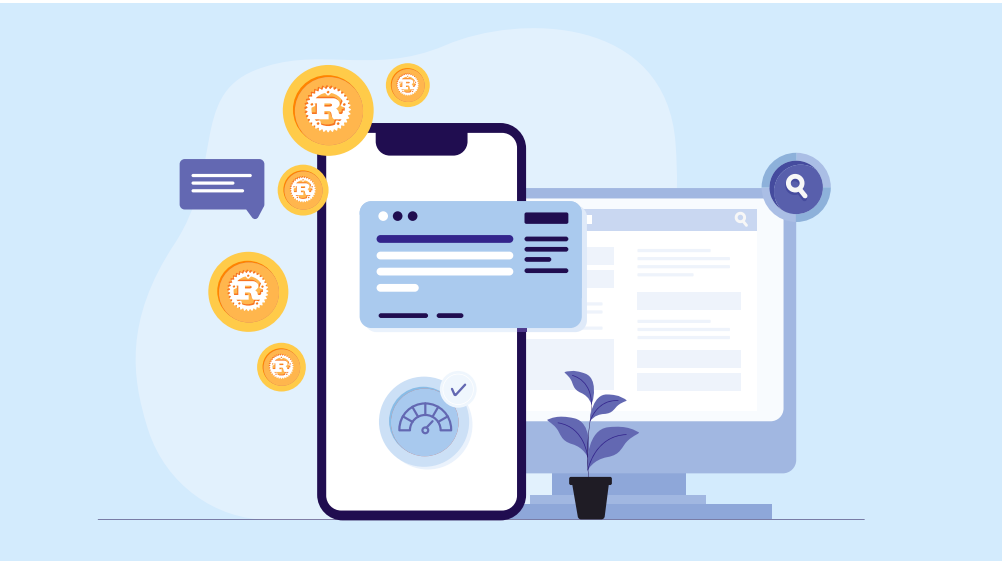


Web-Apps mit Rust und WebAssembly

Rust-Code lässt sich einfach in WebAssembly übersetzen und so im Browser ausführen. Das GUI kann man dabei entweder in JavaScript schreiben oder als HTML-Elemente in den Rust-Code einbetten oder über eine GUI-Bibliothek für Rust erstellen.

Von Gerhard Völkl

Ein Pluspunkt der Programmiersprache Rust ist die sichere Ausführung zur Laufzeit. Übersetzt der Rust-Compiler ein Programm ohne Fehlermeldung, sind Abstürze sehr unwahrscheinlich. Bei anderen Sprachen und Frameworks kommen sie weit häufiger vor.

Browser können kompilierten binären Code nicht ausführen. Was aber aktuelle Browser verarbeiten können, ist WebAssembly, ein maschinenunabhängiger Zwischencode, den der Rust-Compiler erzeugt (siehe Kasten „Der W3C-Standard WebAssembly“).

X-TRACT

- Die Kombination von Rust und WebAsssembly bringt Vorteile bei Leistung, Integration und Sicherheit von Webanwendungen.
- Rust-Code lässt sich in WebAssembly kompilieren und somit in Webanwendungen integrieren.
- WebAssembly ist plattformunabhängig und dort ausführbar, wo es einen kompatiblen Webbrowser gibt.
- Mit Rust gibt es verschiedene Wege, eine Webanwendung zu erstellen. Eine einfache Temperaturumrechner-App zeigt die diversen Möglichkeiten.

Viele Wege führen zum Ziel

Wer mit Rust eine Webanwendung erstellen will, hat die Auswahl zwischen drei grundsätzlichen Herangehensweisen, je nach Ausgangslage und Zielstellung des Projekts:

- JavaScript kümmert sich um die Oberfläche und die Interaktion. Es lagert vor allem komplexe Berechnungen an Rust aus, um eine bessere Performance zu erreichen.
- Die ganze Anwendung ist in Rust geschrieben, wobei Rust Webtechnologien wie HTML, CSS oder Ähnliches verwendet.
- Es existiert bereits eine Anwendung mit einer betriebssystemunabhängigen Oberfläche in Rust. Es ist möglich, diese Applikation in WebAssembly neu zu kompilieren und geringfügig anzupassen.

