Wat is de huidige status van Rust voor het bouwen van webapplicaties?

Branco Bruyneel

28 mei 2022

Woord vooraf

Ter afsluiting van mijn opleiding Media & Creative Technologies aan de Hogeschool West-Vlaanderen te Kortrijk, schreef ik deze bachelorproef. In het vierde semester van deze opleiding koos ik 'AI Engineer' als uitstromingsprofiel. Een semester later, koos ik bij de module 'Research Project' een onderzoeksvraag die niks te maken heeft met Artificiële Intelligentie. De onderzoeksvraag luidt als volgt: „Wat is de huidige status van Rust voor het bouwen van webapplicaties?" en is dan ook het onderwerp voor deze bachelorproef.

Deze bachelorproef bespreekt eerst de research en technische demo uit de module 'Research Project'. De demo bestond uit een 'speed typing' applicatie, waar een gebruiker zo snel mogelijk een willekeurig code fragment moet over typen om erna zijn prestaties te kunnen bekijken.

Na de analyse reflecteer ik het verkregen resultaat met het werkveld. Hiervoor nam ik contact op met een van de core maintainers van Yew genaamd Julius Lungys en mastersstudent industrieel ingenieur Emiel Van Severen. Op basis hiervan volgt een vrijblijvend advies voor bedrijven.

Graag bedank ik Stijn Walcarius die mij de kans gaf om dit onderzoek te mogen uitvoeren sinds deze onderzoeksvraag bestemd was voor een ander uitstromingsprofiel. Waardoor ik met plezier een nieuwe programmeertaal heb ontdekt en het ook meteen mijn favoriete taal is geworden.

Tot slot bedank ik Niek Candaele en Jonas De Meyer voor het nalezen en constructieve feedback op deze bachelorproef.

Abstract

Deze bachelorproef onderzoekt de vraag „Wat is de huidige status van Rust voor het bouwen van webapplicaties?”. Het onderzoek start met de taal Rust en WebAssembly, vervolgens zal er praktisch gekeken hoe we een webapplicatie bouwen en of het productie klaar is.

Voor het technisch onderzoek wordt er een demo applicatie gemaakt die verder inspeelt op de resultaten uit het onderzoek. De demo is een simpele 'speed typing' webapp waarbij de gebruiker een willekeurig code fragment zo snel mogelijk moet over typen.

Om Rust op het web te draaien, wordt het gecompileert naar WebAssembly. De technologie is nog volop in ontwikelling maar toont al veel potentieel. Buiten de browser toont Rust ook uitstekende prestaties waardoor grote bedrijven het 'Rust Foundation' project zijn gaan sponsoren en de taal gebruiken in hun stack.

Rust zal Javascript niet vervangen. Het zal eerder in samenhang gebruikt worden, zodat Rust kan inspelen waar er snelle prestaties worden vereist.

conclusie.

Inhoudsopgave

Lijst van figuren

Lijst met afkortingen

API Application Programming Interface.

CSR Client Side Rendering.

JS Javascript.

NaCI Google Native Client.

SSR Server Side Rendering.

W3C World Wide Web Consortium.

WASI WebAssembly System Interface.

Wasm WebAssembly.

Verklarende woordenlijst

Document Object Model (DOM) Is de gegevensweergave van de objecten die de structuur en inhoud van een document op het web vormen.

Object Relational Mapper (ORM) Een techniek waarmee gegevens uit een database kunnen worden opgevraagd en gemanipuleerd met behulp van een objectgeoriënteerd paradigma.

Hoofdstuk 1

Inleiding

Al 6 jaar op een rij is Rust verkozen tot meest geliefde programmeertaal bij de developers enquête van Stack Overflow. Daarbij verdienen Rust developers ook nog eens de op drie na hoogste salaris en versloegen daarmee Python en Typescript. [1] De taal produceert razendsnelle en veilige code zonder een runtime of garbage collector te gebruiken. De veilige code bereikt Rust door het ownership model waardoor u vele soorten bugs kunt elimineren tijdens het compileren. Tegenwoordig zien we de taal voornamelijk terug in command-line tools, embedded software, networking en WebAssembly.

Naast de prestaties legt het ook de nadruk op productiviteit. De taal komt met geweldige documentatie, een eerste klasse compiler met nuttige foutmeldingen en uitstekende tooling. Zo heeft het een ingebouwde package manager en build tool, slimme multi-editor ondersteuning met auto-completion en type inspections, een auto-formatter en meer. [2]

De twee laatste domeinen genaamd networking en WebAssembly komen aanbod in deze bachelorproef. Er wordt namelijk onderzocht naar de huidige status van Rust voor het bouwen van webapplicaties en doet daarvoor onderzoek naar de volgende deelvragen:

- Wat is Rust?
- Wat is WebAssembly & Hoe werkt het?
- Welke front- & backend frameworks zijn er ter beschikking?
- Hoe bouwt u een webapp in Rust?
- Hoe bouwt u een API in Rust?
- Is Rust productie klaar?

Met de kennis uit het onderzoek wordt er een technisch onderzoek uitgevoerd door een demo applicatie te bouwen. De demo is een simpele 'speed typing' webapp waar bij de gebruiker een willekeurig code fragment zo snel mogelijk moet over typen. Verder wordt er gereflecteerd op het behaalde resultaat met externen uit de praktijk. Uit de reflectie wordt een advies geformuleerd en een conclusie die de onderzoeksvraag beantwoordt.

De methode die in deze studie is gehanteerd, is een gemende aanpak gebaseerd op onderzoek uit verschillende bronnen, praktische resultaten en informatie verkregen uit het werkveld.

Hoofdstuk 2

Research

2.1 Wat is Rust?

Robust, veilig, correct en snel. Met die eigenschappen in zijn achterhoofd bedacht Graydon Hoare in 2006 een nieuwe programmeertaal als hobby project, genaamd Rust. Graydon werkte destijds bij Mozilla die in 2009 interesse toonde in zijn project door het te sponseren. Later in 2010 werd het voor het eerst publiekelijk aangekondig door Mozilla.

Rust is geschreven als een gecompileerde multi-paradigma programmeertaal. Geïnspireerd door de programmeertalen C en C++ met als doel hun pijnpunten op te lossen. Dit wordt gerealiseerd door gebruik te maken van een krachtig typesysteem en een borrow checker. Hiermee kan Rust een hoog niveau van geheugenveiligheid garanderen zonder een garbage collector nodig te hebben. Rust beoogt moderne computersystemen efficiënter te benutten. Hiervoor maakt het onder meer gebruik van het ownership model dat geheugen in een blok toewijst en daarnaast strikt toeziet op de stacktoewijzing. Hierdoor kunnen problemen zoals stackoverflows, bufferoverflows en niet-geïnitialiseerd geheugen voorkomen worden. Verder staat Rust ook geen null-pointers, dangling-pointers of data-races toe in veilige code.

2.1.1 Syntax

Voor velen die zich niet herkennen in programmeertalen zoals C++, Haskell of OCaml lijkt Rust een aparte syntax te hebben in tegenstelling tot conventionele talen. Laat ons even kijken naar een paar syntactische voorbeelden in Rust.

Hier een simpel voorbeeld dat `"Hello world!"` schrijft naar de standaard output.

```
fn main() {
  println!("Hello, world!");
}
```

Listing 1: Hello, world!

Merk op dat `println!` geen functie is maar een macro. Geïnspireerd door de functionele programmeertaal Scheme, zijn macro's een manier van code schrijven dat andere code schrijft, wat bekend staat als metaprogramming. Metaprogramming is handig voor het verminderen van code dat u zelf hoeft te schrijven en onderhouden, wat ook een van de rollen is van functies. Toch verschillen macro's met functies. Zo kunnen macro's een variabel aantal parameters hebben en worden ze uitgebreid vooraleer de compiler de betekenis van de code interpreteert.

```
#[derive(Debug)]
struct Rectangle {
  width: u32,
  height: u32,
}

impl Rectangle {
  fn square(size: u32) -> Rectangle {
    Rectangle: {
      width: size,
      height: size,
    }
  }
  fn area(&self) -> u32 {
    self.width * self.height
  }
}

fn main() {
  let rect1 = Rectangle::square(4);
  println!(
    "The area of the rectangle is {} square pixels.",
    rect1.area()
  );
}
```

Listing 2: structs

Een `struct` in Rust is gelijkaardig aan een `Object` in object georiënteerde programmeertalen. Het wordt gebruikt om samenhorende waarden te groeperen en optioneel kan men associated functions implementeren. Associated functions die geen `self` als hun eerste parameter hebben zijn geen methodes en kunnen gebruikt worden als constructors die een nieuwe instantie retourneren van de `struct`.

Rust heeft geen null pointers tenzij men een dereferentie op een null pointer wil toepassen (dan moet die in een `unsafe` blok worden geplaatst). Als alternatief voor null maakt Rust gebruik van een `Option` type waarmee gekeken kan worden of een pointer wel `Some` of geen `None` waarde bevat. Dit kan afgehandeld worden door syntactische sugar, zoals het `if let` statement om toegang te krijgen tot het innerlijke type, in dit geval een `string`:

```
fn main() {
  let name: Option<String> = None;
  // Als name niet None was, zou het hier geprint worden
  if let Some(name) = name {
    println!("{}", name);
  }
}
```

Listing 3: `Option` type

Naast de `if` en `else` controlestructuren is er ook `match` en `if let`. `match` is vergelijkbaar met een `switch` statement uit andere talen. Het neemt een waarde en test het tegen een serie van patronen. Op basis van welk patroon er overeenkomt wordt de code uitgevoerd. Patronen kunnen opgemaakt worden uit waarden, variabel namen, wildcards, en veel meer. De power van `match` komt van het feit dat de compiler zal bevestigen bij het compileren dat alle mogelijke gevallen zijn afgehandeld. Soms wit u niet alle gevallen expliciet afhandelen en wilt u slechts een patroon afhandelen terwijl u de rest negeert. In dat geval kunt u `if let` gebruiken, wat minder boilerplate code is dan `match`.

```
let message = match maybe_digit {
  Some(x) if x < 10 => process_digit(x),
  Some(x) => process_other(x),
  None => panic!(),
};

let config_max = Some(3u8);
if let Some(max) = config_max {
  println!("The maximum is configured to be {}", max);
}
```

Listing 4: `if let` & `match` operators

2.1.2 Geheugen

Bij vele programmeertalen hoeft u geen zorgen te maken over het geheugen gebruik. Dit is mogelijk door een garbage collector te gebruiken die voortdurend zoekt naar niet langer gebruikt geheugen terwijl het programma aan het uitvoeren is. In andere talen, moet de programmeur het geheugen expliciet toewijzen en vrijmaken. Rust gebruikt geen van beide methodes en komt met een uniek concept genaamd ownership. Hiermee kan het geheugen veiligheid garanderen zonder een garbage collector nodig te hebben. [3]

Vooraleer we verder gaan is het belangrijk dat we de twee begrippen genaamd stack en heap begrijpen. De twee datastructuren maken deel uit van het geheugen en zijn beschikbaar voor uw code om te gebruiken tijdens runtime, maar ze zijn op verschillende manieren gestructureerd. Wat maakt dat de ene zorgt voor snellere dataopslag dan de andere.

Stack

De stack slaat waarden op in de volgorde waarin hij ze krijgt en verwijdert de waarden in de omgekeerde volgorde. Dit wordt aangeduid als last in, first out. Alle gegevens die op de stack worden opgeslagen moeten een bekende, vaste grootte hebben. Gegevens waarvan de grootte op het moment van compileren onbekend is of die van grootte kunnen veranderen, moeten in plaats daarvan op de heap worden opgeslagen.

Heap

De heap is minder georganiseerd: als u gegevens op de heap zet, vraagt u een bepaalde hoeveelheid ruimte aan. De memory allocator vindt een lege plek in de heap die groot genoeg is, markeert die als in gebruik, en geeft een pointer terug, dat is het adres van die locatie. Dit proces wordt 'allocating on the heap' genoemd en wordt soms afgekort als gewoon allocating. Het 'pushen' van waarden op de stack wordt niet beschouwd als allocating. Omdat de pointer naar de heap een bekende, vaste grootte heeft, kunt u de pointer op de stack opslaan, maar als u de eigenlijke gegevens wilt hebben, moet u de pointer volgen.

