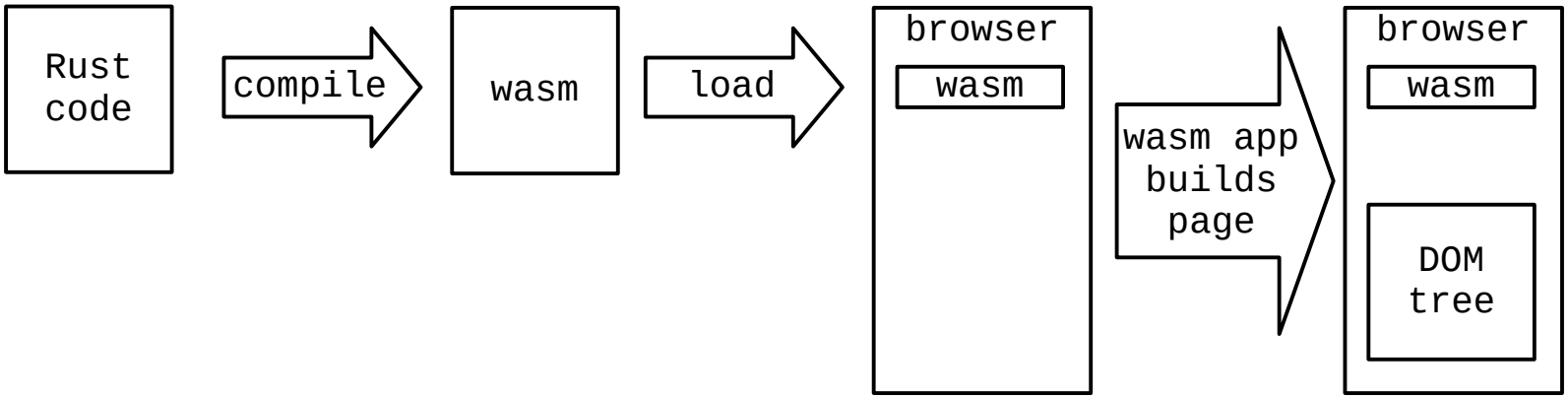
Web UI in Rust

How does it work?

Because of LLVM, Rust code can be compiled to WebAssembly, which can be natively executed by all modern web browsers.

WebAssembly can call any API exported for JS, including DOM manipulation API, which makes possible to build a whole HTML page with WebAssembly code only.



This is very similar to how React works: React JS application is loaded into the browser, and then it builds the page by building a virtual DOM and then applying it to the page.

The only difference is that instead of React application written in JavaScript (or TS), we have WASM application written in Rust.

About WebAssembly

WebAssembly – is a binary byte-code format that is:

- platform independent
- safe and deterministic
- compact, fast for decoding and designed for streaming

Both JS and WebAssembly are executed in the same sandbox: JS can call WebAssembly functions and vice versa.

Because WebAssembly can be initially called only from JS (at least at the moment), WebAssembly based applications are always represented as a bundle: a wasm file and a small JS script that calls an entry point function in wasm.

WebAssembly is just a bytecode format, similar to machine code. This means, that the size and the efficiency of a WebAssembly application highly depends on the compiler that has generated wasm file. E.g. a wasm generated from Rust will be smaller and faster than a wasm generated from Golang, because Golang has to put into the wasm file the whole Golang runtime, including the GC.

WebAssembly format

WebAssembly is a binary instruction format for a stack-based virtual machine (no registers).

It declares:

- only numerical primitive types (i32, i64 and f32, f64), but no strings
- v128 - 128 bit vector of packed integer, floating-point data, or a single 128 bit type
- instructions for working with stack-based
- instructions for working with memory
- comparison instructions: eqz, eq, ne, lt, gt, le,ge
- flow control instructions: block, loop, if, else, end, br, br_if, br_table, return, call, call_indirect
- integer instructions: clz, ctz, popcnt, add, sub, mul, div, rem, and, or, xor, shl, shr, rotl, rotr
- floating-point instructions: abs, neg, ceil, floor, trunc, nearest, sqrt, add, sub, mul, div, min, max, copysign
- instructions for converting data: wrap, extend, trunc, convert, demote, reitrpret
- SIMD instructions (since 2021)

Files with WebAssembly byte-code usually have format *.wasm.

There is also a text WebAssmebly representation (similar to how for a machine code there is assembly source code, but in case of WebAssembly it is one to one correspondence).

Files with textual form have *.wat extension.

WAT

WebAssembly text representation uses S-expression syntax, that is very similar to LISP.

Module in WebAssembly consists of (module) expression that contains declarations:

```
(module
  (imports … )
  (functions …)
  (exports …)
  …
)
```

Possible declarations: functions, import/export directives, variables, table declarations, memory chunk declarations.

E.g. module that exports a function that takes i32 parameter and returns it will look like:

```
(module
  (func $echoi32 (param $x i32) (result i32)
    local.get $a ;; place the argument on the stack
  )              ;; The top value on the stack - is the return result
  (export "echoi32" (func $echoi32))
)
```

Running WebAssembly

To run a WebAssembly function, we need to use WebAssembly class to compile wasm files into an executable representation.

```
WebAssembly.compile(wasm_bytes, js_to_import_into_wasm)
```

Like:

```
let response = await fetch("http://localhost/mymodule.wasm");
let bytes    = await response.arrayBuffer());
let module   = await WebAssembly.compile(bytes, {}));
let wasm_lib = await WebAssembly.instantiate(module, {});
```

After that we can call a function defined in the wasm file:

```
wasm_lib.instance.exports.my_wasm_function(args);
```

