Rewriting the Modern Web in Rust

> *Building a modern web app with Rust, Bazel, Yew and Axum.*

Earlier this year I rewrote my website with Next.js, React, tsx, and mdx. Having tried full-stack rust in the past, I didn't think its developer experience was on par with the Next.js stack. Well times have changed, and I wanted to see just how far I could push rust to feel like Next.js. So I did what any developer would do and rewrote my personal site… again.

The Destination

This post is a summary of my journey to a full-stack rust web application. I'll give an overview of how I used Yew and Axum to build a single-page application (SPA) with server-side rendering (SSR), Hooks (Yew function components), Markdown with embedded Yew components (MDX), and code syntax highlighting with Prismjs. The entire build uses Bazel's rust support, including a local development server, cross-compiling to linux with zig, and assembling a container image for deployment to a serverless or container runtime.

Let's get started!

> *Feel free to checkout out the source code directly on GitHub.*

Yew Function Components (Hooks)

When I last rewrote my personal site, I found React Hooks to be an elegant way to write UI state and render logic. I wanted a similar experience for this rewrite, and was stoked to see Yew now has Hooks, by the name of Function Components.

A function component is a modular UI element represented as a function. The function takes in parameters (called Props), evaluates any necessary business logic, sets up interactive callbacks (like `onclick` handlers), and finally returns an HTML-like rendering of the UI. Function components map onto MVC where the Props are the model, the function body is the controller, and the return value is the view.

```
// A simple function component
#[function_component]
fn Counter() -> Html {
    //  store an integer count
    let count = use_state(|| 0);
    //  increment on click
    let click = use_callback(|_, [count]| count.set(**count + 1)

    html! {
        <button onclick={ click }>
            {"Counter "}{*count}
        </button>
    }
}
```

The above code renders as: Counter 0 . It's that easy to create an interactive component that stores state and responds to input. It's also trivial to compose them by just adding `<ComponentName />` to the return value of the parent component.

Single-Page Application (SPA) Routing

Now that we've covered the core app logic using function components, the next piece of the puzzle is how components are organized in a hierarchy.

One advantage of writing an application as a SPA is that the browser does not "reload" or flash when the user navigates to another page. Instead, when the user clicks a button or link, local application code on the frontend renders and replaces the DOM. This also eliminates the latency of requesting a new page from the server.

Yew comes with SPA support included in the form of `yew-router`. Similar to `react-router`, `yew-router` lets you define a hierarchy of application routes, and then map each route to a specific component. Here's how my website is structured:

```
//  My website routes
enum Route {
    #[at("/")]
    Home,
    #[at("/blog")]
    BlogIndex,
    #[at("/blog/:slug")]
    BlogPost { slug: String },
    #[at("/projects")]
    Projects,
}
```

Each variant of the `Route` enum is a different page on my website. The `#[at(..)]` attribute macro tells Yew what the path of each page should be. Routes can even have parameters, which I used in the `BlogPost` variant to specify which post should be rendered.

Here's the (simplified) route-to-component routing logic for this website:

```
fn switch(route: Route) -> Html {
    match route {
        Route::Home => html! {
            <h1>{"impl Future {}"}</h1>
        },
        Route::BlogIndex => blog::blog_index(),
        Route::BlogPost{slug} => blog::render(&slug),
        Route::Projects => html! {
```

```
                <Projects />
            },
        }
    }
```

Since `Route` is a rust enum, I am forced to implement every possible route. If I were to add another route to the enum, the compiler would error until I also implement the render logic in this `switch` function. The `BlogPost` variant takes advantage of the arbitrary structure of rust enums – I know that if `route` is `BlogPost`, there is also a valid `slug: String` field that was parsed by `yew-router`.

Server-Side Rendering (SSR)

SPAs are great but pose a couple of problems:

- The client has to download all of the application code (in this case WASM) before the page can be rendered.
- Web crawlers will run a limited amount of code to understand the content of your website, resulting in bad Search Engine Optimization (SEO).

Similar issues exist in the React/javascript world, and the most common solution is Server-Side Rendering. Here is the typical SSR flow:

- The server receives the initial HTTP request from the client
- The server runs its own copy of the application code on the path of the request
- The resulting DOM is serialized to a string and injected into the initial HTML response
- The client immediately renders the initial HTML page and downloads the application code
- The client starts the local application, attaching to the initial DOM nodes (also called hydration)

Luckily, Yew has implemented both the render-to-string and hydration steps! This just leaves tying it all together with a web server.