Toepassing

Het efficiëntste is dus om data naar de stack weg te schrijven dan naar de heap, omdat de allocator nooit hoeft te zoeken naar een plaats om nieuwe gegevens op te slaan. Die plaats is altijd bovenaan de stack. Het alloceren van ruimte op de heap vergt meer werk, omdat de allocator eerst een ruimte moet vinden die groot genoeg is om de gegevens op te slaan en dan de boekhouding moet doen om de volgende allocatie voor te bereiden.

Rust heeft dus een voorkeur om zijn variabelen weg te schrijven naar de stack. Maar zoals gezegd kunt u niet elke variabele wegschrijven naar de stack, daarom is er een onderscheid tussen simpele en complexe types. Bij simpele types kent u de grootte voor het compileren, daarmee worden ze dan ook opgeslagen op de stack. In tegenstelling tot complexe types waarbij de grootte kan veranderen, worden ze opgeslagen in de heap.

De simpele types zijn dus: integer, floating-point, boolean, tuple, array.

Een paar voorbeelden van complexe types: string, vector, struct.

Nu de begrippen zijn opgeklaard kunnen we kijken naar het ownership model. Het model bestaat uit drie regels:

1. Elke waarde in Rust heeft een variabele die de owner wordt genoemd
2. Er kan maar een owner per keer zijn
3. Wanneer de owner buiten scope gaat, zal de waarde worden verwijderd

Deze regels worden gecontroleerd bij het compileren met behulp van de borrow checker. Als een van de regels wordt overtreden zal het programma niet compileren.

Laat ons eens kijken naar een simpel voorbeeld.

```
{
  let s = String::from("hello"); // s is geldig vanaf deze lijn
  // doe iets met s
} // hier eindigt de scope, en s is niet langer geldig
```

Listing 5: ownership

In het voorbeeld gebruiken we het complexe `String` type. Als we de regels volgen, wordt de variabele s owner over de string `hello`. De variabele `s` blijft geldig zolang hij binnen de scope wordt aangeroepen. Op het einde van de scope zal Rust de `drop` functie uitvoeren en dus het geheugen terug vrijgeven. Dit is de basis van hoe het ownership model werkt in Rust. In realiteit komen er natuurlijk nog wat eigenaardigheden bij kijken, voor meer info hierover verwijs ik u graag door naar het „The Rust Programming Language" boek. [4]

2.1.3 Ecosysteem

Het Rust team & de community hebben veel aandacht besteed aan de gebruiksvriendelijkheid. Zo is de installatie van de taal zeer eenvoudig met de `rustup` toolchain installer. De installer laat ook toe om makkelijk te wisselen tussen verschillende versies zoals stable, beta en nightly compilers en houdt ze up to date.

Rust-installaties worden geleverd met Cargo, als package manager. Cargo downloadt de dependencies van uw Rust package, compileert uw packages, maakt distribueerbare packages, en upload ze naar `crates.io`, de package registry van de Rust community. Buiten het beheren van packages heeft het nog een aantal features. Met Cargo kan u tests uitvoeren, documentatie genereren, automatisch warnings oplossen en nog veel meer.

Naast de ingebouwde tools, heeft de Rust community een groot aantal development tools gemaakt. Benchmarking, fuzzing, en property-based testing zijn allemaal gemakkelijk toegankelijk en worden goed gebruikt in projecten. Extra compiler lints zijn beschikbaar via Clippy en automatische idiomatische formattering wordt geleverd door rustfmt. De IDE-ondersteuning is gezond en wordt elke dag beter.

2.2 Wat is WebAssembly & Hoe werkt het?

Tegenwoordig wordt Javascript gebruikt om interactieve webapps te creëren. Ondanks de succesvolle inspanningen van browsermakers om hun Javascript-engines in elke versie weer wat efficiënter te maken, is dat voor veel toepassingen nog steeds niet genoeg. In 2011 lanceerde Google Native Client (NaCl), een sandbox voor het efficiënt en veilig uitvoeren van gecompileerde C en C++ code in de browser. Het bracht prestaties en low-level controle van native code naar moderne webbrowsers, zonder de veiligheid en portabiliteit van het web op te offeren. [7]

Vervolgens kwam Mozilla in 2013 met een andere aanpak: asm.js, een subset van Javascript die browsers heel efficiënt kunnen uitvoeren. Hiermee kan een webapp uit een taal zoals C gecompileerd worden naar asm.js en de browser zal dit uitvoeren als gewone Javascript.

Het voordeel van asm.js is dat het simpelweg in alle webbrowsers werkte, maar Mozilla botste tegen de snelheidsgrenzen van Javascript aan. Sinds Javascript een tekstformaat is, vereist het parsen veel rekenkracht, vooral op mobiele apparaten met een iets zwakkere processor. Daarom werd WebAssembly (afgekort Wasm) in 2015 geboren, een binair instructieformaat voor een stack-gebaseerde virtuele machine in uw browser. Het is ontworpen als een overdraagbaar compilatiedoel voor programmeertalen, waardoor het gebruik op het web mogelijk wordt voor client- en servertoepassingen. [8]

2.2.1 Doel

Het is dus geen nieuwe programmeertaal, maar een binair formaat voor uitvoerbare programma's. Het wordt gecreëerd als een open standaard binnen de World Wide Web Consortium (W3C) met de volgende doelstellingen:

- **Snel, efficiënt en overdraagbaar** - wasm code kan op bijna native snelheid worden uitgevoerd op verschillende platforms door gebruik te maken van gemeenschappelijke hardware mogelijkheden.
- **Leesbaar en foutopspoorbaar** - wasm is een lage assembleertaal, maar het heeft een menselijk leesbaar tekstformaat (aan de specificatie wordt nog gewerkt) waarmee code met de hand kan worden geschreven, bekeken en foutopsporing mogelijk is.
- **Veilig** - wasm is gespecificeerd om te worden uitgevoerd in een veilige, sandboxed omgeving. Net als andere webcode, zal het de browser's same-origin en permissies beleid afdwingen.
- **Maak het web niet kapot** - wasm is zo ontworpen dat het goed samengaat met andere webtechnologieën en achterwaartse compatibiliteit behoudt.

WebAssembly is niet beperkt tot het web. Maar tot nu toe heeft het grootste deel van de ontwikkeling van WebAssembly zich gericht op het web. Dat komt omdat men betere ontwerpen kan maken als men zich richt op het oplossen van concrete use cases. De taal zou zeker op het Web moeten draaien, dus dat was een goede use case om mee te beginnen. [9]

Buiten het web kan wasm worden uitgevoerd door `wasmtime` [10]. Een project door *Bytecode Alliance* om wasm te draaien als een command-line utility of een library in een ander project. Theoretisch, gebaseerd op de aard van wasm heeft het geen toegang tot de 'host' en de API van het systeem, dit is waar WASI in het verhaal komt. WASI staat voor *WebAssembly System Interface* en is een modulair systeeminterface voor wasm dat het gemakkelijker maakt om de 'host' met de runtime te verbinden.[11]

2.2.2 Hoe werkt het?

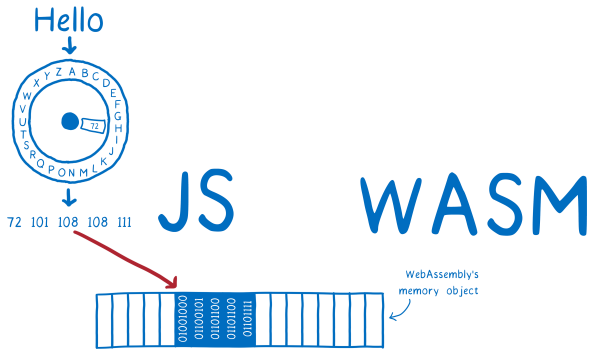
Nu we weten wat wasm op een hoog niveau inhoudt, is het ook goed om eens praktisch te kijken hoe het precies werkt. Er zijn namelijk een aantal opties voor het compileren naar wasm:

- C/C++ applicatie omzetten naar wasm met Emscripten
- wasm rechtstreeks op assembly niveau schrijven of genereren
- een Rust applicatie schrijven en wasm als compilatie target gebruiken
- AssemblyScript gebruiken, wat vergelijkbaar is met Typescript en compileert naar een wasm binary

In deze bachelorproef werd het technisch onderzoek uitgevoerd in Rust. Met zijn kleine runtime, betrouwbaar en rijk typesysteem is het een van de populairste keuzes voor het bouwen van webapps met wasm. Helaas is wasm nog niet compleet en hebben we nog altijd Javascript nodig om te praten met de DOM. Dus laat ons even kijken naar een voorbeeld hoe we vanuit Rust Javascript functies kunnen gebruiken en andersom.

Een van de moeilijkste onderdelen van het werken met wasm is om verschillende soorten waarden in en uit functies te krijgen. Dat komt omdat wasm momenteel slechts twee types kent: **integers** en **floating-point** getallen.

Dit betekent dat u niet zomaar een string in een wasm functie kunt stoppen. In plaats daarvan moet u een aantal stappen doorlopen om een string voor te stellen als getallen. Als u complexere types heeft, zal u zelfs een ingewikkelder proces hebben om de gegevens heen en weer te sturen. Gelukkig bestaat er de library `wasm-bindgen` die deze stappen voor ons doet. Met een paar annotaties aan uw Rust code, zal het automatisch de code maken die nodig is (aan beide kanten) om complexere types te laten werken.



Figuur 2.1: JS string naar wasm conversie

Dit betekent dat JS functies vanuit Rust worden aangeroepen met de types die deze functies verwachten:

```rust
#[wasm_bindgen]
extern {
  type console;
  #[wasm_bindgen(static = console)]
  fn log(s: &str);
}
```

```rust
#[wasm_bindgen]
pub fn foo() {
  console::log("hello!");
}
```

Of structs gebruiken in Rust en ze laten werken als klassen in JS:

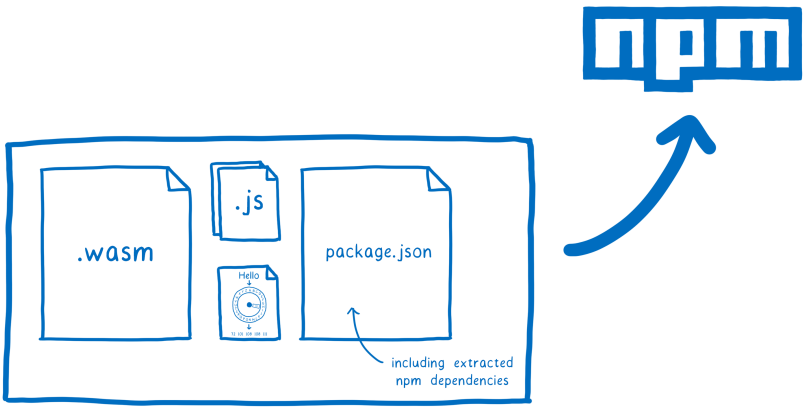
```rust
// rust
#[wasm_bindgen]
pub struct Foo {
  contents: u32,
}

#[wasm_bindgen]
impl Foo {
  pub fn new() -> Foo {
    Foo { contents: 0 }
}
```

```javascript
// JS
import {Foo} from "./js_hello_world";

let foo = Foo.new();
assertEq(foo.add(10), 10);
foo.free();
```

Na het compileren, zijn er een heleboel bestanden. Om van die bestanden naar een npm package te gaan, is er de `wasm-pack` library. Het zal `wasm-bindgen` voor u draaien. Dan, zal het alle bestanden nemen en ze verpakken. Het zal een package.json er aan toevoegen, waarin alle npm dependencies van uw Rust code zijn ingevuld. Ten slotte, alles wat u hoeft te doen is `npm publish`.



Figuur 2.2: publiceren van wasm als npm package

2.3 Welke front- & backend frameworks zijn er ter beschikking?

Om efficiënt webapplicaties te bouwen heeft u een degelijk framework nodig die voor jou al het zware werk doet. Gelukkig heeft Rust mits zijn jonge jaren, al een aardig aantal frameworks ter beschikking voor het bouwen van webapplicaties.

Dit zijn de top 3 front- en backend frameworks. Gerangschikt naar mate van hun populariteit (GitHub stars).

2.3.1 Frontend

Yew - 21k

Yew is het populairste frontend framework met meer dan 21k GitHub stars. Het beschikt over een component-gebaseerd framework dat het makkelijk maakt om interactieve UI's te maken. Ontwikkelaars die ervaring hebben met frameworks als React en Elm zouden zich helemaal thuis moeten voelen bij het gebruik van Yew. Naast de aangename developer experience brengt het ook geweldige prestaties met zich mee. Dit bereiken ze door het minimaliseren van het aanroepen naar de DOM en door ontwikkelaars te helpen om gemakkelijk taken te ontladen naar achtergrond threads met behulp van web workers.