WebAssemly.compile takes bytes array, which means that wasm file should be fully downloaded. But WebAssembly format developed in the way that allows to compile it in the streaming mode while the wasm file is downloaded. There is another function - *WebAssembly.instantiateStreaming* that does the same, but in the streaming way:

```
WebAssembly.instantiateStreaming(fetch("mymodule.wasm"), {})
  .then( (wasm_lib) => {
    console.log(wasm_lib.instance.exports.my_wasm_function(args);
  });
```

Example 1: Exporting a function from WebAssembly to JS

Let's write a simple WebAssembly function that sums two numbers:

sum2.wat

```
(module
  (func $sum2 (param $a i32) (param $b i32) (result i32)
    local.get $a  ;; push a to the stack
    local.get $b  ;; push b to the stack
    i32.add)      ;; pop two elements from the stack and push their sum
  (export "sum2" (func $sum2))
)
```

Translate textual form into binary that can be executed:
(To get wat2wasm tool, you need install https://github.com/webassembly/wabt.

```
wat2wasm sum2.wat -o sum2.wasm
```

HTML page with simple script that loads our binary wasm file, and call the function:

index.html

```
<html>
</html>
<script>
WebAssembly.instantiateStreaming(fetch("sum2.wasm")).then(
  (wasm_lib) => {
    console.log(wasm_lib.instance.exports.sum2(1, 2));
  });
</script>
```

The folder should look like:

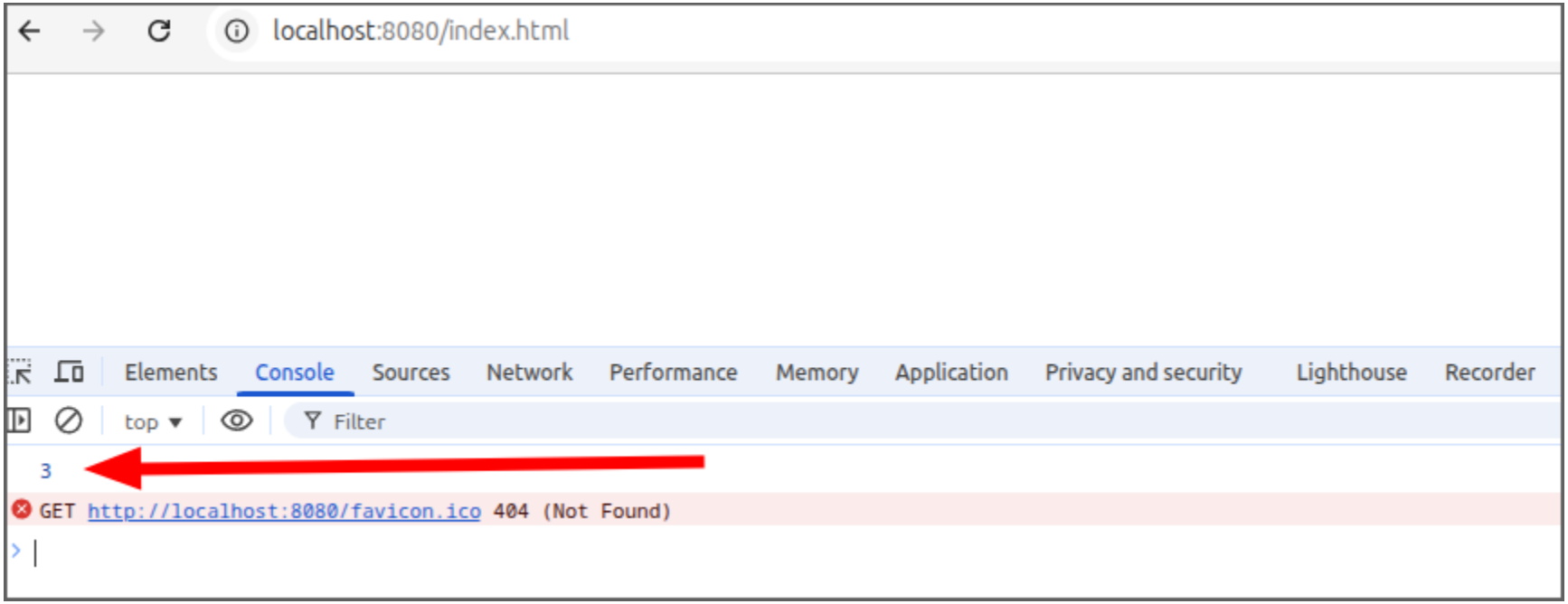
```
/
├─ sum2.wat
├─ sum2.wasm
└─ index.html
```

Now we can start an http service and specify the folder as the server root directory:

(simple-http-server is a simple and convenient HTTP server written in Rust https://github.com/TheWaWaR/simple-http-server)

```
simple-http-server --cors -i -p 8080 ./
```

Result in browser console:



Tools for wasm

There are two major popular instruments for working with wasm:

- watb – https://github.com/webassembly/wabt
- wasm-tools – https://github.com/bytecodealliance/wasm-tools

Both allow to:

- view the content of wasm files
- translate between binary and text representation
- validate

The difference is that *watb* is written in C++ and should be downloaded and installed as a regular tool, when *wasm-tools* is written in Rust and can be installed by calling `cargo install wasm-tools`.

E.g. to view the list of declarations in a wasm file, we can call:

```
$ wasm-tools print --skeleton sum2.wasm
(module
  (type (;0;) (func (param i32 i32) (result i32)))
  (export "sum2" (func 0))
  (func (;0;) (type 0) (param i32 i32) (result i32) ...)
)
```

Example 2: Importing JS function to WebAssembly

To be able to call a function defined JS, the function should be imported into the WebAssembly using so called import obj - second argument of *WebAssembly.compile* function.

```
<html>
</html>
<script>
let importObj = {
  console: console
};

WebAssembly.instantiateStreaming(fetch("print_num.wasm"), importObj).then(
  (wasm_lib) => {
    wasm_lib.instance.exports.entry_point();
  });
</script>
```

print_num.wat

```
(module
  (import "console" "log" (func $log_number (param i32)))
  (func $entry_point
    i32.const 5
    call $log_number)
  (export "entry_point" (func $entry_point))
)
```

Why Rust and WebAssembly?