Let's see how we can make this all work with Axum.

The basic logic we need to handle an incoming request is:

- If the path matches a path in our Yew app, serve `index.html` with the first render of that page injected into it.
- Otherwise, try serving a static file from `static/`.

For the first part, we need a tower service that detects whether a path matches our Yew app, and otherwise calls another fallback service. I couldn't find anything in axum or tower that would do this out of the box, so I wrote my own service:

```
//  tower service that matches a request to a Yew App route, or
//  another service (lots of boilerplate omitted)
struct RoutableService<S, F> {
    yew_service: S,
    fallback_service: F,
}

impl <R, S, F> Service for RoutableService
where
    R: yew_router::Routable, S: Service, F: Service,
{
     fn call(&mut self, req: Request<Body>) -> Self::Future {
        match <R as Routable>::recognize(req.uri().path()).is_sc
            //  if request path matches Yew route, serve S
            true => self.yew_service.call(req),
            //  else serve F
            false => self.fallback_service.call(req),
        }
    }
}
```

For the full version, check it out on GitHub.

With a Yew-route-aware service, everything can now be pulled together:

```
fn yew_ssr(req: Request) -> impl IntoResponse {
    let props = ServerAppProps {
        path: url.uri().path().to_owned().into(),
        queries,
    };
    let mut out = String::new();
    yew::ServerRenderer::<implfuture::ServerApp>::with_props(prc
        .render_to_string(&mut out)
        .await;
    //  index.html contents read at compile-time, with first rer
    //  into <body>
```

```
        INDEX_HTML.replace("<body>", &format!("<body>{}", out));
    }

    //  static files like wasm, js, images, and css
    let static_serve = tower_http::ServeDir::new("static");

    //  Try Yew app first, fall-back to static files.
    let serve = RoutableService {
        yew_service: yew_ssr,
        fallback_service: static_serve,
    };
```

MDX

While Yew's `html!` macro is great for writing small components, writing a blog post out by hand would be painstaking. I loved using MDX with embedded typescript React components in my last website rewrite, and wanted to bring the same experience to rust/yew.

I added an `mdx!` macro to the `yew_macro` crate which lets me write

```
mdx! {r#"
    # Title
    A list of things:
    - thing One
    - thing Two
    ```rust
    // rust code block
    fn main() {}
    ```
"#}
```

instead of the equivalent `html!` syntax:

```
html! {
    <h1> {"Title"} </h1>
    <p>{"A list of things:"}</p>
    <ul>
        <li>{"thing One"}</li>
        <li>{"thing Two"}</li>
    </ul>
    <pre><code>
```

```
    // rust code block
    fn main() {}
    </code></pre>
}
```

> *Note: the string literal wrapper in `mdx!` is a work-around to have a whitespace-sensitive proc-macro. If anyone knows a cleaner alternative let me know!*

The `mdx!` proc-macro is a Markdown frontend to the `yew::html!` macro. It uses the `pulldown-cmark` crate to convert `#` to `<h1></h1>`, `` ` `` to `<code></code>`, and so on.

Yew components can be embedded with their usual html syntax:

```
fn MyComponent() { html! {<p>{"Component"}</p>}}

mdx! {r#"
    <Component />
"#}
```

I also added support for replacing all instances of a markdown element with a custom Yew component. For example, all header tags on my blog are labelled with an `id` based on their text and are turned into clickable links.

```
mdx_style!(
    h1: MyH1,
);

fn MyH1(c: &ChildProps) -> Html {
    let slug = /* turn header text into string */;
    html! {
        //  Make all headers deeplink-able
        <a href={format!("#{slug}")}>
            <h1 id={tag}>
                {c.children.clone()}
            </h1>
        </a>
    }
}
```

To top it off, I added an `include_mdx!` macro to parse an external Markdown file from rust code. This lets me write Markdown with all of the usual IDE support, and without the `r#"..."#` wrapper.

```
blog.mdx

# Blog Post

> Subtitle
```

```
fn blog() -> Html {
    include_mdx!("blog.mdx")
}
```

All of these features can be previewed in the mdx macro unit tests.

Bazel

deep breath...

Alright, we've covered a lot already. Between SSR, wasm, and external Markdown files, our build is already more than a little *complicated*.