De documentatie over de huidige release versie 19.0 is uitgebreid geschreven. Er kan zelfs al gekeken worden naar de volgende release 'Next', die refereert naar de master branch. In de 'Next' versie is belangrijkste nieuwe feature SSR, want de huidige versie heeft alleen maar CSR.

Dixous - 3.8k

Dixous is een opkomend en jong UI framework als concurrent voor Yew. Het is net als Yew ontworpen om React-achtig te zijn. Zo heeft het ook een component-gebaseerde architectuur, state management, props en nog veel meer. Buiten de verschillende syntax tegen over Yew, heeft Dixous nog een aantal troeven zoals:

- de keuze tussen JSX-achtige of hun eigen macro gebaseerde RSX syntax als templating systeem ingebouwde globale state en error handler
- components en hooks kunnen worden hergebruikt voor te renderen op het web, desktop, mobiel, server en meer
- uitgebreide inline documentatie
- SSR

Seed - 3.3k

Seed is een frontend framework voor het maken van prestatiegerichte en betrouwbare webapps die een Elm-achtige architectuur heeft. Het heeft een minimale configuratie en boilerplate, en heeft duidelijke documentatie die het voor iedereen gemakkelijk maakt om mee te beginnen.

Ook Seed gebruikt een eigen templating systeem met een macro syntax waardoor Rustaceans zich meteen thuis voelen. Dit betekent dat linting, formatteren en commentaar geven zullen werken, en het is allemaal in Rust. Dit in tegenstelling tot een JSX-achtige syntax (zoals die van Yew) die afhankelijk is van IDE-extensies om de developer experience te verbeteren.

Seed beschikt niet over SSR en heeft nog geen plannen om het te implementeren.

2.3.2 Backend

Rocket – 17.2k

Rocket is een populair webframework dat het voor developers gemakkelijk maakt om snelle webapps te schrijven zonder te bezuinigen op veiligheid, flexibiliteit of functionaliteiten. Het heeft ondersteuning voor het testen van libraries, cookies, streams, routes, templates, databases, ORMs, boilerplates, en nog veel meer. Rocket heeft ook een grote en actieve community.

Actix-web – 14.1k

Net als Rocket, is Actix een ander krachtig backend web framework. Actix heeft een architectuurpatroon gebaseerd op het actor-systeem van Rust en is goed uitgerust voor het bouwen van schrijfdiensten en micro apps. Het heeft ondersteuning voor routing, middleware, testen, WebSockets, automatisch server reloading en kan gehost worden op NGINX. Actix kan worden gebruikt om een volledige web app en API te bouwen.

Axum – 4.7k

Alhoewel er nog populairdere frameworks zijn dan Axum verdient het zeker een plaats in de top 3. Axum is deel van het populaire Tokio project, met sponsors als aws, azure en facebook. Tokio is een asynchrone runtime voor Rust, die de bouwstenen biedt die nodig zijn voor het schrijven van netwerktoepassingen.

Axum is een laag bovenop Tokio's HTTP client genaamd hyper, die een relatief low-level library is en bedoeld is al bouwsteen voor libraries en applicaties. Het framework focust op ergonomie en modulariteit. Wat het nog onderscheidt met andere frameworks, is dat het geen eigen middleware systeem heeft maar gebruikt in plaats daarvan de tower::Service module van het Tokio project. Dit betekent dat Axum gratis timeouts, tracing, compressie, authorisatie en meer krijgt.

Dit alles maakt met zijn jonge 1-jarige leeftijd toch een library om naar uit te kijken.

2.4 Hoe bouwt u een webapp in Rust?

Sinds Yew het populairste framework is en ook is gebruikt als framework voor het bouwen van de speed typing applicatie, zal de vraag „Hoe bouwt u een webapp in Rust" beantwoord worden met Yew als framework. Het bouwen van een webapplicatie in een ander framework zal in grote lijnen hetzelfde zijn. Dit zal een praktische kijk zijn op hoe we Yew kunnen gebruiken voor het bouwen van Webapplicaties. [12]

In dit voorbeeld gaan we een simpele Todo applicatie bouwen.

2.4.1 Tools installeren

Rust

Om Rust te installeren hebben we de rustup toolchain installer nodig. Met het onderstaand script kan je het installeren op jouw UNIX machine. Als u Rust al hebt staan maak dan zeker dat u de laatste versie heeft door `rustup update` uit te voeren.

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

WebAssembly

Rust kan source codes compileren voor verschillende 'targets' (m.a.w verschillende processors). Het compilatie target voor een browser gebaseerd WebAssembly heet wasm32-unkown-unkown. Het volgende commando zal het WebAssembly target toevoegen aan uw development environment.

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

Trunk

De documentatie van Yew raad aan om Trunk te gebruiken voor het beheren van deployment en packaging.

```
cargo install --locked trunk
```

Nu ons development enviroment opgezet is, kunnen we een nieuw cargo project aanmaken.

```
cargo new yew-app
cd yew-app
```

Om te verifieren dat het Rust environment juist is opgezet, voert u het initiele project uit met de cargo build tool. Na de output van de het build process, zou u normaal `"Hello, world!"` te zien krijgen.

```
cargo run
```

Statische pagina

Om deze simpele command line applicatie naar een basis Yew web applicatie te convertern, zijn er een paar aanpassingen nodig. Pas de volgende bestanden aan als volgt:

```
[package]
name = "yew-app"
version = "0.1.0"
edition = "2021"

[dependencies]
yew = "0.19"
```

Listing 6: Cargo.toml

```
use yew::prelude::*;

#[function_component(App)]
fn app() -> Html {
    html! {
        <h1>{ "Hello World" }</h1>
    }
}

fn main() {
    yew::start_app::<App>();
}
```

Listing 7: main.rs

Maak nu een index.html aan in de root folder van het project.

```
<!DOCTYPE html>
<html lang="en">
  <head> </head>
  <body></body>
</html>
```

Listing 8: index.html

Start de development server

Voer het volgende commando uit om de applicatie te builden en lokaal te draaien.

```
trunk serve --open
```

2.4.2 Statische pagina

Bouwen van HTML

Yew maakt gebruik van de procedurele macro's van Rust en biedt ons een syntax die lijkt op JSX (een uitbreiding van JavaScript waarmee u HTML-achtige code kunt schrijven in JavaScript) om de opmaak te maken.

Converteren van HTML naar Rust

Aangezien we al een vrij goed idee hebben van hoe onze website eruit zal zien, kunnen we onze mentale opzet eenvoudig vertalen naar een voorstelling die compatibel is met html!. Als u eenvoudige HTML kunt schrijven, moet het geen probleem zijn om markeringen in html! te schrijven. Het is belangrijk op te merken dat de macro op een paar punten verschilt van HTML:

- Uitdrukkingen moeten tussen accolades staan ({ })
- Er mag maar één root node zijn. Als u meerdere elementen wilt hebben zonder ze in een container te wikkelen, wordt een lege tag/fragment (<> ... </>) gebruikt
- Elementen moeten goed worden afgesloten.

We willen een layout bouwen die er ongeveer zo uitziet in ruwe HTML:

```
<main>
  <h1>My todo list</h1>
  <ul>
    <li>
      <input type="checkbox" />
      <label>Take dog out for a walk</label>
    </li>
    <input type="checkbox" />
    <label>Feed the cats</label>
    <li>
      <input type="checkbox" />
      <label>Take out the trash</label>
    </li>
    <li>
      <input type="checkbox" />
      <label>Water plants</label>
    </li>
  </ul>
</main>
```

Laten we nu deze HTML in html! omzetten. Type (of kopieer/plak) het volgende knipsel in de body van de app functie, zodat de waarde van html! wordt geretourneerd door de functie.

```
#[function_component]
pub fn App() -> Html {
  html! {
    <main>
      <h1>{ "My todo list" }</h1>
      <ul>
        <li>
          <input type="checkbox"/>
          <label> { "Take dog out for a walk" } </label>
        </li>
          <input type="checkbox"/>
          <label> { "Feed the cats" } </label>
        <li>
          <input type="checkbox"/>
          <label> { "Take out the trash" } </label>
        </li>
        <li>
          <input type="checkbox"/>
          <label> { "Water plants" } </label>
        </li>
      </ul>
    </main>
  }
}
```

Listing 9: app.rs

Components

Components zijn de bouwstenen van Yew applicaties. Door components te combineren, die weer uit andere components kunnen worden opgebouwd, bouwen we onze applicatie. Door onze components te structureren voor herbruikbaarheid en ze generiek te houden, kunnen we ze in meerdere delen van onze applicatie gebruiken zonder code of logica te hoeven dupliceren.

In feite is de app functie die we tot nu toe hebben gebruikt een component, genaamd `App`. Het is een 'function component'. Er zijn twee verschillende soorten componenten in Yew:

- Struct Components
- Function Components

In dit voorbeeld zullen we function components gebruiken.

Laten we nu onze `App` component opsplitsen in kleinere componenten. We kunnen onze Todo lijst opspliten in 2 components genaamd `Task` en `TaskList`.

```
#[derive(Properties, Debug, PartialEq)]
pub struct TaskProps {
  pub id: String,
  pub title: String,
  pub completed: bool,
}

#[function_component]
pub fn Task(
  TaskProps {
    id,
    title,
    completed,
  }: &TaskProps,
) -> Html {
  html! {
    <li>
      <input
        type="checkbox"
        id={id.clone()}
        checked={*completed}
      />
      <label
        for={id.clone()}>{title.clone()}
      </label>
    </li>
  }
}
```

Listing 10: task.rs

Let op de parameters van onze `Task` function component. Een function component heeft slechts één argument dat zijn 'props' (kort voor 'properties') definieert. Props worden gebruikt om gegevens door te geven van een ouder component naar een kind component. In dit geval is `TaskProps` een `struct` die de props definieert.

```
#[derive(Properties, PartialEq)]
pub struct TaskListProps {
  pub children: Children,
}
#[function_component]
pub fn TaskList(TaskListProps { children }: &TaskListProps) -> Html {
  html! {
    <ul>
      { for children.iter() }
    </ul>
  }
}
```

Listing 11: task_list.rs

Nu kunnen we onze `App` component updaten met onze nieuwe components `Task` & `TaskList`.

```
#[function_component]
pub fn App() -> Html {
  let tasks = vec![
    html! {
      <Task id={"1"} title={"Take dog out for a walk"} completed={true} />
    },
    // ... andere tasks
  ];
  html! {
    <main>
      <h1>{ "My todo list" }</h1>
      <TaskList>
        {tasks}
      </TaskList>
    </main>
  }
}
```

Listing 12: app.rs

Interactief

Momenteel doet onze applicatie niet veel anders dan onze voor gedefineerde lijst te tonen. Uiteindlijk willen we onze taken uit de todo lijst kunnen schrappen. Om deze interactie te laten werken zullen we een aantal zaken aanpassen.

Om bij te houden of de gebruiker de taak geschrapt heeft al dan niet, zullen we een `completed_state` gebruiken met de `use_state` hook. Zo verliezen we niet de waarde van de variabele `completed_state` mocht het component re-renderen.

```
#[function_component]
pub fn Task(
    ...
) -> Html {
    let completed_state = use_state(|| *completed);
    ...
}
```

Het volgende is natuurlijk het `onclick` event afhandelen als de gebruiker op het label of input element klikt. Hiervoor hebben we een `Callback` functie nodig die onze `completed_state` aanpast naargelang de vorige state. Voor we de `completed_state` kunnen gebruiken in onze Callback closure functie moeten we die eerst clonen. De reden hiervoor is het keyword `move` die voor de closure parameters staat, `move` zorgt ervoor dat alle references die gebruikt worden in de closure scope hun waarden worden verplaatst binnen de scope. Dus om te voorkomen dat we onze `completed_state` nergens meer kunenn gebruiken clonen we eerst de state.