Der Artikel setzt eine einfache Anwendung in drei Beispielen nach diesen drei Grundprinzipien um. Daran ist zu sehen, wie sich das auf den Programmaufbau auswirkt und welche Werkzeuge verwendet werden. Die einfache Beispielanwendung rechnet die Temperatur von Celsius in Fahrenheit um und umgekehrt (Abbildung 2). Die Idee dazu stammt von Eugen Kiss, der sieben Aufgaben definiert hat, um GUI-Applikationen zu erstellen (siehe ix.de/z4zz). Jede Aufgabe wird immer schwieriger – vergleichbar mit den Levels in einem Videospiel. Bei allen drei Umsetzungen kommen Bibliotheken und Frameworks zum Einsatz, die bei der praktischen Arbeit sehr viel Unterstützung liefern.

Der Temperaturumrechner, Aufgabe 2, besteht aus zwei Eingabefeldern: einem für die Celsiustemperatur und einem für die Fahrenheittemperatur. Schreibt jemand einen numerischen Wert in das Celsiusfeld, aktualisiert das Programm den entsprechenden Wert im Fahrenheitfeld und umgekehrt. Bei nicht numerischen Werten ändert sich nichts.

Unabhängig davon, welche Werkzeuge bei der Webanwendung zum Einsatz kommen, ist der Rust-Compiler immer mit im Spiel. Rust kann Anwendungen für andere Plattformen erzeugen – ganz gleich, auf welchem Betriebssystem der Compiler läuft. Für die Übersetzung in WebAssembly kann man einfach eine weitere Zielplattform beim Rust-Compiler hinzufügen:

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

Das wird häufig vom verwendeten Framework übernommen; Entwickler brauchen es nicht explizit anzugeben.

Rust und JavaScript zusammen einsetzen

In der ersten Version des Beispielprogramms Temperaturumrechner erledigt JavaScript die Hauptarbeit wie Oberfläche und Anwenderkommunikation. Die in Rust zu erstellende WebAssembly-Bibliothek stellt die Umrechnungsfunktionen zur Verfügung:

```
cargo new --lib rust-wasm-task-2
```

Der Befehl erzeugt ein Rust-Projekt, das als Ergebnis eine WebAssembly-Library

rust_wasm_task_2.wasm zum Einbinden in JavaScript liefert. Der zugehörige Sourcecode ist in der Datei src/lib.rs zu erstellen (Listing 1).

Wenn Rust auf JavaScript-Funktionen zugreifen will, hat es sich an bestimmte Aufrufformate zu halten. Genauso kann JavaScript nicht so einfach Rust-Funktionen in WebAssembly verwenden. Dank der Rust-Bibliothek wasm-bind kann man Rust so programmieren, wie man es gewohnt ist. Um den Rest kümmert sich die Bibliothek:

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

Will Rust mit einer JavaScript-Funktion wie console.log arbeiten, definiert es sie als externe Funktion:

```
#[wasm_bindgen]
extern {
    #[wasm_bindgen(js_↵
                   namespace=console)]
    pub fn log(s: &str); ↵
                // import console.log
}
```

Vor dem Befehlswort `extern` steht das Attribut `#[wasm_bindgen]`. Dies stellt sicher, dass sich ein prozedurales Makro um die Anbindung an JavaScript kümmert. Da console.log keine globale Funktion im Browser ist, sondern im Namensraum `console` liegt, braucht `wasm_bindgen` den Hinweis

```
js_namespace=console
```

Dann kann Rust die Funktion `log` wie jede andere Funktion verwenden.

Namenskonventionen der Sprachen im Blick

Rust-Funktionen, die umgekehrt für Aufrufe in JavaScript zur Verfügung stehen sollen, müssen Entwicklerinnen und Entwickler ebenfalls mit dem Attribut `#[wasm_bindgen]` versehen. Wollen sie die unterschiedlichen Namenskonventionen der verschiedenen Programmsprachen berücksichtigen, können sie mit

```
js_name = toFahrenheit
```

die Funktion für JavaScript umbenennen.

Die Steuerdatei Cargo.toml definiert, wie Rust in WebAssembly umzusetzen ist (Listing 2). Unter dem Abschnitt `lib` steht, welche Art von Bibliothek zu erstellen ist. Der Typ `cdylib` bedeutet normalerweise, dass der Compiler eine dynamische Systembibliothek erstellt: in Microsoft Windows mit der Endung .dll, auf macOS oder Linux mit der Endung .so. Für WebAssembly bedeutet es jedoch einfach: Erstelle eine Wasm-Datei ohne die Funktion `main`. Die einzige Bibliothek, die das Beispielprogramm verwendet, ist wasm-bindgen.