When starting this project, I initially used `trunk` to build the wasm. Then I used normal `cargo` to build the native server binary and `include!`-ed the compiled yew wasm to serve at runtime. I tied all of this together with a bash script and `cargo-watch` to automatically re-compile the server on every code change. Unfortunately, I ran into issues with `cargo-watch` not always recognizing when to re-compile that I never figured out how to completely fix. I also couldn't figure out a way to parallelize the building of the app and the server, and instead just built them sequentially. Things got even more complicated when incorporating TailwindCSS and other tooling we'll cover later in this post.

Along comes Bazel, Google's open-source version of their internal `blaze` build system. Bazel is a hermetic build system, where each build step (also called a rule) declares its dependencies, outputs, and how to build it. This explicitness

allows bazel to know exactly what needs to be re-built and what can be parallelized.

For the rest of this post, bazel will be the glue with which we tie everything together.

Bazel Rust Project Layout

The `rules_rust` project adds rust support to bazel, including support for crates.io and wasm_bindgen.

The project is split into the core Yew application based in the root directory, and the web server in the `server/` directory. The root `BUILD` file defines the Yew app twice:

- first as a library for our server to use in SSR
- second as a binary to run on the web client in “hydration” mode

```
# /BUILD
rust_library(
    name = "implfuture",
    srcs = glob(
        include = [
            "src/**/*.rs",
        ],
        # exclude hydration entry-point
        exclude = ["src/bin/**"],
    ),
    # include mdx files for include_mdx!()
    compile_data = glob(["src/**/*.mdx"]),
    edition = "2021",
    # pulls crates from crates.io/crates_universe
    deps = all_crate_deps(
        normal = True,
    ),
)

# hydration wasm
rust_binary(
    name = "app",
    srcs = ["src/bin/app.rs"],
    edition = "2021",
    deps = all_crate_deps(
        normal = True,
    ) + [
```

```
        # depends on core app logic
        ":implfuture",
        "@rules_rust//wasm_bindgen/3rdparty:wasm_bindgen",
    ],
)
```

We can then compile our client-side hydrating application code to wasm using the `rust_wasm_bindgen` rule from `rules_rust`:

```
# /BUILD

rust_wasm_bindgen(
    name = "app_wasm",
    target = "web",
    wasm_file = ":app",
)
```

Finally, our server `BUILD` file ties it all together:

```
# /server/BUILD

rust_binary(
    name = "server",
    srcs = glob(["src/**/*.rs"]),
    # files served at runtime
    data = [
        "//:app_wasm",
        "//:static_files",
    ],
    deps = all_crate_deps(
        normal = True,
    ) + [
        # application code for SSR
        "//:implfuture"
    ],
)
```

The `data` field tells bazel that the web server needs the app wasm code at runtime. Bazel uses this information to know it can parallelize compilation of the wasm code and the native code. Bazel also knows exactly what to recompile. For example, if we change a CSS static file, bazel knows that it does not need to recompile the server since CSS is just a runtime data dependency.

With all of this set up, we can run a local development server with

```
$ bazel run //server
iBazel: Starting...
starting server on 127.0.0.1:8080
```

If we want things to automatically rebuild, all we need to do is use `ibazel` instead of `bazel`

```
$ ibazel run //server
iBazel: Starting...
starting server on 127.0.0.1:8080
```

Tailwind

While writing React apps, I grew accustomed to using TailwindCSS for styling individual elements. Tailwind defines many short class names for commonly used CSS. For example, `class="py-4"` in HTML will correspond to `.py-4 { padding-top: 1rem; padding-bottom: 1rem; }` in CSS.

To minimize the size of Tailwind's CSS file, it scans your source code to see which classes you are using. This is normally done on javascript or typescript, but Tailwind can be configured to run on any file, including rust:

```
// /tailwind.config.js
module.exports = {
  content: ["src/**/*.rs"],
  // ...
};
```

With Tailwind configured, we need a way for bazel to run the Tailwind cli, `tailwindcss`, and generate our CSS file. Instead of relying on the developer to have `tailwindcss` installed, we can do things the Bazel way and have a specific version of the Tailwind toolchain installed.