Daarnaast kunnen we ook een css class meegeven met de `classes!` macro van yew, die conditioneel het html label zal doorstrepen al dan niet.

```rust
#[function_component]
pub fn Task(
    TaskProps {
        id,
        title,
        completed,
    }: &TaskProps,
) -> Html {
    let completed_state = use_state(|| *completed);

    let onclick = {
        let completed_state = completed_state.clone();

        Callback::from(move |_| {
            if *completed_state {
                completed_state.set(false);
            } else {
                completed_state.set(true);
            }
        })
    };

    let completed_class = {
        if *completed_state {
            "line-through"
        } else {
            ""
        }
    };

    html! {
        <li>
            <input
                {onclick}
                type="checkbox"
                id={id.clone()}
                checked={*completed_state}
            />
            <label
                class={classes!(completed_class)}
                for={id.clone()}>{title.clone()}
            </label>
        </li>
    }
}
```

Listing 13: task.rs

Data extern ophalen

In de speed typing applicatie komen de code snippets van een API in plaats van hard gecodeerd te zijn in de frontend. Laten we onze todo lijst ophalen van een externe bron. Hiervoor moeten we de volgende crates toevoegen:

- reqwasm voor het maken van de fetch call.
- serde met derive functies Voor het de-serialiseren van het JSON antwoord
- wasm-bindgen-futures voor het uitvoeren van Rust Future als een Promise

Laten we de dependencies in het Cargo.toml bestand bijwerken:

```
[dependencies]
yew = { git = "https://github.com/yewstack/yew/", features = ["csr"] }
serde = { version = "1.0", features = ["derive"] }
wasm-bindgen-futures = "0.4"
```

Pas de `TaskProps` struct aan om de `Deserialize` trait af te leiden.

```
#[derive(Properties, Debug, PartialEq, Deserialize)]
pub struct TaskProps {
  pub id: String,
  pub title: String,
  pub completed: bool,
}
```

Als laatste stap moeten we onze App component updaten om de fetch request te maken in plaats van hardcoded data te gebruiken.

Hier gebruiken we de `use_effect_with_deps` hook met lege dependencies als tweede parameter om de tasks slechts eenmaal op te halen bij de eerste render van het App component. In de closure gebruiken we `wasm-bindgen-futures` om de Javascript Promise die de `Request::get` genereert om te zetten naar een Future type. Met de `Request::get` halen we de JSON todos op en slaan ze op als een `vec` met `TaskProps`. Vervolgens vullen we onze `tasks` state met de `fetched_tasks` en zullen de tasks worden weergeven in de browser!

```
#[function_component]
pub fn App() -> Html {
let tasks = use_state(std::vec::Vec::new);
  {
    let tasks = tasks.clone();
    use_effect_with_deps(move |_| {
      wasm_bindgen_futures::spawn_local(async move {
        let fetched_tasks: Vec<TaskProps> = Request::get("url")
          .send()
          .await
          .unwrap()
          .json()
          .await
          .unwrap();

        tasks.set(fetched_tasks.iter().take(10).map(|props| {
            html! {
              <Task
                id={props.id.clone()}
                title={props.title.clone()}
                completed={props.completed}
              />
            }
          }).collect()
        );
      });
      || ()
    }, ());
  }

  html! {
      <main>
          <h1>{ "My todo list" }</h1>
          <TaskList>
              { (*tasks).clone() }
          </TaskList>
      </main>
  }
}
```

Listing 14: app.rs

2.5 Hoe bouwt u een API in Rust?

Volgend op de Todo applicatie die we gebouwd hebben in „Hoe bouwt u een webapp in Rust?", zullen we een REST API bouwen die de taken uit een database zal halen en opslaan. Als API framework zullen we Actix-web gebruiken, samen met diesel.rs als ORM voor het beheren van de database. Om het simpel te houden gebruiken we net zoals in de speed typing test applicatie SQLite als database. [13] [14]

2.5.1 Opzet project

Maak een nieuw Rust project aan met de volgende dependencies.

```
cargo new api
cd api/
```

```
[dependencies]
diesel = { version = "1.4.8", features = ["sqlite", "r2d2"] }
dotenv = "0.15.0"
actix-web = "4"
actix-cors = "0.6.1"
uuid = { version = "0.8", features = ["serde", "v4"] }
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
log = "0.4"
env_logger = "0.9.0"
```

Listing 15: Cargo.toml

We zullen uitleggen waarom we deze dependencies nodig hebben als we verder gaan.

Database

Diesel biedt een aparte CLI tool om uw project te helpen beheren. Omdat het een standalone binary is, en geen directe invloed heeft op de code van uw project, voegen we het niet toe aan Cargo.toml. In plaats daarvan installeren we het gewoon op ons systeem.

```
echo DATABASE_URL=todo.db > .env
```

Nu kan Diesel CLI alles voor ons opzetten.

```
diesel setup
```

Dit zal onze database aanmaken (als die nog niet bestond), en een lege migrations directory aanmaken die we kunnen gebruiken om ons schema te beheren.

```
CREATE TABLE tasks (
  id VARCHAR NOT NULL PRIMARY KEY,
  title VARCHAR NOT NULL,
  completed INTEGER NOT NULL DEFAULT 0
)
```

Listing 16: up.sql

```
DROP TABLE tasks
```

Listing 17: down.sql

Diesel support momenteel alleen een database-first aanpak. Hierbij maken we eerst het database schema om dan met migrations het schema naar Rust om te zetten. Dat gezegd zijnde, laat ons een eerste migration aanmaken.

```
diesel migration generate create_tasks
```

Diesel CLI zal twee lege bestanden (`up.sql` en `down.sql`) voor ons aanmaken in de vereiste structuur. In deze bestanden schrijven we SQL voor de Task tabel aan te maken in `up.sql` en als we de migration willen ongedaan maken in `down.sql`.

Nu kunnen we onze eerste migration uitvoeren met:

```
diesel migration run
```

Dit zal onze SQL migration uitvoeren op de `todo.db` database en het nodige Rust schema genereren met de nodige types. In het Rust schema wordt met de `table!` macro een hoop code gegeneerd gebaseerd op het database schema om alle tabellen en kolommen weer te geven. Zo kunnen we nu gebruik maken van Rust zijn krachtig typesysteem om volledige SQL queries te gaan bouwen met code. We zullen in het volgende voorbeeld zien hoe we dat precies kunnen gebruiken.

Telkens wanneer we een migratie uitvoeren of terugdraaien, wordt dit bestand automatisch bijgewerkt.

2.5.2 API

Om de basis functionaliteiten van onze Todo applicatie in de front-end te ondersteunen, zal onze API een nieuwe taak kunnen aanmaken en de lijst met taken ophalen. Dus zullen we de volgende endpoints hebben:

- GET /tasks - retourneert alle taken
- POST /tasks - voegt een nieuwe taak toe

Om onze code overzichtelijklijk te houden, zullen we de volgende structuur hanteren:

```
src/
├── main.rs
├── models.rs
├── schema.rs
├── handlers.rs
└── db.rs
```

Bij het opzetten van ons project heeft cargo een al een `main.rs` bestand aangemaakt. Laten we dat bewerken en onze eerste `"Hello World!"` route schrijven.

```rust
use actix_web::{web, App, HttpServer};

#[actix_web::main]
async fn main() -> std::io::Result<()> {
  HttpServer::new(|| {
    App::new()
      .route("/hello", web::get().to(|| async { "Hello World!" }))
  })
  .bind(("127.0.0.1", 5000))?
  .run()
  .await
}
```

Listing 18: main.rs

Onze main functie heeft nu het `#[actix_web::main]` attribuut, die zal onze main functie markeren als start functie en uitvoeren in de runtime van `actix_web`.

Een belangrijk punt om op te merken is dat we een `Result` type teruggeven. Dit stelt ons in staat om de ? operator in main te gebruiken, die elke fout die door de geassocieerde functie wordt teruggegeven naar de aanroeper koppelt.

Het tweede ding om op te merken is `async/await`. Hiermee geven we aan dat Rust onze functies asynchroon kan uitvoeren zonder andere threads te blokkeren.

In onze main, instantiëren we een `HttpServer`, voegen er een `App` aan toe die dient als een Application factory en voegen onze eerste route er aan toe.

Als alles goed gaat zou u nu de API kunnen opstarten met `cargo run` en bij een GET request naar `localhost:8080/hello` `"Hello world!"` als repons te zien krijgen.

```
#[get("/task")]
async fn get_tasks() -> Result<HttpResponse, Error> {
  todo!()
}

#[post("/task")]
async fn add_task() -> Result<HttpResponse, Error> {
  todo!()
}
```

Listing 19: handlers.rs

In de module `handlers` zullen we onze requests per endpoint afhandelen. Voorlopig gebruiken we hier nog de `todo!` macro sinds we nog geen communicatie maken de database. Laat ons dat eerst aanpakken.

Onze eerste migration die we eerder hebben uitgevoerd heeft voor ons al een `schema.rs` module aangemaakt, zodat diesel onze queries kan controleren tijdens het compileren. Om met queries te kunnen werken in Rust hebben we models nodig. Zo kunnen we het resultaat van een query mappen naar een model of een model gebruiken voor nieuwe data in te voegen.

Met de traits `Queryable` en `Insertable` zorgen we er voor dat de struct `Task` als resultaat voor een query kan gebruikt worden en als struct om nieuwe data in te voegen.

`Deserialize` en `Serialize` komen van de crate `serde`, zij zorgen ervoor dat we de models kunnen omzetten naar json en andersom.

```rust
#[derive(Debug, Serialize, Deserialize, Queryable, Insertable)]
pub struct Task {
  pub id: String,
  pub title: String,
  pub completed: bool,
}

#[derive(Debug, Serialize, Deserialize)]
pub struct InputTask {
  pub title: String,
}
```

Listing 20: models.rs

Nu we onze models hebben kunnen we in `db` functies schrijven om tasks op te vragen en aan te maken. De functies spreken voorzich, we gebruiken diesel zijn sql implementaties om met onze database te communiceren.

Merk op dat we bij elke functie `schema::tasks::dsl::*` opnieuw toevoegen aan de lokale scope. Dit is puur conventie sinds we maar 1 tabel hebben `Tasks`, moesten we meerdere tabellen hebben zouden we onze scope kunnen vervuilen door bijvoorbeeld zelfde kolomnamen die elkaar overschrijven.

```rust
type DbError = Box<dyn std::error::Error + Send + Sync>;

pub fn list_all_tasks(conn: &SqliteConnection)
  -> Result<Option<Vec<Task>>, DbError> {
  use crate::schema::tasks::dsl::*;
  Ok(tasks.load::<Task>(conn).optional()?)
}
pub fn insert_new_task(t: &str, conn: &SqliteConnection)
  -> Result<Task, DbError> {
  use crate::schema::tasks::dsl::*;
  let new_task = Task {
    id: Uuid::new_v4().to_string(),
    title: t.to_owned(),
    completed: false,
  };
  diesel::insert_into(tasks).values(&new_task).execute(conn)?;
  Ok(new_task)
}
```

Listing 21: db.rs

Nu we onze helper functies geschreven hebben kunnen we de requests afhandelen in `handlers` als volgt:

Elke handler functie krijgt een connection pool als parameter, die we later zullen definieren in main. De connection pool zal verschillende connecties met de database openhouden zodat ze efficient kunnen hergebruikt worden door anderen. Om te voorkomen dat een andere thread de connectie gebruikt tijdens het uitvoeren van een `db` functie blokkeren we deze thread met `web::block`. Als alles goed verloopt geven we de resultaten terug, indien niet geven we een `InternalServerError` terug.

```
#[get("/task")]
async fn get_tasks(
  pool: web::Data<DbPool>,
) -> Result<HttpResponse, Error> {
  let tasks = web::block(move || {
    let conn = pool.get()?;
    db::list_all_tasks(&conn)
  })
  .await?
  .map_err(actix_web::error::ErrorInternalServerError)?;

  if let Some(tasks) = tasks {
    Ok(HttpResponse::Ok().json(tasks))
  } else {
    let res = HttpResponse::NotFound().body("No tasks found!".to_string());
    Ok(res)
  }
}
#[post("/task")]
async fn add_task(
  pool: web::Data<DbPool>,
  form: web::Json<models::InputTask>,
) -> Result<HttpResponse, Error> {
  let task = web::block(move || {
    let conn = pool.get()?;
    db::insert_new_task(&form.title, &conn)
  })
  .await?
  .map_err(actix_web::error::ErrorInternalServerError)?;

  Ok(HttpResponse::Created().json(task))
}
```