- It is very convenient to have both back-end and front-end in Rust. Not only it reduces amount of tools and technologies you have to work with, but also makes interaction between back-end and UI more seamless, because they share same types and same code for serialization and de-serialization.

- Surprisingly, it can be easier to write Web UI in Rust.
  When writing Web UI in JS, you usually have to transpile Typescript or JS (from the most recent ECMA version to the older one that is supported by all browsers), combine all libraries into one big JS file, perform minimization and obfuscation. To do all this you need a pipeline of tools like Babel, webpack, etc.
  When you write a Web UI in Rust, you just need Cargo and trunk (to build a WebAssembly bundle).

- For functionality that involves a big amount of computations, WebAssembly can give a significant boost in performance.

Example 3: Exporting Rust WebAssembly function to JS

If you don't have Rust SDK installed, please follow the instruction to install it: https://www.rust-lang.org/tools/install

Install wasm build target for Rust compiler:

```
rustup target add wasm32-unknown-unknown
```

Create a new project

```
cargo new sum2 --lib
```

The project structure should look like:

```
/
├─ Cargo.toml
└─ src/lib.rs
```

Cargo.toml

```
[package]
name = "sum2"
version = "0.1.0"
edition = "2024"

[lib]
crate-type = ["cdylib"]
```

Write wasm module code:

src/lib.rs

```rust
#[unsafe(no_mangle)]
pub unsafe extern "C" fn sum2(a: i32, b: i32) -> i32 {
    a + b
}
```

Compile wasm module:

```
cargo build --target wasm32-unknown-unknown
```

Compiled wasm file should appear in target directory:

```
/
├─ Cargo.toml
├─ src/lib.rs
└─ target
   └─ wasm32-unknown-unknown
      └─ debug
         └─ sum2.wasm
```

Now add *index.html* which almost same as for example with wat.

index.html

```
<html>
</html>
<script>
WebAssembly.instantiateStreaming(fetch("target/wasm32-unknown-unknown/debug/sum2.wasm"))
  .then( (wasm_lib) => {
    console.log(wasm_lib.instance.exports.sum2(1, 2));
  });
</script>
```

Final project structure:

```
/
├─ Cargo.toml
├─ src/lib.rs
├─ index.html
└─ target
   └─ wasm32-unknown-unknown
      └─ debug
         └─ sum2.wasm
```

Run HTTP server in the root folder of the project and open *index.html* in the browser.

```
simple-http-server --cors -i -p 8080 `pwd`
```

Example 4: Importing JS function to Rust WebAssembly

Let’s rewrite previous example in Rust:

src/lib.rs

```rust
mod console {
    #[link(wasm_import_module = "console")]
    unsafe extern "C" {
        pub fn log(x: i32);
    }
}

#[unsafe(no_mangle)]
pub unsafe extern "C" fn entry_point() {
    console::log(5);
}
```

index.html

```html
<html>
</html>
<script>
let importObj = {
  console: console
};

async function run() {
  let response = await fetch("target/wasm32-unknown-unknown/debug/print_num.wasm");
  let wasm_lib = await WebAssembly.instantiateStreaming(response, importObj);
  wasm_lib.instance.exports.entry_point();
}
run();
</script>
```

WebAssembly vs JS

On the internet you can find a lot of benchmarks: some of them show how fast WebAssembly is comparing to JS, and some of them show the opposite.

In reality, if we compare pure performance of WebAssembly and JS, the WebAssembly will be a clear winner. Not only it is executed faster, but it also doesn't require parsing, building of AST, transforming to an inner executable form, etc. (e.g. Figma and AutoCAD are running in browser using WebAssembly)

The only real "problem" that is not solved yet: Browsers do not export a dedicated DOM-manipulation API for WebAssembly, which means for any DOM manipulation, WebAssembly has to call API that is exported for JS. This requires a conversion from WebAssembly data types to JS data types which leads to some slowness that is visible in benchmarks only.

Numbers: https://krausest.github.io/js-framework-benchmark/2025/table_chrome_138.0.7204.50.html

Another common complain that Rust WebAssembly bundle sometimes can be up to two or even 3 times bigger than similar JS bundle (after minimization).

The thing is that let's say 100 KB WebAssembly is not the same that 100 KB JS. WebAssembly has been designed in the way that it is loaded for execution while it is still in the process of downloading. And JS should be first fully loaded, then parsed, validated and only then loaded for an execution.

Libraries & tools

Libraries:

- `wasm-bindgen` — provides interaction between Rust and functions exported for JS
- `web_sys` — Raw bindings to Web APIs (browser Web page API) built using wasm-bindgen
- `gloo-net` — HTTP requests library for WASM Apps

Tools:

- `wasm-pack` — generates both Rust code and JavaScript code while compiling to WebAssembly.
- `trunk` — WASM web application bundler for Rust.
  Builds application to wasm file and supplies is with a small JS script that loads and runs the wasm.

wasm-bindgen

As we have seen already, manually importing functions from JS to WebAssembly modules is tedious.

Plus functions we have worked with before looked simple only because they operated only with numbers that are native primitive types in WebAssembly.

But if we want to work with strings or other complex types, that are not first class citizens in WebAssembly (same as strings are not present in “native” assembly language), the amount of boilerplate code will be increased dramatically.