A `package.json` file defines what libraries we need

```
//  /package.json
{
  "name": "implfuture",
  "version": "0.1.0",
  "dependencies": {},
  "devDependencies": {
    "tailwindcss": "^3.1.8"
  }
}
```

And in our `WORKSPACE` file, we install those packages

```
# /WORKSPACE
# use bazel rules_nodesjs to install
yarn_install(
    name = "root_npm",
    package_json = "//:package.json",
    yarn_lock = "//:yarn.lock",
)
```

The `tailwindcss` binary is now available to other bazel rules we write. Bazel has a special rule called `genrule` that lets you define a rule using a shell command. Here's the `genrule` to generate the tailwind CSS file.

```
# /BUILD

genrule(
    name = "tailwind",
    # include our rust files to scan for used class names, as we
    # tailwind configuration
    srcs = glob(["src/**/*.rs"]) + ["tailwind.config.js"],
    outs = ["static/tailwind.css"],
    cmd = "$(execpath @root_npm//tailwindcss/bin:tailwindcss) --
    # pull in npm dependency
```

```
    tools = ["@root_npm//tailwindcss/bin:tailwindcss"],
    visibility = ["//:__pkg__"],
)
```

PrismJS

No good developer blog post is complete without syntax highlighting. I tried to find a rust-native implementation of syntax highlighting, but I couldn't find anything as full-featured as PrismJS (which is what I used for the previous rewrite of this site). While I had avoided javascript for as long as possible, I thought this might be a good opportunity to try incorporating a JS library with a Yew app.

My approach was to create a separate Javascript file that would bundle any JS dependencies our app uses and link to it in the `index.html` file. In the interest of keeping things bazel-y, I defined another `package.json` for runtime dependencies in the `bundle/` folder:

```
// /bundle/package.json

{
  "name": "implfuture",
  "version": "0.1.0",
  "dependencies": {
    "prismjs": "^1.28.0"
  }
}
```

```
# /WORKSPACE

yarn_install(
    name = "app_npm",
    package_json = "//bundle:package.json",
    yarn_lock = "//bundle:yarn.lock",
)
```

Since there are bazel rules for it, I used the `esbuild` bundler to bundle prism. There's even an esbuild prismjs plugin that helps configure PrismJS features

and included languages.

The whole app bundle is pulled together in the `bundle/BUILD` file:

```
# /bundle/BUILD

# typescript bundle entrypoint
ts_project(
    name = "tsproject",
    srcs = [
        "app.ts",
    ],
    deps = ["@app_npm//prismjs"],
)

# output bundle, served by web server
esbuild(
    name = "bundle",
    config = ":esbuild_config",
    entry_point = "app.ts",
    visibility = ["//:__pkg__"],
    deps = [
        ":tsproject",
    ],
)

esbuild_config(
    name = "esbuild_config",
    # includes configuration of PrismJS
    config_file = "esbuild.config.mjs",
    deps = [
        "@root_npm//esbuild",
        "@root_npm//esbuild-plugin-prismjs",
    ],
)
```

With PrismJS available to my app at runtime, I could now use it from Yew. PrismJS can be configured to automatically syntax highlight any code it finds in the DOM, but care must be taken when using Yew. Yew expects to be the only code manipulating the DOM. If something else adds or removes DOM nodes, it can cause unexpected behavior when Yew comes back to re-render that part of the page.

To have full control over when Prism highlights a code block, I chose to use the `Prism.highlightElement` function. First, it must be defined in rust as

`extern fn`:

```
mod prism {
    use wasm_bindgen::prelude::*;
    #[wasm_bindgen]
    extern "C" {
        #[wasm_bindgen(js_namespace = Prism)]
        pub fn highlightElement(element: web_sys::Element);
    }
}
```

Then I defined a Yew component that highlights code:

```
fn HighlightCode(c: &super::ChildProps) -> Html {
    //  save code tag to be used for highlighting
    let code_ref = use_state_eq(|| NodeRef::default());
    let mut code_tag = c.children.iter().next().unwrap().clone()
    match &mut code_tag {
        VNode::VTag(t) => t.node_ref = (*code_ref).clone(),
        _ => {}
    };

    use_effect_with_deps(
        move |_| {
            //  highlight code whenever children change, causes
            //  nodes under .codecontainer div
            let element = code_ref.cast::<Element>().unwrap();
            prism::highlightElement(element.clone());

            //  cleanup: remove DOM nodes created by Prism
            move || {
                element
                    .closest(".codecontainer")
                    .ok()
                    .flatten()
                    .map(|e| e.remove());
            }
        },
        c.children.clone(),
    );

    html! {
        //  wrap everything in a .codecontainer div for easier c
        <div class="codecontainer">
            <pre class="overflow-auto m-4 p-6 bg-gray-300/5 rour
                {code_tag}
            </pre>
        </div>
```

```
        }
    }
```

This component makes use of `use_effect_with_deps` to re-highlight code whenever the contents of the DOM node change. The closure returned in `use_effect_with_deps` is the cleanup code, and is run whenever this component is un-mounted from the DOM. The cleanup code is meant to restore the DOM to its previous state before the Prism highlighting. It works in practice, but I still get a warning from Yew (`app_wasm.js:437 Node not found to remove VTag`) when I navigate away from a page with syntax highlighting. Clearly this still needs some tweaking 🙂.