Mit wasm-pack Schritte automatisieren

Damit das Arbeiten mit Rust und WebAssembly möglichst komfortabel ist, braucht man neben den Rust-Standardwerkzeugen wie Rustup und Cargo weitere Komponenten. Häufig ist das Werkzeug wasm-pack die erste Wahl beim Umgang mit WebAssembly:

```
cargo install wasm-pack
```

Die fertig programmierte Bibliothek in Rust setzt wasm-pack in WebAssembly um:

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

Als Erstes startet wasm-pack den Rust-Compiler, der den WebAssembly-Code

Der W3C-Standard WebAssembly

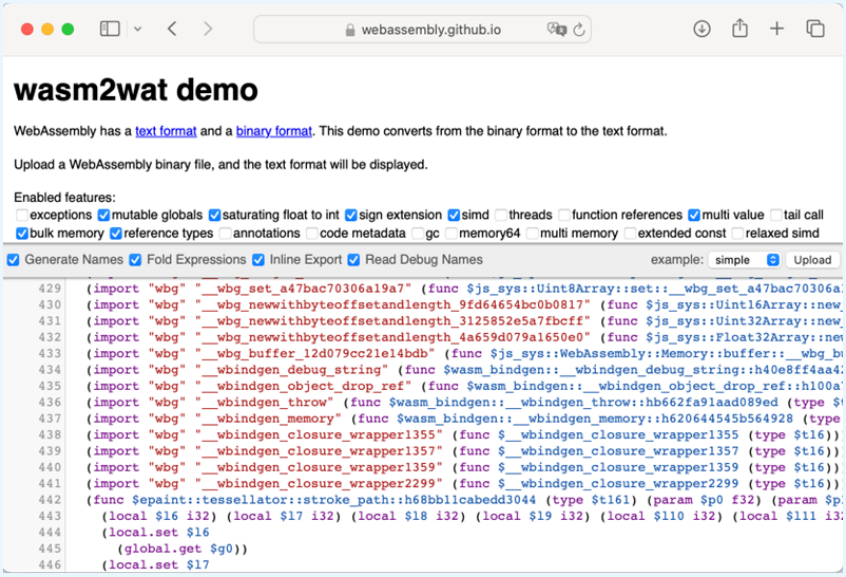
WebAssembly-Programme liegen als Binärdateien vor – häufig mit der Endung .wasm. Eine WebAssembly-Runtime interpretiert den Inhalt der Wasm-Datei und führt sie aus. Wer sich den WebAssembly-Code genauer ansehen möchte, kann ihn sich in einem Textformat ausgeben lassen, etwa über wasm2wat (Abbildung 1).

Diese Vorgehensweise hat zwei Vorteile: Portabilität und Sicherheit. Portabilität heißt in diesem Fall: Überall, wo eine WebAssembly-Runtime vorhanden ist, läuft ein WebAssembly-Programm unverändert. Jeder aktuelle Webbrowser ist mit einer solchen Runtime ausgestattet. Darüber hinaus gibt es weitere Programme, die auf unterschiedlichen Plattformen eine Runtime enthalten, etwa Node.js, Wasmtime, Wasmer und weitere.