`wasm-bindgen` crate fully takes the burned of:

- making imports and exports
- creating glue code on JS side for converting Rust types to JS types and vice versa
- loading and compiling wasm files

Example 5: wasm-bindgen – console log

Create a new project

```
cargo new mybindgen --lib
```

Add wasm-bindgen dependency

```
cargo add wasm-bindgen
```

The Cargo.toml file should be like:

Cargo.toml

```
[package]
name = "mybindgen"
version = "0.1.0"
edition = "2024"

[lib]
crate-type = ["cdylib"]

[dependencies]
wasm-bindgen = "0.2"
```

We will also need wasm-pack, that can be installed using command:

```
cargo install wasm-pack
```

Now let's edit src/lib.rs:

- add binding for JS function console.log
- create an entry point that prints "Hello from WebAssembly"
- create a hello_name function, that can be called from JS

src/lib.rs

```
1  use wasm_bindgen::prelude::*;
2
3  #[wasm_bindgen]
4  extern "C" {
5      #[wasm_bindgen(js_namespace = console)]
6      fn log(a: &str);
7  }
8
9  #[wasm_bindgen(start)]
10 fn startup_hello() {
11     log("Hello from WebAssembly");
12 }
13
14 #[wasm_bindgen]
15 pub fn hello_name(name: &str) {
16     log(format!("Hello, {name}!").as_str());
17 }
```

Now we can build this by calling:

```
wasm-pack build --target web
```

This will create a pkg folder with a lot of files, but we are interested only in two of them:

- `pkg/mybindgen.js` – JS module that loads WebAssembly, runs entry point and exports functions
- `pkg/mybindgen_bg.wasm` – wasm file built of `src/lib.rs`

Now we just need an index.html file that runs the application.

Because of wasm-bindgen and wasm-pack, this time index.html will be much simpler:

index.html

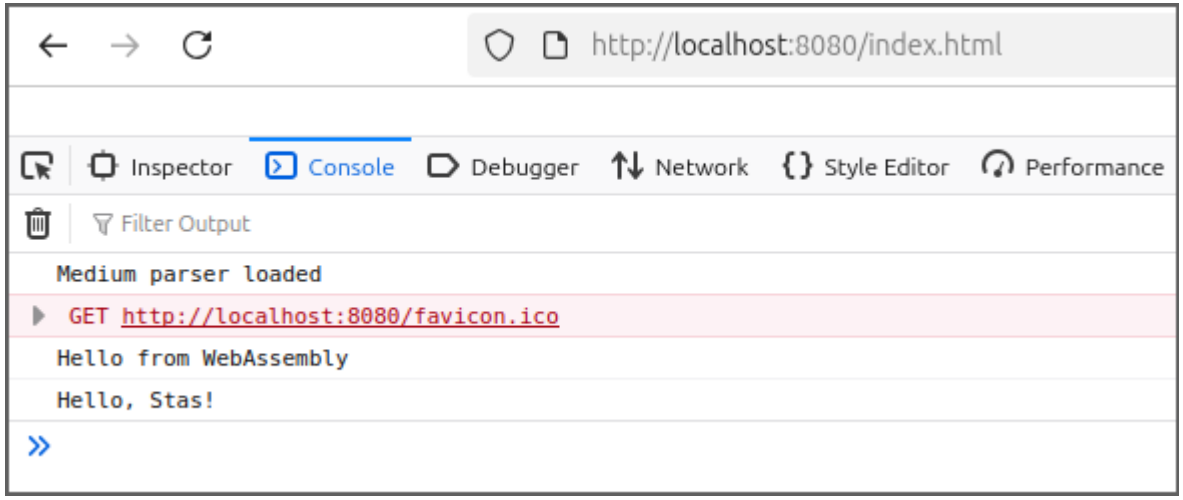
```
<!doctype html>
<html>
</html>

<script type="module">
    import init, { hello_name } from "./pkg/mybindgen.js";
    await init();
    hello_name("Stas");
</script>
```

The project files layout should look like:

```
/
├─ Cargo.toml
├─ src/lib.rs
├─ index.html
└─ pkg
   ├─ mybindgen.js
   ├─ mybindgen.d.ts
   ├─ mybindgen_bg.js
   ├─ mybindgen_bg.wasm
   ├─ mybindgen_bg.wasm.d.ts
   └─ package.json
```

Now we can run the server and check the result in the browser:

```
simple-http-server --cors -i -p 8080 `pwd`
```



Example 6: wasm-bindgen and web-sys – DOM manipulation

Now let’s make another wasm module that uses DOM API (imported via web-sys) to build an HTML page.

Create a new project

```
cargo new wasmdom --lib
```

This time, in addition to wasm-bindgen library, we also need web-sys which is a collection binding for interacting with window and DOM API.

Cargo.toml

```
[package]
name = "wasmdom"
version = "0.1.0"
edition = "2024"

[lib]
crate-type = ["cdylib"]

[dependencies]
wasm-bindgen = "0.2"
web-sys = {version = "0.3", features = ["Document","Element","HtmlElement","Node","Window"] }
```

The code of the module looks very similar to the equivalent JS code, thanks to web-sys.

src/lib.rs

```rust
use wasm_bindgen::prelude::*;

#[wasm_bindgen(start)]
fn run() -> Result<(), JsValue> {
    let window = web_sys::window().unwrap();
    let document = window.document().unwrap();
    let body = document.body().unwrap();

    let h1 = document.create_element("h1")?;
    h1.set_text_content(Some("Hello from WebAssembly"));
    body.append_child(&h1)?;

    Ok(())
}
```

index.html is very similar to the previous example, even though we now use DOM API:

index.html

```html
<!doctype html>
<html>
</html>

<script type="module">
    import init from "./pkg/wasmdom.js";
    await init();
</script>

```

Now we can build the wasm package

```
wasm-pack build --target web
```

and launch the server

```
simple-http-server --cors -i -p 8080 `pwd`
```

Rust Web UI frameworks

Most popular frameworks are:

- **`Leptos`** – Inspired by Solid.js.
  Provides both CSR (client side rendering) and SSR (server side rendering) with Axum or Actix-web.
  Can run as desktop application via Tauri (library for launching web applications using WebKit).
  Has a great documentation. Easy to learn and easy to use.

