[allow(missing_docs)] pub const $name: &str = $content;)*
    )
}

/// Available API definitions
apis! {
    API_URL_LOGIN_CREDENTIALS => "login/credentials",
    API_URL_LOGIN_SESSION => "login/session",
    API_URL_LOGOUT => "logout",
}
```

api.rs hosted with ❤ by GitHub view raw

Backend changes

The change of the major protocol interface implicates larger adaptions within the backend of the application. As an example, the “login with credentials” (username and password) handler function now looks like this:

```rust
pub fn login_credentials<T>(http_request: &HttpRequest<State<T>>) -> FutureResponse
where
    T: Actor + Handler<CreateSession>,
    <T as Actor>::Context: ToEnvelope<T, CreateSession>,
{
    let (request_clone, cbor) = unpack_cbor(http_request);
    // Verify username and password
    cbor.and_then(|LoginCredentials{username, password}| {
            debug!("User {} is trying to login", username);
            if username.is_empty() || password.is_empty() || username != password {
                return Err(ErrorUnauthorized("wrong username or password"));
            }
            Ok(username)
        })
        // Create a new token
        .and_then(|username| Ok(Token::create(&username)?))
        // Update the session in the database
        .and_then(move |token| {
            request_clone
                .state()
                .database
                .send(CreateSession(token))
                .from_err()
                .and_then(|result| Ok(HttpResponse::Ok().cbor(Login(result?))?))
        })
        .responder()
}
```

At first, we unpack the CBOR content from the HTTP request. Afterwards, a small sanity check for empty username and password will return an HTTP 401 (Unauthorized) error in that case. This is another benefit to the previous WebSocket based approach since we now have a more clear error reporting interface via HTTP codes. Furthermore, it is now possible to use the more concise future-based programming style the Rust futures crate offers. After the sanity checks we create a new session token and put it directly into the database via the Diesel object-relational mapping (ORM). In general, the request handling is now better separated and much more clearer than the initial approach. Another big “hooray”! 👏

Frontend changes

The first major change within the frontend is that the router has now its own repository and can be used by other yew-based applications too. Routing to dedicated components of the frontend is now really easy and can be done like this (pseudo) example:

```rust
#[macro_use]
extern crate yew_router;

// Define application routes via the macro
// This creates the `RouterTarget` struct
routes! {
    Login =>  "/login",
    Content => "/content",
}

// Implement `Renderable` for this target
impl Renderable<RootComponent> for RouterTarget {
    fn view(&self) -> Html<RootComponent> {
        match *self {
            RouterTarget::Login => {
                html! {
                    <LoginComponent:/>
                }
            }
            RouterTarget::Content => {
                html! {
                    <ContentComponent:/>
                }
            }
        }
    }
}


// Create a component containing a router_agent instance
router_agent.send(yew_router::Request::ChangeRoute(
    RouterTarget::Content.into(),
));
```

A fully working example is included within the RootComponent of the webapp.rs application.

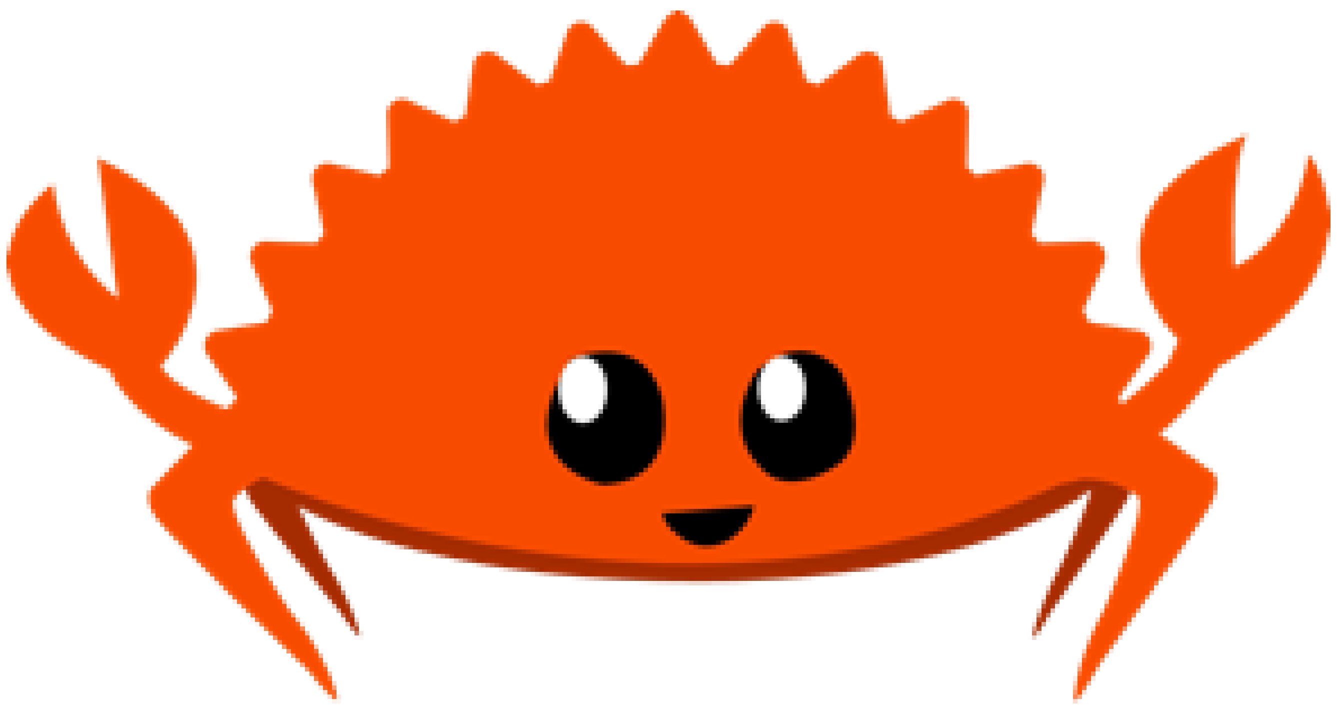
Since we removed the WebSocket, the backend communication can now be done using the standard fetch API, where yew already includes an interface. I decided to create a macro which allows an easier usage like this:

```rust
self.fetch_task = fetch! {
    LoginCredentials { // the request struct from the core
        username: "username".to_owned(),
        password: "password".to_owned(),
    } => API_URL_LOGIN_CREDENTIALS, // reuse the global API constants
    self.component_link, // the current component
    Message::Fetch, // the message type for the component
    || {}, // Do something on success
    || {}, // Do something on error
};
```

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

Again, the core data structures can be reused here for convenience, which is pretty awesome and makes backend interaction within the frontend seamlessly possible!

I'd like to mention that yew is just in the development phase where it is not ready for a real production use yet. I think yew needs a higher attention from the Rust community. So please, developers, contribute to that great project! In general, Rust is on a pretty good track for replacing web application stacks in the future, whereas the major construction areas are related to testing and WebAssembly topics.



So that’s the update for now. Feel free to directly get in touch with me, ask me questions or provide any feedback in one of the published communities.

> *Thank you very much for reading and keep on Rusting!* ❤️