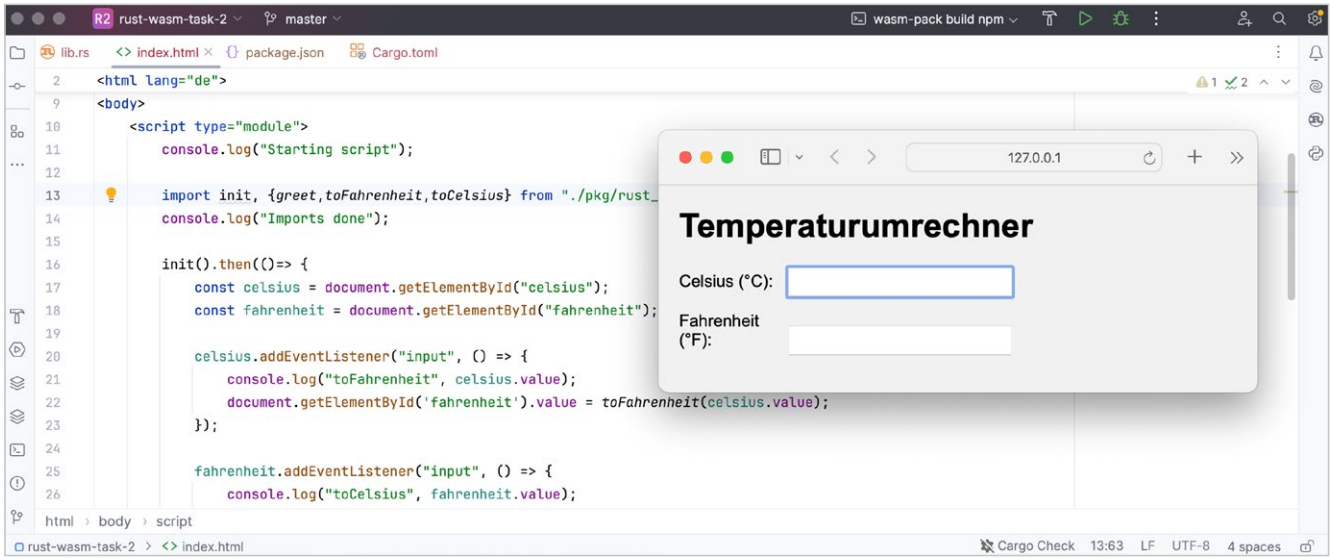
Der Sicherheit kommt zugute, dass ein WebAssembly-Programm nie direkt auf das Betriebssystem zugreift. Es verwendet ausschließlich die Funktionen der Runtime (Sandboxing) und ist damit abgeschottet. Die Runtime lässt sich administrieren; dort kann man festlegen, was dem WebAssembly-Programm erlaubt wird.

Will ein WebAssembly-Programm etwas ausgeben, muss die Runtime eine entsprechende Funktion zur Verfügung stellen. Ein Browser enthält bereits viele Funktionen, aber bei größeren Programmen wäre das umständlich. Daher sind einige Leute auf die Idee gekommen, eine Standardschnittstelle für WebAssembly zur Verfügung zu stellen: WebAssembly System Interface (WASI).

Viele Runtimes wie Wasmtime oder Wasmer implementieren WASI, doch viele Browser tun sich damit schwer. Es gibt einige Workarounds, aber derzeit ist keiner bekannt, der diese Schnittstelle direkt unterstützt. WASI ist aber ein offener Standard der WASI Subgroup, eines Teils der W3C WebAssembly Community Group. Deshalb stehen die Chancen gut, dass sich das künftig verbessert.



Werkzeuge wie wasm2wat setzen Wasm-Dateien in Text-Dateien (WAT) um, damit die Inhalte lesbar sind (Abb. 1).



Webanwendungen mit Rust durch das Kompilieren in WebAssembly zu erzeugen, ist eine Möglichkeit für einfache, aber auch komplexere Applikationen (Abb. 2).

erzeugt. Darüber hinaus erstellt es einen Ordner pkg, in dem es verschiedene Dateien ablegt, die die Anbindung an JavaScript einfacher machen:

- rust_wasm_task_2_bg.wasm ist die erstellte WebAssembly-Bibliothek;
- rust_wasm_task_2.js enthält JavaScript-Funktionen, mit denen sich die Wasm-Bibliothek relativ einfach anbinden lässt;
- package.json beinhaltet alle Beschreibungen für npm oder ähnliche Paketierwerkzeuge;
- rust_wasm_task_2.d.ts erlaubt eine Anbindung an TypeScript.

Bevor JavaScript die Funktionen aus der WebAssembly-Bibliothek nutzen kann, muss es sie erst einmal einbinden (Listing 3):

```
import init, {greet,toFahrenheit,↵
   toCelsius} from "./pkg/rust_wasm_↵
                          task_2.js"
```

Der JavaScript-Befehl `import init` importiert die Standardexportfunktion (in diesem Fall `_ _wbg_set_wasm`) aus der JavaScript-Datei, die das Werkzeug wasm-pack erstellt hat. Diese Funktion initialisiert und lädt das WebAssembly-Modul. Alle weiteren Funktionen in geschweiften Klammern sind die mit Rust erstellten und somit für JavaScript externe Funktionen:

```
init().then(()=> { … }
```