- **`Dioxus`** – Inspire by React.
  Out of the box can be built for CSR, SSR and as a desktop, IOS and Android application.
  Very fast.

- **`egui`** – A non-HTML Web UI library.
  Unlike Leptos and Dioxus, egui doesn't use HTML DOM to build a UI. Instead it creates just a canvas, and then uses WebGL API to render the UI. It is very similar to how traditional desktop GUI libraries (like GTK and Qt) work.

More Rust GUI libraries: https://github.com/flosse/rust-web-framework-comparison

Leptos

Leptos stands on two concepts:

- Component – composable function that build DOM sub-tree.
- Signal – reactive “variable” that on update automatically re-renders components that use this signal.

Example 7: Leptos – Hello World

```
hello-leptos
  ├─ index.html
  ├─ Cargo.toml
  └─ src
      └─ main.rs
```

index.html

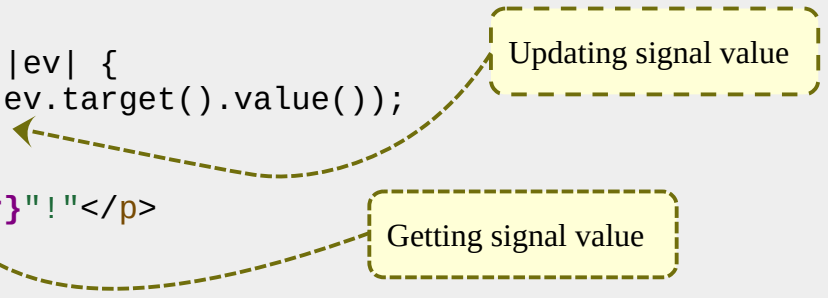
```
<!DOCTYPE html>
<html>
  <head></head>
  <body></body>
</html>
```

Cargo.toml

```
[package]
name = "hello-leptos"
version = "0.1.0"
edition = "2024"

[dependencies]
leptos = { version = "0.8.2", features = ["csr", "tracing"] }
```

main.rs

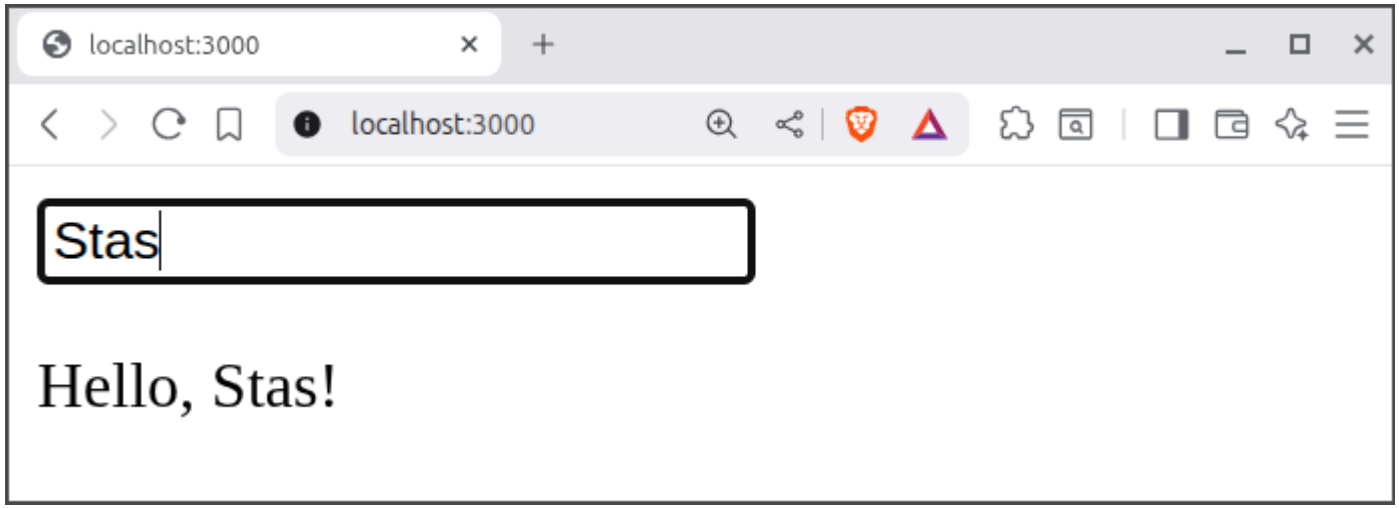
```
1  use leptos::prelude::*;
2
3  fn main() {
4      leptos::mount::mount_to_body(App)
5  }
6
7  #[component]
8  fn App() -> impl IntoView {
9      let (input_getter, input_setter) = signal::<String>("".to_string());
10
11     view! {
12         <input type="text"
13             on:input:target=move |ev| {
14                 input_setter.set(ev.target().value());
15             }
16         />
17         <p>"Hello, "{input_getter}"!"</p>
18     }
19 }
```



To launch the Web UI in developer mode we use command:

```
trunk serve --port 3000 --open
```

It runs developer server on port 3000 that provides the Web UI.