Listing 22: handler.rs

Wat ons nu nog rest te doen, is onze connection pool instantiëren en de handler functies linken aan onze App. Vooraleer dat we dat doen laden we eerst ons .env bestand in het enviroment met de `dotenv` crate en loggen we info over de applicatie naar `stdout` met `env_logger`. Daarna kunnen we onze connection pool opzetten met de `DATABASE_URL` uit het enviroment. Ten slotte voegen we onze hanlder functies toe aan een nieuwe service met als scope `/api` zodat al onze routes worden ge prefixed met `/api`.

```
type DbPool = r2d2::Pool<ConnectionManager<SqliteConnection>>;
#[actix_web::main]
async fn main() -> std::io::Result<()> {
  dotenv::dotenv().ok();
  env_logger::init_from_env(
    env_logger::Env::new().default_filter_or("info")
  );
  // set up database connection pool
  let conn_spec = std::env::var("DATABASE_URL").expect("DATABASE_URL");
  let manager = ConnectionManager::<SqliteConnection>::new(conn_spec);
  let pool = r2d2::Pool::builder()
    .build(manager)
    .expect("Failed to create pool.");
  log::info!(
    "{}",
    format!("starting HTTP server at http://localhost:5000"
  ));
  // Start HTTP server
  HttpServer::new(move || {
    App::new()
      .app_data(web::Data::new(pool.clone()))
      .wrap(middleware::Logger::default())
      .service(
        web::scope("/api")
        .service(get_tasks)
        .service(add_task)
      )
    })
  .bind(("127.0.0.1", 5000))?
  .run()
  .await
}
```

Listing 23: main.rs

2.6 Is Rust productie klaar?

Wanneer een taal 'productie klaar' benoemd kan worden is eigenlijk een moeilijke vraag om te beantwoorden. Wat 'productie klaar' betekent hangt af van verschillende factoren voor een specifiek domein. Rust schittert bijvoorbeeld in productief veilige en performante applicaties te bouwen. Top bedrijven zoals Amazon, Facebook, Microsoft kunnen dit beamen sinds ze ondertussen al een aantal jaar Rust gebruiken [15]. Zo heeft Amazon in 2018 Firecracker gelanceerd [16]. Een open source virtualisatietechnologie die AWS Lambda en andere serverloze aanbiedingen aandrijft. Daarnaast gebruiken ze Rust ook voor services te leveren zoals Amazon Simple Storage Service (Amazon S3), Amazon Elastic Compute Cloud (Amazon EC2), Amazon CloudFront, en meer. Amazon heeft dan ook samen met Google, Microsoft, Huawei en Mozilla de Rust foundation opgericht in 2020, met een missie om het ontwikkelen van Rust te ondersteunen. We kunnen dus concluderen dat de taal op zich wel een stabiele omgeving is. Dit onderzoek gaat natuurlijk wel over hoe we Rust kunnen gebruiken voor het bouwen van webapplicaties. We richten onze focus dan ook op dit domein. We zullen de deelvraag "Is Rust productie klaar?" opsplitsen in twee delen namelijk frontend en backend.

2.6.1 Backend

Laten we beginnen met de backend. Bij het bouwen van hedendaagse webapplicaties heeft u over het algemeen 2 zaken nodig, een API en een database. Uiteraard is dit niet altijd het geval maar laten we onze focus hierop leggen.

Om een web API te bouwen gebruikt u gewoonlijk een web framework. Het huidige rust ecosysteem heeft een aantal stabiele web frameworks ter beschikking. Uit de deelvraag "Hoe bouwt u een API in Rust?" hebben we gekozen voor Actix-web te gebruiken als framework. Actix-web voorziet alles wat u kan verwachten van een web framework, van routing en middleware, tot templating en JSON/form afhandeling. Het is ook een van de snelste web frameworks ter beschikking. Zo staat actix momenteel op de 5de plaats bij een benchmark ranking van alle populaire web frameworks (ook andere programmeertalen dan Rust) [17].

Het actix framework ondersteunt verschillende soorten databases zoals mongodb, postgres, redis, sqlite, graphql, elasticsearch en meer. De ene database heeft meer ondersteuning dan de andere, maar naarmate de tijd vordert zal de ondersteuning alleen maar verbeteren.

2.6.2 Frontend

Bij de frontend komt er wat meer bij kijken. Er zijn namelijk twee opties waarvoor Rust gebruikt kan worden:

- Een gedeelte in Rust schrijven en compileren naar wasm om vervolgens te importeren als een node module in een bestaande Javascript applicatie.
- Een volledige frontend in Rust schrijven met een web framework als Yew en compileren naar wasm

De eerste optie is meteen ook de optie waarvoor wasm het meest zal gebruikt worden. Zo kan u bijvoorbeeld Javascript gebruiken voor de UI en wasm voor de logica van de applicatie. Om mogelijks meer performatie te winnen en correctere code te schrijven. Deze optie wordt al volop in productie gebruikt door meerdere bedrijven [18]:

- 1Password – gebruikt wasm om hun browser plugin te versnellen
- Figma – gebruikt wasm om hun originele C++ applicatie te exporteren naar de browser
- SketchUp – gebruikt wasm om een 3d modelling tool op het web te draaien, wat snelle en voorspelbare prestaties verreist

De andere optie, een volledige frontend bouwen met Rust, is een optie die momenteel meer gebruikt wordt voor hobby projecten dan voor productie. Een van de redenen hiervoor is dat het ecosysteem nog zeer jong is. Het is bijvoorbeeld niet te vergelijken met het ecosysteem van populaire JS frameworks, wat ook logisch is. Desondanks het ecosysteem wordt Yew aan een snel tempo ontwikkeld. Waar sommige libraries als React jaren over doen om features uit te brengen als SSR doet Yew in enkele maanden.

Het is ook merkwaardig dat Yew degelijk scoort op benchmarks. Natuurlijk presteert Yew beter dan Javascript frameworks in het werken met geheugen, maar wat vooral opvalt is dat het ook redelijk presteert in het aanpassen van de DOM. Dit is nu nog een struikel punt voor WebAssembly in het algemeen, sinds er nog altijd javascript functies worden gebruikt voor het aanpassen van de DOM. Dat is een van de redenen waarom we hogere bundel grootes over het netwerk zien bij Yew. Een andere reden is dat WebAssembly zelf niet beschikt over geheugen beheer en garbage collectors. Om die reden wordt vaak een runtime meegeleverd met de logica van uw applicatie. In sommige gevallen (Rust) is deze runtime vrij licht, en in andere (Blazor) is het zeer zwaar. We zullen waarschijnlijk zien dat het gewicht van deze runtimes in de loop van de tijd aanzienlijk zal afnemen door nieuwe WebAssembly mogelijkheden (garbage collection, module caching) en betere compilatietechnieken (ahead-of-time compilation).

De resultaten van de benchmarks zijn terug te vinden in Bijlage A. Let wel op de bundle grootte van Yew is nog niet geoptimaliseert, met optimalisaties is het mogelijk om de bundle grootte naar ∼100kB te krijgen.

2.6.3 Conclusie

Om Rust in productie te gebruiken kan dus zeker. De vraag is alleen of dit enige meerwaarden heeft voor het bedrijf. Het antwoord hierop is terug te vinden Hoofdstuk 4 reflectie waar de vraag wordt afgetoetst met mensen uit de praktijk.

Hoofdstuk 3

Technisch onderzoek

3.1 Voorbereiding

Het technisch onderzoek startte bij het lezen van 'The book' een boek over de taal Rust geschreven door Steve Klabnik, Carol Nichols en met contributies van de Rust community. Tijdens het lezen waren er kleine code voorbeelden die u kon mee volgen. Dit maakte het makkelijker om de geavanceerde hoofdstukken goed te begrijpen.

Na een paar hoofdstukken diep in het boek was het tijd om de geleerde zaken tot de test te brengen. De opdracht was om de miniversie van het gekende Linux commando grep te maken. Zo werd de geleerde kennis over structs, ownership, enums en pattern matching, error handling, lifetimes en tests praktisch toegepast.

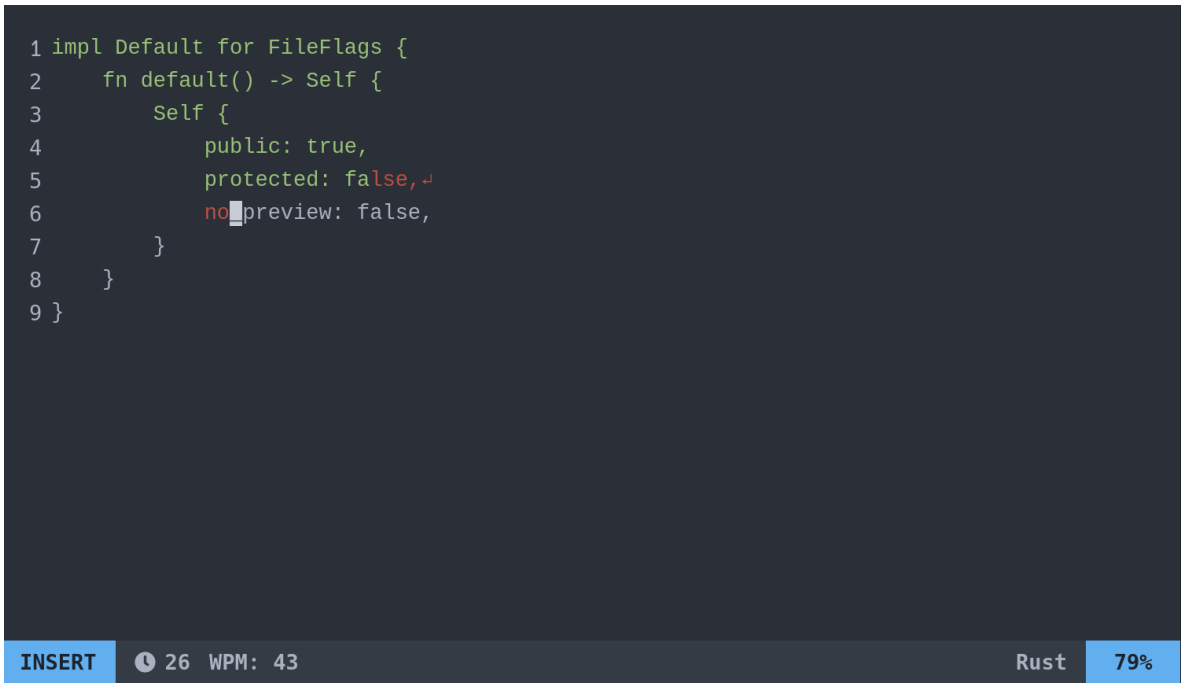
Op het einde van het boek was er een finaal project dat ook de laatste hoofdstukken bevatte. Het project was het bouwen van een 'simpele' multi-threaded server. Daar lag de focus op het werken met meerdere threads, wat niet voor de hand ligt als u rekening moet houden met het geheugen.

Na de taal onder de knie te hebben, werd er onderzoek gedaan naar WebAssembly. Mits het nog een piepjonge technologie is, was er al heel wat documentatie/artikelen over te vinden op het web. Zo begon het onderzoek bij de documentatie van Mozilla. Daar is alles te vinden om van start te gaan met WebAssembly. Er zijn ook doorverwijzingen naar hun uitstekende geschreven blogposts die dieper gaan op hoe WebAssembly precies werkt.

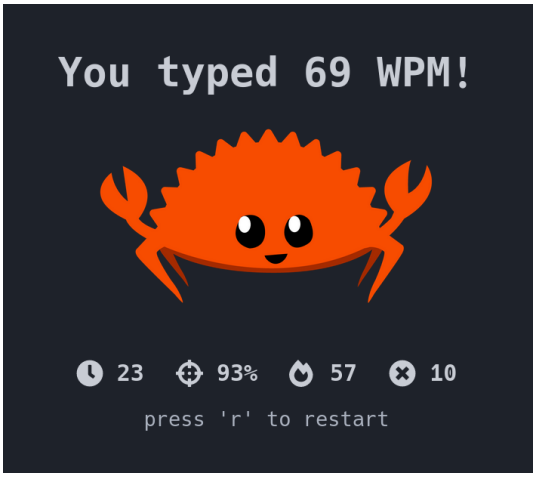
Vervolgens moest er een keuze gemaakt worden welke frameworks we zullen gebruiken voor het bouwen van de webapplicatie. Na wat onderzoek dat beschreven staat in „Welke front- end backend frameworks zijn er ter beschikking?", is er gekozen voor Yew als frontend en Actix web samen met diesel.rs als backend frameworks.

3.2 Omschrijving

Om de huidige status van Rust voor het bouwen van webapplicaties te onderzoeken, werd er gekozen voor een speed typing test applicatie te maken.