Listing 1: Kommunikation zwischen Rust und JavaScript (rust-wasm-task-2/lib.rs)

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

#[wasm_bindgen]
extern {
    pub fn alert(s: &str);
}

#[wasm_bindgen]
extern {
    #[wasm_bindgen(js_namespace=console)]
    pub fn log(s: &str); // import console.log
}

#[wasm_bindgen]
pub fn greet(name: &str) {
    log("greet");
    alert(&format!("Hallo, {}",name));
}

#[wasm_bindgen(js_name = toFahrenheit)]
pub fn to_fahrenheit(celsius: &str)->String {
    log(celsius);
    if let Ok(celsius_value) = celsius.parse::<f32>() {
        let fahrenheit = (celsius_value * 9.0 / 5.0) + 32.0;
        format!("{:.2}", fahrenheit)
    } else {
        String::from("")
    }
}

#[wasm_bindgen(js_name = toCelsius)]
pub fn to_celsius(fahrenheit: &str)->String {
    if let Ok(fahrenheit_value) = fahrenheit.parse::<f32>() {
        let celsius = (fahrenheit_value - 32.0) * 5.0 / 9.0;
        format!("{:.2}", celsius)
    } else {
        String::from("")
    }
}
```

Listing 2: Cargo.toml für rust-wasm-task-2

```
[package]
name = "rust-wasm-task-2"
version = "0.1.0"
edition = "2021"

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

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

Listing 3: WebAssembly in JavaScript verwenden

```
<!DOCTYPE html>
<html lang="de">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Temperaturumrechner</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <script type="module">
        console.log("Starting script");

        import init, {greet,toFahrenheit,toCelsius} from "./pkg/rust_wasm_task_2.js"
        console.log("Imports done");

        init().then(()=> {
            const celsius = document.getElementById("celsius");
            const fahrenheit = document.getElementById("fahrenheit");

            celsius.addEventListener("input", () => {
                document.getElementById('fahrenheit').value = toFahrenheit(celsius.value);
            });

            fahrenheit.addEventListener("input", () => {
                document.getElementById('celsius').value = toCelsius(fahrenheit.value);
            });

            greet("Rust");
        }).catch((error) => {
            console.error("Error initializing WebAssembly module:", error);
        });

    </script>
    <h1>Temperaturumrechner</h1>
    <div class="input-container">
        <label for="celsius">Celsius (°C):</label>
        <input type="text" id="celsius">
    </div>
    <div class="input-container">
        <label for="fahrenheit">Fahrenheit (°F):</label>
        <input type="text" id="fahrenheit">
    </div>
</body>
</html>
```

Nachdem die Funktion `init` ausgeführt ist, stehen JavaScript alle weiteren Funktionen zur Verfügung. Beispielsweise startet die Funktion `toFahrenheit` im Browser immer dann, wenn sich etwas im entsprechenden Eingabefeld ändert.

Eine Webanwendung mit Rust, aber ohne JavaScript

Die zweite Variante des Temperaturumrechners verwendet überhaupt kein JavaScript mehr. Die Oberfläche ist HTML und alles andere WebAssembly-Code, den der Browser ausführt. Um das ganze Drumherum wie die Kommunikation mit der Browser-API brauchen sich Programmierer nicht zu kümmern. Das übernimmt ein Rust-Webframework.

Davon gibt es einige. Dieser Artikel verwendet Dioxus, dessen Vorbild das JavaScript-Framework React ist. Dioxus bringt sein eigenes Werkzeug `dx` mit, das vom Erstellen von WebAssembly bis zur Ausführung im eigenen HTML-Server alles übernimmt. Der Befehl

```
cargo install dioxus-cli
```

installiert das Werkzeug. Ein neues Projekt erzeugt man mit

```
dx new
```

`dx` fragt dabei einige Parameter ab: zunächst, welche Art von Applikation man erstellen möchte. Dioxus kann WebView-Applikationen erstellen, die direkt auf einem Rechner oder einem mobilen Gerät laufen. Alternativ lässt sich eine Web-App für den Browser erzeugen, was für das Beispielprogramm am einfachsten ist. Dabei führt Dioxus alles per WebAssembly im Browser aus. Eine weitere Variante ist eine Serverkomponente, die HTML erzeugt und es an den Client überträgt.