WASM Code-Size Optimization

One obstacle with compiling rust to wasm is that the resulting wasm file can be pretty large. As of writing this blog, the unoptimized wasm for this site is 5.1MB. This is clearly way too big for a website. Even though SSR will give the user a contentful page immediately, all interactivity like `onclick`s and `use_effect`s is delayed until the app wasm fully loads and hydration runs.

The rustwasm book has some great tips on optimizing wasm code size.

wasm-opt

`wasm-opt` is a part of the Emscripten SDK and is included in its bazel toolchain. `wasm-opt` optimizes wasm code for both performance and code-size. Using `wasm-opt -Os` reduced the app wasm size from 4.9MB to 1.3MB.

Incorporating `wasm-opt` into bazel in a cross-platform way was a little tricky. The `emsdk` bazel toolchain exposes `wasm-opt` at different paths depending on the platform. This meant that a `genrule` would have to be aware of which platform it was running on in order to locate the correct `wasm-opt`. Instead, I wrote a custom rule that invokes `wasm-opt` using starlark:

```
# /emsdk/emsdk.bzl

# expects exactly one .wasm file and one output path
```

```
def _wasmopt_impl(ctx):
    tc = ctx.toolchains["//emsdk:toolchain_type"]
    info = tc.emsdkinfo
    wasm_srcs = [f for f in ctx.attr.src.files.to_list() if f.pa
    if len(wasm_srcs) != 1:
        fail("expected 1 wasm file, got %s" % wasm_srcs)
    wasm_src = wasm_srcs[0]

    ctx.actions.run(
        inputs = [wasm_src],
        outputs = [ctx.outputs.out],
        executable = info.wasmopt,
        arguments = ["-Os", wasm_src.path, "-o", ctx.outputs.out
    )
```

This rule uses the bazel toolchain construct to select the version of `emsdk` compatible with the host. I also defined `emsdk` toolchains, similar to how `rust` or `go` toolchains are defined:

```
# /emsdk/emsdk.bzl

def declare_toolchains(name):
    for tc in TOOLCHAINS:
        emsdk_toolchain(
            name = "emsdk_{tc}".format(tc = tc),
            linker_files = "@emscripten_bin_{tc}//:linker_files"
        )
        native.toolchain(
            name = "emsdk_{tc}_toolchain".format(tc = tc),
            toolchain = ":emsdk_{tc}".format(tc = tc),
            exec_compatible_with = TOOLCHAINS[tc]["exec"],
            toolchain_type = "//emsdk:toolchain_type",
        )

def register_toolchains():
    native.register_toolchains(
        *["//emsdk:emsdk_{tc}_toolchain".format(tc = tc) for tc
    )
```

Any bazel project then just needs to call `register_toolchains()` in its `WORKSPACE` file and the correct `emsdk` will be downloaded when the `wasmopt` rule is used.

Brotli

While `wasm-opt` brought the app down from 4.9MB to 1.3MB, that's still way too big. The next option I looked at was compression. After experimenting with flate, gzip, and brotli, I found that brotli achieved the smallest size. With both `wasm-opt -Os` and `brotli -9`, the wasm file came down to 331KB in total.

Incorporating brotli into bazel was pretty simple since it's a Google project that already uses bazel for its build. First I added it to my `WORKSPACE`:

```
# /WORKSPACE

git_repository(
    name = "brotli",
    commit = "9801a2c5d6c67c467ffad676ac301379bb877fc3",  # 2022
    remote = "https://github.com/google/brotli",
)
```

Then I wrote a simple `genrule` to compress the output of `wasm-opt`:

```
# /BUILD

genrule(
    name = "app_wasm_opt_br",
    srcs = [":app_wasm_opt"],
    outs = ["app_wasm_bg_opt.wasm.br"],
    cmd = "$(execpath @brotli) -9 $<",
    tools = ["@brotli"],
)
```

Referencing `@brotli` in the `tools` argument automatically compiles `brotli` for the host architecture. This way, I can be sure exactly which version of `brotli` is being used, and don't need to rely on any host having it installed beforehand.