You need to have trunk and wasm32 target installed. You can do it by executing commands:

```
cargo install trunk
rustup target add wasm32-unknown-unknown
```

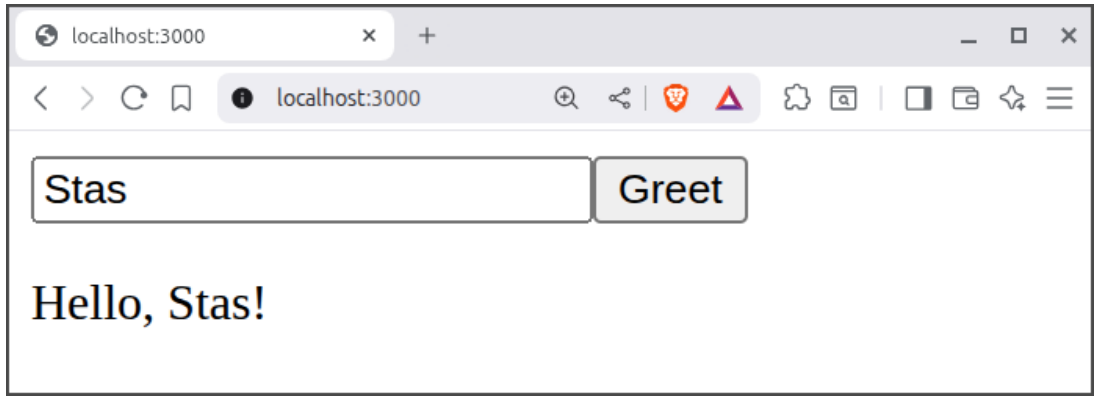
Example 8: Leptos – Composing components

Component view can include other components by specifying them as tags.

```rust
use leptos::prelude::*;

fn main() {
    leptos::mount::mount_to_body(App)
}

#[component]
fn App() -> impl IntoView {
    let (input_getter, input_setter) = signal::<String>("".to_string());
    view! {
        <NameInput setter=input_setter />
        <DisplayHellow getter=input_getter />
    }
}

#[component]
fn NameInput(setter: WriteSignal<String>) -> impl IntoView {
    view! {
        <input type="text"
            on:input:target=move |ev| {
                setter.set(ev.target().value());
            }
        />
    }
}

#[component]
fn DisplayHellow(getter: ReadSignal<String>) -> impl IntoView {
    view! {
        <p>"Hello, "{getter}"!"</p>
    }
}
```

Example 9: Leptos – Handling button click

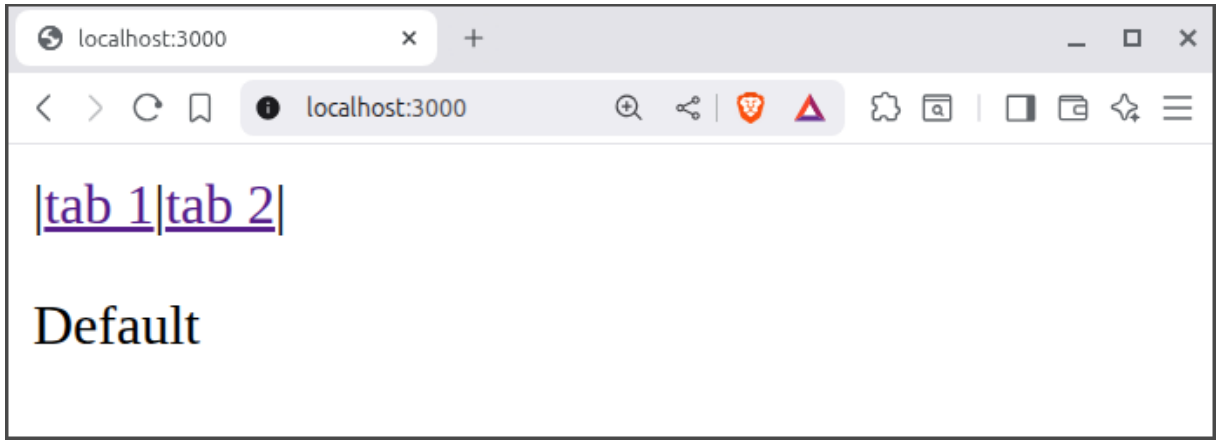
```rust
use leptos::prelude::*;

fn main() {
    leptos::mount::mount_to_body(App)
}

#[component]
fn App() -> impl IntoView {
    let input_sig = RwSignal::new("".to_string());
    let hello_str_sig = RwSignal::new("".to_string());

    view! {
        <input type="text" bind:value=input_sig />

        <button
            on:click=move |_| hello_str_sig.set(format!("Hello, {}!", input_sig.get()))
        >Greet</button>

        <p>{hello_str_sig}</p>
    }
}
```