Mit weiteren Parametern beim Erstellen eines neuen Projektes gibt man an, ob man mit CSS arbeiten und ob das Rust-Programm den Dioxus-Router verwenden soll. Überdies führt `dx` das fertige Programm aus:

```
dx serve
```

Rust trifft auf React

Die Philosophie von Dioxus ist dieselbe wie bei React: Eine Webapplikation setzt sich aus Komponenten zusammen. Eine Komponente kann wiederum aus anderen Komponenten bestehen. Diese erhalten Input und machen daraus HTML.

Der Temperaturumrechner in Dioxus enthält zunächst die Komponente `App`, die für die gesamte Webanwendung steht (Listing 4). Sie beinhaltet die Komponente `TemperatureCalculator`, die die

Weitere Artikel zum Thema JavaScript-Alternativen

Elm als Java-Script-Alternative
(iX 1/2024, S. 107)

PyScript: Python im Browser
(iX 2/2024, S. 120)

Listing 4: Web-App mit dem Framework Dioxus

```
use dioxus::prelude::*;

#[derive(PartialEq, Clone, Props)]
struct TemperatureInputProps {
    label: String,
    valuecontext: Signal<String>,
    oninput: EventHandler<FormEvent>,
}

pub fn TemperatureInput(props: TemperatureInputProps ) -> Element {
    let mut temp_value= props.valuecontext;

    rsx! {
        div{
            class: "input-container",
            label {
                display: "inline-block",
                width: "150px",
                "{props.label}"
            }
            input{
                font_size: "20px",
                padding: "5px",
                width: "200px",
                r#type: "text",
                value: "{temp_value}",
                oninput: move |evt| props.oninput.call(evt) ↵
                // oninput: move |event| temp_value.set↵
                                (props.on_update(event.value()))
            }
        }
    }
}

pub fn to_fahrenheit(celsius: &str)->String {
    if let Ok(celsius_value) = celsius.parse::<f32>() {
        let fahrenheit = (celsius_value * 9.0 / 5.0) + 32.0;
        format!("{:.2}", fahrenheit)
    } else {
        String::from("")
    }
}

pub fn to_celsius(fahrenheit: &str)->String {
    if let Ok(fahrenheit_value) = fahrenheit.parse::<f32>() {
        let celsius = (fahrenheit_value - 32.0) * 5.0 / 9.0;
        format!("{:.2}", celsius)
    } else {
        String::from("")
    }
}

pub fn TemperatureCalculator() -> Element {
    //let mut count = use_signal(|| 0);
    let mut celsius_context =  use_context_provider(|| ↵
                                        Signal::new("".to_string()));
    let mut fahrenheit_context =  use_context_provider(|| ↵
                                        Signal::new("".to_string()));

    rsx! {
        div{
            h1 {"Temperaturumrechner"}
            TemperatureInput{
                label:"Celsius (°C)",
                valuecontext: celsius_context,
                oninput: move |event:FormEvent| fahrenheit_↵
                  context.set(to_fahrenheit(event.value().as_str()))
            }
            TemperatureInput{
                label: "Fahrenheit (°F)",
                valuecontext: fahrenheit_context,
                oninput: move |event:FormEvent| celsius_↵
                    context.set(to_celsius(event.value().as_str()))
            }
        }
    }
}

pub fn App() -> Element {

    rsx! {
        div { class:"App",
            background_color: "#f0f0f0", margin: "20px",  font_↵
                                        family: "Arial, sans-serif",
            font_size: "20px",
            TemperatureCalculator {}
        }
    }
}

fn main() {
    launch(App);
}
```

Maßeinheiten umrechnet. Dafür greift sie auf zwei Komponenten vom Typ `TemperatureInput` zurück.

Das Beispielprogramm startet mit der Funktion `launch` die Komponente `App`, die ein Ergebnis des Typs `Element` zurückliefert. Das Element enthält die Ergebnisse. Die eigentliche Arbeit, HTML erstellen, übernimmt der Befehl `rsx!` – in Rust als Makro bezeichnet, da mehr passiert als bei einem einfachen Funktionsaufruf. Es ist das Äquivalent zu `jsx` in React.