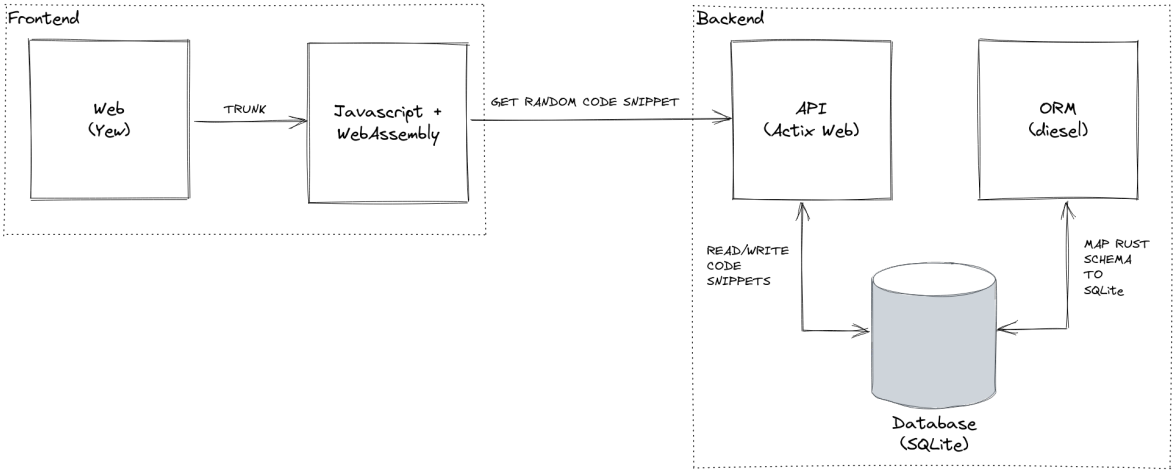


Het is een simpele applicatie waarbij de gebruiker een willekeurig code fragment krijgt die hij zo snel mogelijk moet overtypen. Tijdens het typen wordt er live, zijn tijd en woorden per minut (WPM) bijgehouden. Als de gebruiker het volledige fragment heeft overgetypt krijgt hij een resultaten pagina te zien. Daar kan hij zijn statistieken bekijken zoals: WPM, verlopen tijd, accuraatheid en het aantal fouten. Daarna kan de gebruiker opnieuw spelen door op de letter 'r' in te drukken.



Er was voor ogen om bij dit project nog een aantal functionaliteiten eraan toe te voegen. Zoals authenticatie met SSO, de statistieken opslaan in de database zodat er een lijn grafiek kon getoond op basis van de historiek, andere database, profiel pagina, ci/cd enz. Helaas is het onderzoek niet zo ver gekomen. Het leren van Rust en WebAssembly nam meer tijd in beslag dan verwacht. Zonder voorkennis van Rust of een gelijkaardige systeem level programmeertaal was dit project al een hele uitdaging op zich.

3.3 Opbouw/structuur



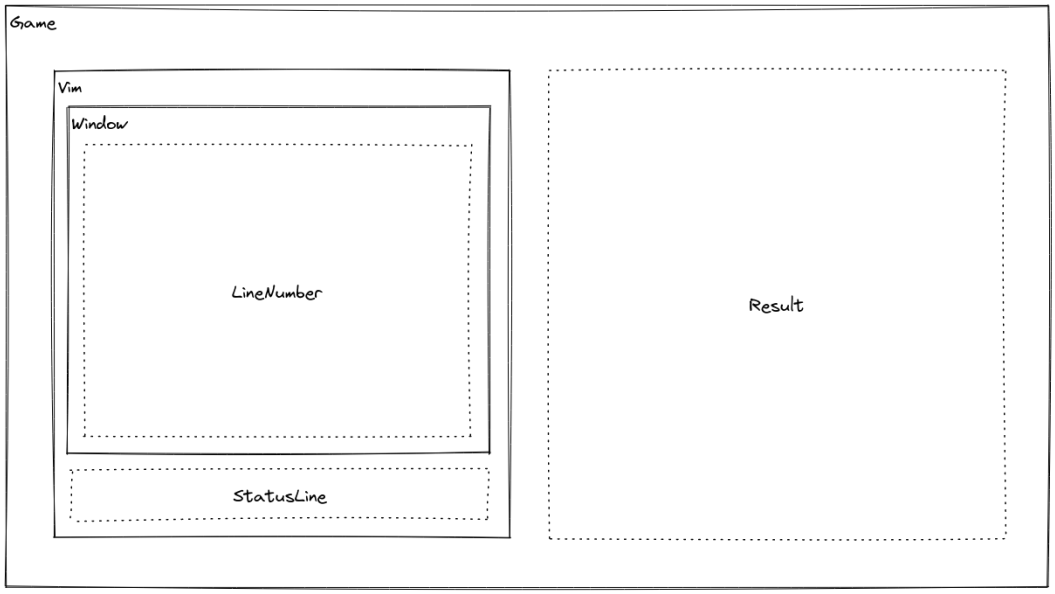
De frontend van de applicatie is volledig geschreven in Rust. Dit is mogelijk gemaakt door het gebruik van Yew als framework. Daarbij is er Tailwindcss gebruikt als CSS framework. Tailwindcss is een utility-first CSS framework om snel custom user interfaces te bouwen. Samen met een component based framework als Yew hoeven we zelf geen CSS meer schrijven maar kunnen we direct tailwindcss zijn utility classes toepassen op mijn components. Wat maakt voor een geweldige developer experience.

Om alles van frontend te kunnen compileren en uitvoeren is er trunk gebruikt. Trunk is een WASM web applicatie bundelaar. Het gebruikt een eenvoudige config voor het bouwen en bundelen van WASM, JS snippets en andere assets (images, css, scss) via een source HTML bestand. Met een simpel commando als `trunk build` bouwt hij de hele frontend en met `trunk serve --open` draait hij een lokale dev server.

Bij het laden van de startpagina haalt de frontend een willekeurig code fragment op via de API. De API haalt de fragmenten op uit een SQLite database. Om de database in sync te houden met de Rust code werd er diesel gebruikt als ORM framework.

3.4 Werking frontend

De startpagina is geïnspireerd door mijn favoriete editor vim. Zo krijgt u een simpele versie van vim te zien als editor voor het typen. Hieronder ziet u een schema van de compositie van de belangrijkste componenten.



`Game`: rendert op basis van de game status `Vim` of `Result`

`Vim`: de tekst editor voor de code en bevat `Window` & `StatusLine` als children

`Window`: rendert de tekst samen met `LineNumber`

`StatusLine`: toont live statistieken zoals de huidige taal, tijd, WPM, progress

`Result`: toont alle statistieken op het einde van de game

Het bouwen van de interface was redelijk eenvoudig, maar om de speed typing test te doen werken was het verassend moeilijk. Het duurde enkele iteraties tot de logica goed zat. Om de gebruiker het idee te geven dat hij tekst over typt zijn er vier html elementen gebruikt: een cursor met het huidige karkater, de correct getypte tekst, de foute getypte tekst en de resterende tekst.

```
Text {
  cursor: char,
  remaining: String,
  correct: String,
  wrong: String,
}
```

Correct example

```
def test_function():
    print("Hello world!")
```

Wrong example

```
def test_function():
    print("Hello world!")
```

Bij elk keypress event wordt er gecontroleerd of de key gelijk is aan het volgende karakter. Indien het gelijk is wordt het huidige karakter van de cursor toegevoegd aan de string met de correcte karakters en de cursor schuift een karakter op. Hetzelfde geldt als men een verkeerde key indrukt maar de cursor wordt dan toegevoegd aan de string met de foutieve karakters. Als de gebruiker de "Backspace" toets indrukt kan hij zijn foute karakters verwijderen tot de string leeg is.

Dit was het basisidee, de moeilijkheid van het bouwen lag vooral aan hoe we efficiënt de variabelen van de tekst en de bijhorende statistieken in een state kunnen opslaan zodat elk component slechts rendert als het nodig is.

Mijn eerste oplossing was het gebruik van globale state met de `use_reducer` hook. Als u vertrouwd bent met React zijn de meeste hooks zoals `use_reducer`, `use_state`, `use_effect` overgenomen in Yew. Indien niet leg ik het kort even uit. Om simpele variabelen op te slaan in een state zoals bijvoorbeeld een boolean gebruikt u de `use_state` hook. Die zorgt ervoor dat de boolean doorheen het component lifecycle hetzelfde blijft tenzij u hem aanpast. Als u hem aanpast dan zal het component opnieuw renderen.

De `use_reducer` hook werkt gelijkaardig maar wordt gebruikt voor complexere states. Zo komt het met een dispatch functie die een argument neemt van het type Action. Wanneer deze wordt aangeroepen, worden de actie en de huidige waarde doorgegeven aan de reducer functie die een nieuwe state berekent en retourneert.

Zo heb ik 5 acties gedefinieert: `NewSnippet`, `Keypress`, `Backspace`, `Tick` en `Reset`. In die acties zit de meeste logica van het spel en kan gebruikt worden in components om de state van het spel aan te passen.

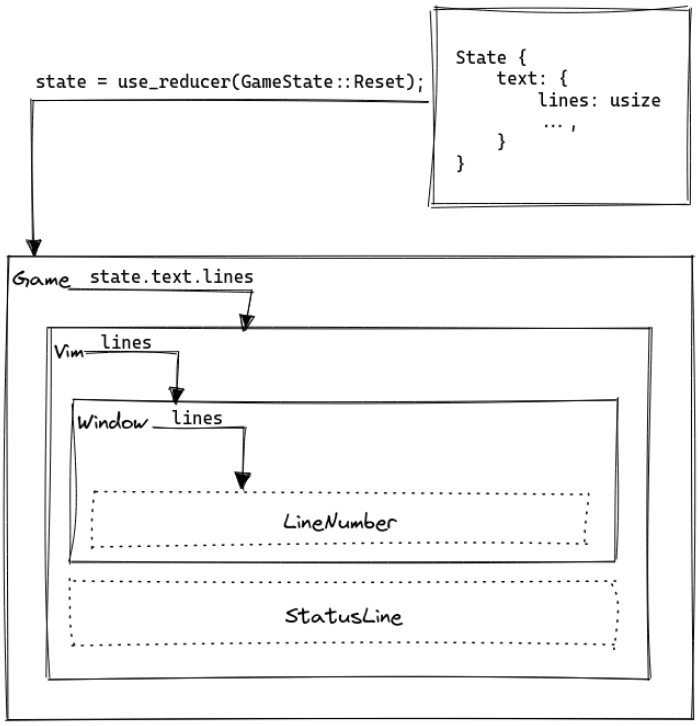
```rust
pub struct Code {
  pub lines: usize,
  pub cursor: Option<char>,
  pub remaining: String,
  pub correct: String,
  pub wrong: String,
}

pub struct Stats {
  pub progress: u8,
  pub mistakes: u8,
  pub wpm: u8,
  pub accuracy: u8,
  pub time: u8,
  pub combos: u8,
}
```

```rust
pub struct GameState {
  pub code: Code,
  pub stats: Stats,
  pub status: Status,
  pub language: String,
}

pub enum Action {
  NewSnippet(Snippet),
  KeyPress(char),
  BackSpace,
  Tick,
  Reset,
}
```

Het probleem met de `use_reducer` hook is dat een component de hele state opneemt. Als de child components een variabel nodig hebben van de state, moeten die als properties worden doorgeven. Hier is niks mis mee tenzij u een diep genest child component hebt die afhankelijk is van de state. Dan heeft u “prop drilling” waarbij u vanuit de parent component de props moet doorgeven aan zijn children die dezelfde props dan doorgeven aan hun children, en zo verder tot dat je bij de gewenste component bent.



Dit werd opgelost door de `use_context` hook te gebruiken samen met de `use_reducer`. Zo kunnen we een `GameStateProvider` component definiëren die de context (state) consumeert en gebruikt kan worden door alle child components. Dit betekent dat we de props niet meer hoefde te 'prop drillen' maar direct kon aanspreken vanuit alle child components onder de `GameStateProvider`. Helaas heeft deze methode ook zijn nadelen:

- Als de state gebruikt wordt in een child component moet de hele state gekloond worden
- Stel een child component gebruikt een variabele A van de globale state, als de hele state wordt aangepast behalve de variabele A zal het child component toch onnodig re-renderen

Uiteindelijk is er de switch gemaakt naar yewdux, een state management library voor Yew die werkt met een globale store/state. Het is verglijkbaar met de populaire React library Redux. Yewdux gebruikt een CoW (clone on write) managementstrategie. Dit betekent dat de state bij elke mutatie een keer wordt gekloond. Door het op deze manier te doen kunnen we beknopt een precieze mutatie uitdrukken zonder extra boilerplate, en gebruik maken van change detection om onnodige re-renders te voorkomen.