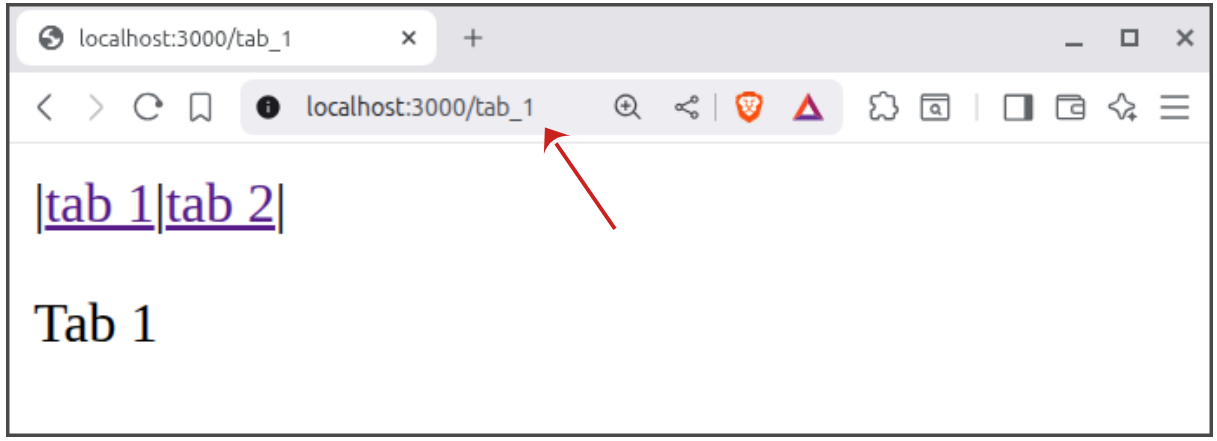


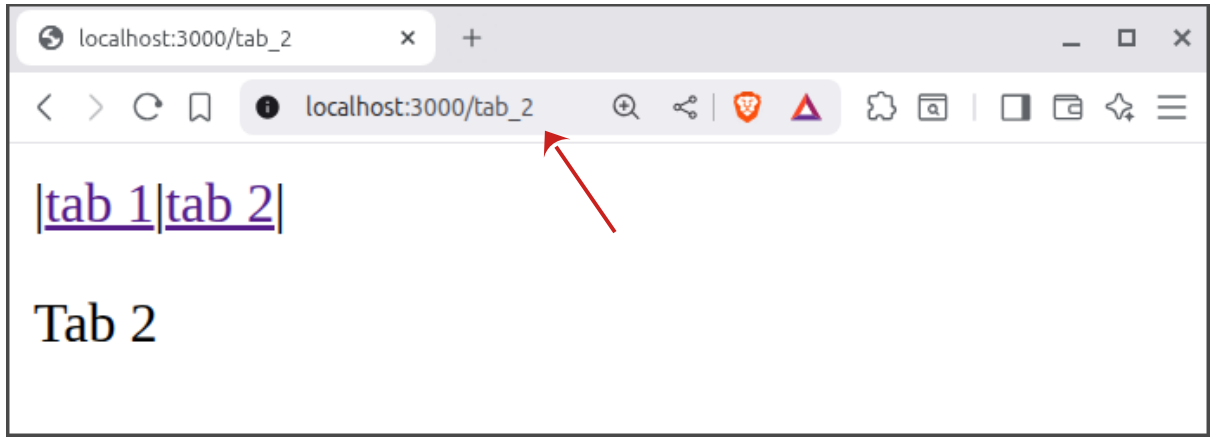
Example 10: Leptos – Routing

```
use leptos::prelude::*;
use leptos_router::{components::{Route, Router, Routes}, path};

fn main() {
    leptos::mount::mount_to_body(App)
}

#[component]
fn App() -> impl IntoView {
    view! {
        <Router>
            <nav>
                | <a href="/tab_1">tab 1</a> | <a href="/tab_2">tab 2</a> |
            </nav>
            <main>
                <Routes fallback=|| "Not found.">
                    <Route path=path!("/")      view=|| view! { <p>Default</p> } />
                    <Route path=path!("/tab_1") view=Tab1 />
                    <Route path=path!("/tab_2") view=Tab2 />
                </Routes>
            </main>
        </Router>
    }
}

#[component]
fn Tab1() -> impl IntoView {
    view! { <p>Tab 1</p> }
}

#[component]
fn Tab2() -> impl IntoView {
    view! { <p>Tab 2</p> }
}
```

(Need to add Cargo dependency for: leptos_router = { version = "0.8.2" })


localhost:3000
localhost:3000
|tab 1|tab 2|
Default

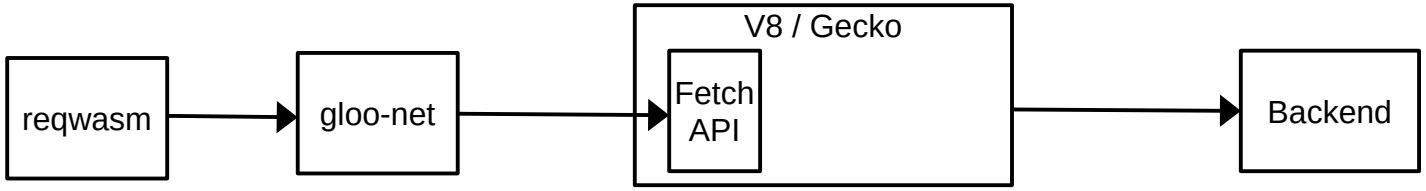

localhost:3000/tab_1
localhost:3000/tab_1
|tab 1|tab 2|
Tab 1


localhost:3000/tab_2
localhost:3000/tab_2
|tab 1|tab 2|
Tab 2

Example 11: Leptos – HTTP calls

For making HTTP calls, JS provides fetch API.

gloo-net wraps this API to make it available from Rust code (see https://github.com/ranile/gloo/blob/master/crates/net/src/http/request.rs#L26).



Let’s look at a small example of a Web UI that taken a name from an input field, and sends it via HTTP to a back-end endpoint, that returns a greeting string for the given name.

Test back-end:

```
hello-axum
 ├─ Cargo.toml
 └─ src
     └─ main.rs
```

Cargo.toml

```
[package]
name = "axum-text"
version = "0.1.0"
edition = "2024"

[dependencies]
tokio = { version = "1", features = ["full"] }
axum = "0.8.4"
tower-http = { version = "^0.6", features = ["cors"] }
```