Listing 5: egui für das Web anpassen

```
// When compiling to web using trunk:
#[cfg(target_arch = "wasm32")]
fn main() {
    eframe::WebLogger::init(log::LevelFilter::Debug).ok();

    let web_options = eframe::WebOptions::default();

    wasm_bindgen_futures::spawn_local(async {
        eframe::WebRunner::new()
            .start(
                "the_canvas_id", // hardcode it
                web_options,
                Box::new(|cc| Box::new( Task2App::new(cc))),
            )
            .await
            .expect("failed to start eframe");
    });
}
```

Mit dem Makro `rsx!` kann man Oberflächenelemente beschreiben und ihre Attribute und Kindelemente festlegen. Inhaltlich ist es vergleichbar mit HTML, allerdings passt die Syntax mit geschweiften Klammern besser zu Rust:

```
rsx! {
    div{
        label { width: "150px",↵
                        "Temperatur"}
        input{r#type: "text",value:↵
                        "{temp_value}“
        }
    }
}
```

Der Code zeigt eine vereinfachte Eingabe für einen Temperaturwert. Das Element besteht aus einem `div`-Element, das ein `label`- und ein `input`-Element enthält. Jedes Element lässt sich mit Attributen versehen. Beim `label`-Element ist es etwa das Attribut `width:`, das die Breite von `"150px"` festlegt. Vor der schließenden geschweiften Klammer des Elements kommt dessen Wert, in diesem Fall die

Konstante `"Temperatur"`. Der Wert des `input`-Elements kann sich jedoch immer ändern und ist daher in der Rust-Variablen `temp_value` abgelegt. Variablenwerte sind mit `"{value}"` direkt in `rsx` verwendbar.

Um mit den Komponenten zu kommunizieren, übergibt ein Programm Properties (Props) an sie. Das entspricht dem Vorgang, wenn man HTML-Elementen Attribute reicht oder Funktionen Parameter. Die Props einer Komponente definieren Entwickler in einer Rust-Struct. Die Komponente `TemperatureInput` hat drei Props: das anzuzeigende Label, den Wert und das Event, wenn eine Eingabe erfolgt. Dass es sich hier um Props handelt, legt der Wert `Props` in der Anweisung `derive` fest. Welche Props eine Komponente hat, bekommt sie bei der Definition als Parameter übergeben:

```
pub fn TemperatureInput(props:↵
            TemperatureInputProps ) ↵
                    -> Element {..}
```

Dioxus integriert die webspezifischen Eigenschaften möglichst unauffällig in Rust. Entwickler können so in einer Welt bleiben und müssen nicht ständig zwischen Rust und JavaScript hin- und herwechseln.

Eine Rust-Applikation im Web

Der dritte Ansatz mit Rust im Web ist, aus einer bestehenden Anwendung eine Webanwendung zu machen, ohne viel zu ändern. Die Voraussetzung dafür ist, dass die Anwendung möglichst wenig oder gar nicht auf das Betriebssystem zurückgreifen muss, weder bei der Dateiverarbeitung noch bei der Oberfläche. Ideal sind daher Programme, die mit Oberflächenbibliotheken wie egui und Bevy arbeiten. Diese erzeugen die Grafik für die Oberfläche selbst. Daher ist es möglich, sie als Webanwendungen im Browser mit dessen Grafikschnittstelle Web-API auszuführen. Da es für egui bereits Beispiele für die Webumsetzung gibt (siehe ix.de/z4zz), war es relativ einfach, dieses Rust-Programm im Browser auszuführen (Listing 5).

Ausblick

Größere Webprojekte profitieren von modernen, typsicheren Sprachen als Alternative zum allgegenwärtigen JavaScript. Das zeigt nicht zuletzt der Erfolg von TypeScript. Doch bei Rust kommt der Geschwindigkeitsvorteil einer kompilierten Sprache und die Speichersicherheit hinzu. Zusammen mit WebAssembly ist es auch für clientseitige Einsätze hinreichend flexibel. Durch die gezeigten unterschiedlichsten Herangehensweisen beim Erstellen von Webapplikationen mit Rust kann man selbst wählen, wie und mit welchen webspezifischen Techniken man arbeiten möchten. (nb@ix.de)

Quellen

Beispiele für GUI-Aufgaben und Temperaturumrechner mit Rust und egui: ix.de/z4zz

GERHARD VÖLKL

ist Fachjournalist für Softwareentwicklung, Data Science, Spieleprogrammierung und Computergrafik.