Een voorbeeld hiervan is de `use_selector` hook van yewdux, waarmee we een variabele van de state kunnen selecteren en slechts re-renderen als de variabel verandert i.p.v. de hele state.

3.5 Werking backend

De werking van de backend verschilt niet veel met het voorbeeld 'Todo' applicatie die we gemaakt hebben in „Hoe bouwt u een API in Rust?". Opnieuw werd er gekozen voor een simpele SQLite database om de code snippets in op te slaan. Zoals eerder gezien wordt het database schema aangemaakt door migrations uit te voeren met diesel CLI. Dit is een database first aanpak, na wat onderzoek lijkt het me dat diesel.rs jammer genoeg geen andere aanpakken ondersteund zoals code first. Met code first kan u vanuit models die gedefinieerd staan in code automatisch migrations aanmaken die het database schema aanpassen/aanmaken.

In de speed typing database gebruiken we twee tabellen 'languages' en 'snippets'. Die een een op veel relatie met elkaar hebben. De tabel languages bevat de soorten programmeertalen en de tabel snippets de code. De code is hier gewoon opgeslagen als tekst in de tabel snippets. Het is wel belangrijk dat de code geen spaties bevat maar tabs. Daarvoor is er kleine parser tool geschreven om makkelijk code snippets te maken. Het neemt een tekstbestand als input en zet alle spaties en enters om naar de respectievelijke karakters '\t' en '\n'. Zo kan de frontend makkelijk de juiste enters en tabs weergeven.

De API kent een aantal routes:

- GET, POST, DELETE /languages
- GET, POST, DELETE /snippets
- GET /snippets/random
- GET /snippets/{language_id}

Uiteindelijk gebruikt de frontend maar een route '/snippets/random' om een willekeurig code fragment op te halen uit de database.

Om ervoor te zorgen dat we niet eerst alle fragmenten op halen uit de database om dan een willekeurig fragment eruit te selecteren. Wordt er eerst via een SQL-query de snippets tabel willekeurig gesorteerd en het eerste resultaat eruit gepakt. Standaard zal Rust of diesel het random type niet kennen. Gelukkig voorziet diesel de `no_arg_sql_function` macro waarmee u SQL functies kunt mappen naar Rust types.

```
no_arg_sql_function!(
  random,
  sql::types::Integer,
  "Represents the SQL RANDOM() function"
)
```

Zo kunnen we het random type importeren vanuit het schema en gebruiken in mijn SQL-query opgesteld door diesel.

Hoofdstuk 4

Reflectie

Het doel van de module 'Research Project' was om onderzoek te doen naar de vraag „Wat is de huidige status van Rust voor het bouwen van webapplicaties?". In dit hoofdstuk wordt er gereflecteerd op het resultaat van het onderzoek en vergelijken we de bevindingen uit de praktijk. Daarvoor is er contact opgenomen met twee core maintainers van Yew die beide professionele ervaring hebben met het bouwen van webapplicaties in Rust.

De eerste core maintainer is Julius Lungys, hij is een Rust developer bij het FinTech bedrijf Nikulipe. Al hun software is geschreven in Rust, de software is voornamelijk backend gericht maar voor de admin websites gebruiken ze Yew. Julius was ook te gast bij *The Rustacean Station Podcast* [19] met een episode over Yew, waar er interessante inzichten zijn verkregen voor het reflecteren over dit onderzoek.

Cecile Tonglet is de tweede core maintainer, in het dagelijkse leven is ze een Rust developer bij HMI Hydronics. Ze maken hydraulische ventielen, inclusief een 'slimme' ventiel met wat embedded code. Cecile werkt niet aan de embedded software maar werkt aan een web app die draait op het slimme ventiel.

Naast de core maintainers, werd er ook gereflecteerd met Emiel Van Severen, een mastersstudent industrieel ingenieur in de informatica aan de Universiteit van Gent. Hij heeft nog geen professionele ervaring met Rust maar is vertrouwd met het ecosysteem en heeft een aantal hobby projecten in Rust.

Ook was er tijdens het onderzoek aardig wat feedback van de Yew community. Yew heeft een relatief kleine maar zeer actieve community. Zo werd er bijna onmiddelijk feedback gegeven bij problemen of vragen.

4.1 Wat zijn de sterke en zwakke punten van het resultaat uit jouw researchproject?

4.1.1 Sterke

Tijdens het interview met Julius Lungys, werd er de vraag gesteld waarom hij zo enthousiast is over Rust. Hij antwoorde dat de taal alles heeft wat hij wilt, een eigen equivalent van npm (Cargo), ingebouwde code formatter, tests, docs, geen garbage collector en macros.

Cargo is ongetwijfeld een van Rust zijn sterke punten. Het is een geweldige package manager en maakt het beheren van packages zeer eenvoudig. Met Cargo kan u dus makkelijk nieuwe crates developen en publiceren naar `crates.io`. Dit zal dus zeker een positieve impact hebben op de groei van het ecosysteem.

Rust is correct met een krachtig typesysteem, dit betekent dat alle edge cases in de code worden opgevangen. Denk maar aan het unwrappen van een `Option` of `Result` type. Gelukkig heeft Rust een intelligente compiler die suggesties geeft wanneer er iets fout loopt tijdens het compileren. Dit zorgt ervoor als de applicatie compileert, u zeker kan zijn dat de meest voorkomende bugs er uit zijn.

In deelvraag 2.6 „Is Rust productie klaar?" zagen we dat de taal uitstekend presteert bij de benchmarks zowel bij de frontend als backend. Als we de frontend resultaten vergelijken met de populaire Javascript frameworks zien we dat Rust efficiënter omgaat met het geheugen. Wat ook logisch is sinds Javascript een dynamische getypeerde taal is en gebruik maakt van een garbage collector wat zorgt voor veel overhead. Samen met WebAssembly opent dit de deuren om rekenkrachtige applicaties op het web te draaien die ervoor niet mogelijk waren.

4.1.2 Zwakke

Momenteel is het grootste nadeel het beperkte ecosysteem bij het bouwen van een frontend voor webapplicaties. Hoe sneller een bedrijf een web app kan uitbrengen hoe minder kosten ze hebben. Dat is een van de redenen waarom er bijvoorbeeld veel voorgestijlde component libraries op de markt zijn voor Javascript. Yew voorziet al een paar copies van zo'n bekende Javascript libraries maar die zijn allemaal nog te jong om te kunnen gebruiken in productie.

Naast het beperkte ecosysteem hebben de frontend frameworks in Rust vaak nog geen 1.0 versie bereikt en zullen dus in de komende tijd nog zeker wat breaking changes met zich mee brengen.

Samen met Rust is WebAssembly nog jong en volop in beweging. Zoals besproken in de deelvraag 2.6 „Is Rust productie klaar?" is wasm nog niet perfect. Zo hoeven we nog altijd javascript te gebruiken om te praten met de DOM, maar dit is slechts tijdelijk.

Voor dat Julius aan zijn Rust carrière begon was hij een React developer. Wat hem nu stoort is de hoeveelheid boilerplate code dat u nodig heeft in Rust om het zelfde te kunnen bereiken in React. Zo hoeven we bijvoorbeeld voor simpele functies aan te spreken in Javascript libraries gebruiken als `gloo` een high level wrapper rond de `wasm-bindgen` crate die speelt als 'lijm' tussen Rust en Javascript.

4.2 Is 'het projectresultaat' (incl. methodiek) bruikbaar in de bedrijfswereld?

Kort gezegd, ja. De technische demo toonde aan welke stappen er ondernomen moesten worden voor een simpele webapplicatie te bouwen. Deze stappen kunnen we beschouwen als een basis die een bedrijf kan gebruiken voor een webapp te bouwen in Rust. De methodiek werd dan ook afgetoetst bij externen. Julius en Cecile hadden beide geen opmerkingen over de structuur/methodiek van het project.

Julius gebruikt zelf een gelijkaardige structuur in de praktijk. Op zijn werk gebruiken ze een grote mono repo waar all hun services inclusief de web applicaties inzitten. Met als voordeel dat ze een paar crates hebben met code die ze kunnen delen, bijvoorbeeld een crate database entiteiten.

De backend zal wel eerder gebruikt kunnen worden dan de frontend, sinds het gebruikte framework (Yew) nog geen majore versie heeft bereikt. Wat maakt dat er waarschijnlijk nog breaking changes zullen komen.

4.3 Wat zijn de mogelijke implementatiehindernissen voor een bedrijf?

De huidige stabiele release van Yew heeft nog geen SSR. Dit betekent met de huidige CSR, er twee grote nadelen zijn:

- Gebruikers zullen niets kunnen zien totdat de hele WebAssembly bundel is gedownload en de eerste render is voltooid. Dit kan resulteren in een slechte gebruikerservaring als de gebruiker een traag netwerk gebruikt.
- Sommige zoekmachines ondersteunen geen dynamisch gerenderde web content en zij die dat wel doen plaatsen dynamische websites meestal lager in de zoekresultaten.

Het gebrek aan SSR kan dus gezien worden als een implementatiehindernis, indien het bedrijf zijn web app wil optimaliseren.

Een andere implementatiehindernis kan zijn dat het team binnen het bedrijf geen ervaring heeft met Rust. De taal opzich heeft al een steile leercurve, daarnaast moeten ze ook nog bedrijpen hoe WebAssembly werkt. Gelukkig hebben mensen met React ervaring al een voorsprong als ze voor het Yew framework kiezen, sinds het bijna een excate copie is.

4.4 Wat is de meerwaarde voor het bedrijf?

Als een bedrijf de frontend van een applicatie volledig in Rust schrijft, zal dat op dit moment niet veel meerwaarde brengen. Buiten dat als een bedrijf een volledige stack heeft in Rust, ze het makkelijk kunnen vinden om ook hun web applicaties te bouwen in Rust.

Zoals besproken in sectie 2.6 „Is Rust productie klaar?" is de grootste use case momenteel om Rust/WebAssembly te gebruiken waar er grote prestatie vereisten zijn of een gedeelte van de applicatie als de logica in Rust schrijven.

Wat wel een meerwaarde is, is om micro services te schrijven in Rust die gebruikt worden voor

het web. Rust zijn binary grootte zonder te compileren naar WebAssembly is klein. Daarnaast is Rust ook super snel, perfect voor kleine micro services in de cloud. Denk maar bijvoorbeeld aan AWS lambda functies. Emiel wist mij te vertellen dat nu ook Cloudflare workers volledig in Rust geschreven kunnen worden. [20]

4.5 Welke suggesties geven bedrijven en/of community?

Tijdens het zoeken naar externen om te reflecteren, is er ook een bericht geplaatst in de Yew discord server. Zij stelden voor om de wasm code bij de server te comprimeren, zodat het niet de client ∼380kB aan wasm code ongecomprimeerd laat downloaden over het internet. Een andere suggestie van de community was om de `reqwasm` crate te laten vallen, want die is ondertussen toegevoegd aan de `gloo` crate. Dit zou ook de wasm binary grootte kunnen verkleien.

Emiel suggesteeerde om testen te schrijven, Rust heeft daarvoor een heel goed builtin test systeem. Dat had zeker een meerwaarde kunnen zijn om bijvoorbeeld de parser tool uit te werken met behulp van test driven development.

Julius stelde ook voor om de `hooks` te gebruiken van `Trunk`, op die manier zou ik een hook kunnen schrijven die bij het bouwen van de applicatie ook de tailwindcss scripts uitvoert. De huidige manier was via een npm script.

4.6 Suggesties voor een vervolgonderzoek

Vooraleer het onderzoek begon, werd er gevraagd een contractplan in te vullen. Daarin is er opsomming opgelijst van wat het technisch onderzoek resultaat minimaal zal bevatten. Helaas was er tijdens het onderzoek niet genoeg tijd om alles te realiseren. Dit zijn de criterea's die niet gerealiseerd zijn:

1. De applicatie is beveiligd met SSO
2. Er is een profiel pagina aanwezig
3. De statistische resultaten worden opgeslaan in een database

In een vervolgonderzoek zou de applicatie dan ook worden kunnen uitgebreid met deze functionaliteiten. Om de applicatie te beveiligen met SSO, zou ik OAUTH implementeren. Op die manier kunnen gebruikers inloggen met bijvoorbeeld hun Github account. Met een gebruikers account zal ik ook een profiel pagina kunnen aanmaken en de statistische resultaten per gebruiker opslaan in de database.