main.rs

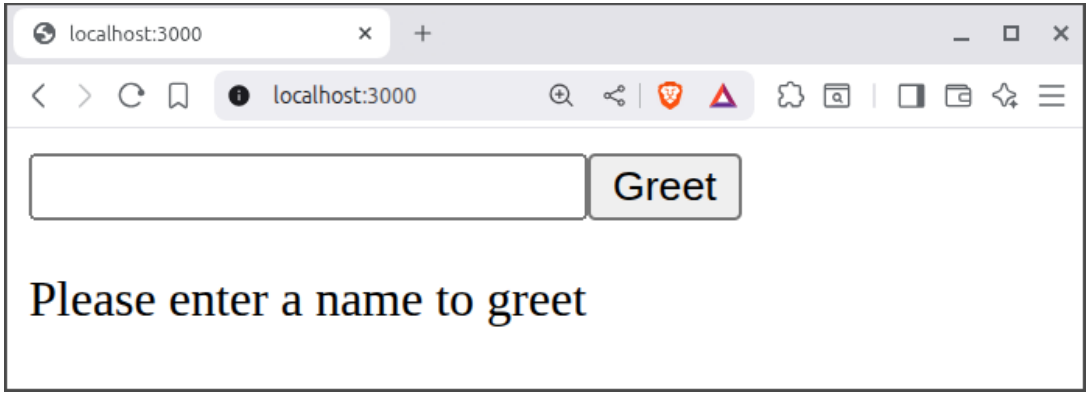
```rust
use std::net::SocketAddr;

use axum::{extract::Path, routing::{get}, Router};
use tokio::net::TcpListener;
use tower_http::cors::CorsLayer;

#[tokio::main]
async fn main() {
    let addr = SocketAddr::from(([0, 0, 0, 0], 8080));

    let app = Router::new()
        .route("/api/hello/{name}", get(greet))
        .layer(CorsLayer::permissive());

    axum::serve(TcpListener::bind(addr).await.unwrap(), app)
        .await
        .unwrap();
}

async fn greet(Path(name): Path<String>) -> String {
    format!("Hello, {name}!")
}
```

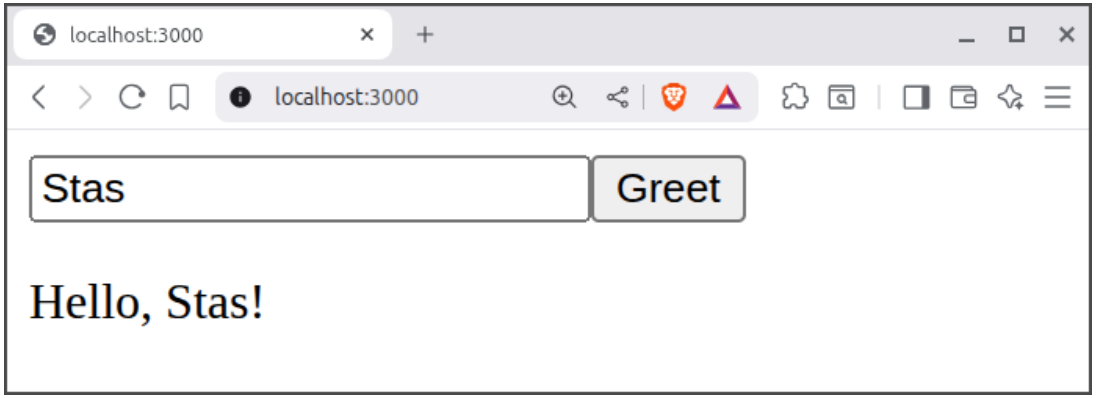
Front-end:

Cargo.toml

```
[package]
name = "frontend"
version = "0.1.0"
edition = "2024"

[dependencies]
leptos = { version = "0.8.2", features = ["csr"] }
gloo-net = "0.6"
```

```rust
use leptos::prelude::*;

fn main() {
    leptos::mount::mount_to_body(App)
}

#[component]
fn App() -> impl IntoView {
    let input_sig = RwSignal::new("".to_string());
    let (name_getter, name_setter) = signal("".to_string());
    let gist_text = LocalResource::new(move || hello(name_getter.get()));

    view! {
        <input type="text" bind:value=input_sig />
        <button on:click=move |_| name_setter.set(input_sig.get())>Greet</button>
        <p>
            {move || {
                match gist_text.get().unwrap_or_default() {
                    Some(greetings) => greetings,
                    None => "Please enter a name to greet".to_string(),
                }
            }}
        </p>
    }
}

async fn hello(name: String) -> Option<String> {
    if name.is_empty() { return None };
    // requires dependency gloo-net = "0.6"
    let response = gloo_net::http::RequestBuilder::new(&format!(
        "http://localhost:8080/api/hello/{name}",
    ))
    .method(gloo_net::http::Method::GET)
    .send().await.unwrap()
    .text().await.unwrap();
    Some(response)
}
```


localhost:3000
localhost:3000
Greet
Please enter a name to greet


localhost:3000
localhost:3000
Stas
Greet
Hello, Stas!

Crates usage

What Rust libraries (crates) can be used in Web UI application?

- Crates that provide any functionality that doesn't use I/O (collections, parsers & formatters, macroses) can be used absolutely seamlessly.
- Crates that do have I/O can be used only if they can be used in browser (e.g. because of security reasons, browsers are limit is accessing the file system), and if they have proper support for wasm32.

E.g. implementation of now() function from the *chrono* crate:

```
#[cfg(not(all(
    target_arch = "wasm32", feature = "wasmbind",
    not(any(target_os = "emscripten", target_os = "wasi", target_os = "linux"))
)))]
#[must_use]
pub fn now() -> DateTime<Utc> {
    let now = SystemTime::now().duration_since(UNIX_EPOCH).expect("system time before Unix epoch");
    DateTime::from_timestamp(now.as_secs() as i64, now.subsec_nanos()).unwrap()
}


#[cfg(all(
    target_arch = "wasm32", feature = "wasmbind",
    not(any(target_os = "emscripten", target_os = "wasi", target_os = "linux"))
))]
#[must_use]
pub fn now() -> DateTime<Utc> {
    let now = js_sys::Date::new_0();
    DateTime::<Utc>::from(now)
}
```

Links

WebAssembly core specification: https://webassembly.github.io/spec/core/

WebAssembly specification documents: https://github.com/WebAssembly/spec

Official Leptos web site: https://leptos.dev/

Official Leptos book: https://book.leptos.dev/

Leptos examples: https://github.com/leptos-rs/leptos/tree/main/examples