Deployment

To top it all off, I needed a way to package and host the web app. When using Next.js, Vercel was the obvious option. Instead, I decided to cross-compile the web server to linux and package it into a container.

Cross-Compilation

Rust includes support for cross-compilation, and adding the platforms I needed was as simple as adding these lines to my `WORKSPACE`:

```
# /WORKSPACE

rust_register_toolchains(extra_target_triples = [
    "wasm32-unknown-unknown",
    "x86_64-unknown-linux-gnu",
    "aarch64-unknown-linux-gnu",
])
```

`rules_rust` uses the same bazel toolchain abstraction I mentioned in the `wasm-opt` section of this post, and the cross-compilation really Just Works.

However, many rust crates also use C or C++, and I needed a toolchain that could cross-compile those languages. When looking for the easiest cross-compilation toolchain to use, I stumbled upon the blog post "Zig Makes Go Cross Compilation Just Work" and a follow-on "Zig Makes Rust Cross Compilation Just Work". These posts showcase using `zig` as a drop-in replacement for a C compiler and linker with great success.

Luckily, the `bazel-zig-cc` project provides bazel toolchain abstractions for the `zig` C/C++ toolchain. After recommending a small fix to address how `cargo` invokes `zig` as a linker, all I had to do was register the linux `zig` toolchains in my `WORKSPACE`.

One minor issue I ran into was that `cargo` would have issues when using a cross-compilation toolchain and compiling for the host architecture. To address this, I wrote a wrapper around zig toolchain registration that omits the zig toolchain for the host architecture.

Container Image

With my cross-compiled server binary ready to go, the next step was packaging it into a container. `rules_docker` provides bazel rules to create and manage OCI containers.

I first used the built-in `pkg_tar` rule to bundle all of the relevant files into one archive:

```
# /server/BUILD

pkg_tar(
    name = "opt_tar",
    # :opt refers to optimized server artifacts that have passed
    # wasm-opt and brotli
    srcs = [":opt"],
    include_runfiles = True,
    package_dir = "/app",
    # keeps relative path consistent for static files in both de
    strip_prefix = "/server",
)
```

Then a single `container_image` rule packages everything into a container:

```
# /server/BUILD

container_image(
    name = "image-amd64",
    architecture = "amd64",
    # References google-maintained base image defined in /WORKSP
    base = "@cc_base//image",
    cmd = ["/app/opt"],
    env = {
        # configure binary for production
        "HTTP_LISTEN_ADDR": "0.0.0.0:8080",
    },
    tars = [":opt_tar"],
    workdir = "/app",
)
```

> *Note: `container_image` uses a custom go library to build the image instead of requiring a Docker cli/daemon, however this tool doesn't currently work on Windows.*

Serverless

Infrastructure-wise, all I really needed was custom domain support, TLS termination, and some execution environment. The first thing that came to

mind was something like AWS ECS (Elastic Container Service), but keeping a container running all the time for just a personal website felt wasteful.

Instead, I settled on AWS Lambda, with a TLS certificate stored in ACM, and API Gateway as a frontend. This setup provided unlimited auto-scaling, and didn't even require setting up my own load balancer (load balancers are usually the most expensive part of deploying a small project).

All I needed was for my web server to support lambda instead of expecting incoming TCP traffic. Luckily, there's an awesome `lambda-web` crate that adapts common rust web servers like Axum to the lambda runtime API. All I had to add to my web server was:

```
// /server/src/main.rs

if lambda_web::is_running_on_lambda() {
    lambda_web::run_hyper_on_lambda(route_service)
        .await
        .map_err(...)?;
} else {
    //  tcp listen code...
}
```

With `lambda-web`, the same axum handlers, including static file serving and SSR, work for both my local development server *and* the production lambda runtime.

Wrapping Up

Thank you for joining me on this rust + bazel journey! I was able to push full-stack rust much further than I thought I would be able to. I converged on a relatively convenient developer and deployment experience, building on the shoulders of giants like bazel, rules_rust, and zig.

Don't get me wrong, there's still quite a few improvements I would like to see in this project / the rust web ecosystem in general. To name a few:

- code splitting (right now the entire application wasm is loaded no matter which

page the user is on)

- hot-reloading of app wasm so I can see updates without refreshing the page

- bazel remote build cache to speed up github worker builds (currently takes 20-30m on a free worker)

- blog table of contents (could likely be implemented in a component that scans the blog DOM after first render)

- compiling the server to `wasi` architecture

I hope you learned a bit about rust, wasm, or bazel along the way! See you next post!