Tijdens het onderzoek heb ik geen aandacht gegeven aan CI/CD. Ondertussen heb ik al een kleine pipeline opgezet om de frontend te deployen naar een Azure Static Web App en de api naar een Azure API App. Dit is vlug opgezet en kan zeker ook verder onderzocht / geoptimaliseert worden.

Wat ik ook nog niet heb onderzocht is het optimaliseren van de wasm binary.[21] Volgens de community zijn er een aantal stappen die u kan ondernemen om uw binary te verkleinen. Wat belangrijk is, want we hebben gezien bij de benchmark resultaten dat Yew een grotere bundle size over het internet stuurt dan Javascript frameworks.

Als laatste zou ik proberen een javascript module als bijvoorbeeld `Chart.js` gebruiken in het web project. Dit zou mogelijk moeten zijn door de `wasm-bindgen` crate te gebruiken om de functies van de JS library te mappen naar Rust. Met `Chart.js` zou ik dan een grafiek kunnen tonen van de afgelopen resultaten.

Hoofdstuk 5

Advies

5.1 Bruikbaarheid

Een van de grote doelen in 2018 was voor de Rust community om een webtaal te worden. Door zich te richten op WebAssembly, kan Rust nu ook worden uitgevoerd op het web net als JavaScript. Dit betekent niet dat Rust Javascript volledig zal vervangen. Javascript biedt nog altijd meer voordelen bij het bouwen van een web app. Dit is omdat JS een goede keuze is voor de meeste dingen. Het is snel en gemakkelijk om aan de slag te gaan met JavaScript. Bovendien is er een levendig ecosysteem vol met JavaScript ontwikkelaars die ongelooflijk innovatieve benaderingen hebben gecreëerd voor verschillende problemen op het web.

Nu in 2022 zijn er al een aantal tools die het makkelijk maken om Rust te gebruiken op het web. We hebben `webpack` en `trunk` die voor ons de Rust code bundelt en omzet naar wasm. Daarnaast kunnen we eenvoudig met javascript praten met behulp van `wasm-bindgen` en `web-sys`. Toch is het nog verre van perfect. Uiteindelijk willen we de browser API direct kunnen aanspreken vanuit WebAssembly. Een voorgestelde oplossing waar nu aan wordt gewerkt is interface types [9]. Het probleem dat het tracht op te lossen is het vertalen van waarden tussen verschillende types wanneer een module met een andere module praat (of rechtstreeks met een host, zoals de browser). Want de huidige versie van WebAssembly kan alleen maar met nummers praten.

Wanneer WebAssembly dus niet met Javascript of de browser moet praten kan het dus profiteren van de snelle prestaties die met een low level taal als Rust komt.

Buiten de browser kan Rust gewoon worden gebruikt in (micro) services voor het web waar prestaties van cruciaal belang zijn. Het zal wat extra werk vereisen om 'correcte' en geheugen veilige code te schrijven volgens de borrow checker, maar dit zal resulteren in extra prestaties en een robuust systeem.

5.2 Aanbevelingen

Het onderzoek uit deze bachelorproef legde al een stevige basis voor het bouwen van een webapplicatie. In de praktijk moeten er nog een aantal stappen ondernomen worden voor het resultaat te gebruiken in productie.

Een eerste stap in de juiste richting, is het schrijven van testen. Zo voorkomt u dat er veelvoorkomende fouten terrecht komen in productie. Rust beschikt al over een ingebouwd systeem voor het schrijven van testen met `cargo test`. Optioneel kan u de crate `cargo-nextest` gebruiken. Het biedt wat meer functionaliteiten dan de ingebouwde test runner, zoals een mooiere user interface en bij sommige code bases tot 60% sneller dan `cargo test`.

In het technisch onderzoek is `Diesel.rs` gebruikt als ORM. Het gebruik van een ORM zal vaak de achterliggende SQL queries verborgen houden. Er bestaat ook een andere populaire crate genaamd `sqlx` [22]. Het is een puur asynchroon Rust + SQL crate die queries kan valideren voor het compileren. Het doet dit echter niet door een Rust API of DSL (domain-specific language) te bieden voor het bouwen van queries. In plaats daarvan biedt het macro's die gewone SQL als invoer nemen en ervoor zorgen dat deze geldig is voor uw database. De manier waarop dit werkt is dat SQLx tijdens het compileren verbinding maakt met jouw development database om de database zelf uw SQL queries te laten verifiëren (en er info over terug te geven). Als u geen fan bent van een ORM, kan dit een andere optie zijn om vanuit de api met een SQL gebaseerde database te communiceren.

Ook belangrijke stap, is het optimaliseren van de wasm code. Er zijn verschillende technieken hiervoor om uw wasm code zo klein mogelijk te krijgen. Tijdens het onderzoek is hier niet verder op in gegaan maar als we naar de code voorbeelden kijken van Yew, zien we dat zij hun wasm groottes hebben kunnen verkleinen naar ~100kB. [21]

```
[profile.release]
# less code to include into binary
panic = 'abort'
# optimization over all codebase ( better optimization, slower build )
codegen-units = 1
# optimization for size ( more aggressive )
opt-level = 'z'
# optimization for size
# opt-level = 's'
# link time optimization using using whole-program analysis
lto = true
```

Listing 24: Voorbeeld van Yew hun optimalisaties

In sectie 2.3 is het Axum framework vermeld, wat een interessant framework is om naar uit te kijken. Met dat Tokio zo een populair async framework is en Axum deel uitmaakt van het project, zou het een optie kunnen zijn om Actix-web te vervangen door Axum.

Het technisch onderzoek in deze bachelorproef was een klein en simpel project. Bij grotere code bases mag u met Rust lange compilatie tijden verwachten. Daarom is het best op plaatsen waar we moeten compileren, zoveel mogelijk proberen om niet elke keer vanaf nul te compileren. Een mogelijke oplossing hiervoor is cachen.

Om de API te kunnen deployen naar de cloud is er in het technisch onderzoek een Docker container gebruikt. Hier wilt u dus vermijden dat u terug vanaf nul gaat compileren bij het aanpassen van uw code. Ontmoet `cargo-chef`, een cargo-subcommando gemaakt om Rust docker builds te versnelling met behulp van Docker layer caching. Met `cargo-chef` krijgt u tot 5x snellere build times. Hieronder in oplijsting 25 is een voorbeeld hoe `cargo-chef` kan gebruikt worden voor de api crate.

```
FROM ekidd/rust-musl-builder:latest AS chef
USER root
RUN cargo install cargo-chef
WORKDIR /app

FROM chef AS planner
COPY . .
RUN cargo chef prepare --recipe-path recipe.json

FROM chef AS builder
COPY --from=planner /app/recipe.json recipe.json
RUN cargo chef cook --release --target x86_64-unknown-linux-musl \
  --recipe-path recipe.json
COPY . .
RUN cargo build --release --target x86_64-unknown-linux-musl \
  --bin crabtyper-api

FROM alpine AS runtime
ENV DATABASE_URL=/var/lib/snippets.db
ENV PORT=5000
EXPOSE 5000
COPY --from=builder /app/snippets.db /var/lib/snippets.db
COPY --from=builder \
  /app/target/x86_64-unknown-linux-musl/release/crabtyper-api \
  /usr/local/bin/
CMD ["/usr/local/bin/crabtyper-api"]
```

Listing 25: `cargo-chef` voorbeeld

Hoofdstuk 6

Conclusie

Met wat extra werk tegenover andere webprogrammeertalen, biedt Rust snelle prestaties, een rijk typesysteem en correcte code. De prestaties verschillen echter per domein. Indien het gebruikt wordt voor web services te schrijven, dan kan er geprofiteerd worden van de uitstekende prestaties samen met een kleine 'runtime'. Met die reden gebruiken top bedrijven Rust in (micro) services.

Met die zelfde redenen wordt Rust gezien als de taal om te compileren naar WebAssembly. Zoals gezien zijn er twee opties om wasm in de browser te draaien. Bij de eerste optie schrijven we een gedeelte van de applicatie in wasm, die gewoonlijk krachtige prestaties verwacht zonder de browser aan te spreken. Hier benutten we de volledige kracht van wasm samen met een low level taal als Rust die veel efficiënter is dan Javascript. Deze optie zien we dan ook het vaakst terug in productie.

De tweede optie, een volledige webapp schrijven in Rust, verliest prestaties. Sinds wasm nog niet direct de browser API kan aanspreken en dus hiervoor Javascript gebruikt. Toch hebben we gezien dat een framework als Yew, betere benchmark resultaten haalt tegenover populaire Javascript frameworks. Terwijl Julius en Cecicle beiden Yew gebruiken in productie, blijft het over het algemeen bij hobby projecten door het jonge framework en ecosysteem er rondt.

We kunnen dus concluderen dat de huidige status van Rust voor het bouwen van webapplicaties nog volop in ontwikkeling is.

Hoofdstuk 7

Referenties

[1] „2021 Developer Survey.” (2022), adres: https://insights.stackoverflow.com/survey/2021.

[2] R. Team. „Rust.” (2022), adres: https://www.rust-lang.org/.

[3] S. Klabnik en C. Nichols. „What is ownership?” (2022), adres: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.

[4] S. Klabnik en C. Nichols, *The Rust Programming Language*. 2022. adres: https://doc.rust-lang.org/book/.

[5] Mozilla. „JavaScript modules.” (), adres: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules.

[6] S. Klabnik en C. Nichols. „Managing Growing Projects with Packages, Crates, and Modules.” (2022), adres: https://doc.rust-lang.org/book/ch07-00-managing-growing-projects-with-packages-crates-and-modules.html.

[7] „Native client.” (), adres: https://developer.chrome.com/docs/native-client/welcome-to-native-client/.

[8] Redactie. „Wat is WebAssembly en wat kun je er mee?” (2020), adres: https://techacademy.id.nl/blog/wat-is-webassembly-en-wat-kun-je-er-mee/.

[9] L. Clark. „WebAssembly interface types.” (21 aug 2019), adres: https://hacks.mozilla.org/2019/08/webassembly-interface-types/.

[10] (), adres: https://wasmtime.dev/.

[11] wasi. „The WebAssembly Interface.” (2022), adres: https://wasi.dev/.

[12] „Getting started.” (), adres: https://yew.rs/docs/next/getting-started/introduction.

[13] „Getting started.” (), adres: http://diesel.rs/guides/getting-started.html.

[14] „Documentation.” (), adres: https://actix.rs/docs/.

[15] G. Dreimanis. „9 Companies that use Rust in Produtcion.” (18 nov 2020), adres: https://serokell.io/blog/rust-companies.

[16] S. Miller en C. Lerche. „Sustainability with Rust.” (11 feb 2022), adres: https://aws.amazon.com/blogs/opensource/sustainability-with-rust/.

[17] „Web Frameworks Benchmarks." (8 feb 2021), adres: `https://www.techempower.com/benchmarks/#section=data-r20&hw=ph&test=composite`.

[18] A. Turner. „Made with WebAssembly." (2022), adres: `https://madewithwebassembly.com/all-projects`.

[19] A. Wyma en J. Lungys, *The Rustacean Station Podcast*, 7 jan 2022.

[20] S. Manuel. „Native Rust support on Cloudflare Workers." (9 sep 2021), adres: `https://blog.cloudflare.com/workers-rust-sdk/`.

[21] „Shrinking .wasm size." (2022), adres: `https://rustwasm.github.io/docs/book/game-of-life/code-size.html`.

[22] launchbadge. „SQLx." (2022), adres: `https://github.com/launchbadge/sqlx`.

[23] „WebAssembly." (), adres: `https://webassembly.org/`.

[24] L. Clark. „Making WebAssembly better for Rust & for all languages." (14 mrt 2018), adres: `https://hacks.mozilla.org/2018/03/making-webassembly-better-for-rust-for-all-languages/`.

[25] L. Clark. „Rust 2018 is here... but what is it?" (6 dec 2018), adres: `https://hacks.mozilla.org/2018/12/rust-2018-is-here/`.

[26] R. community. „The cargo book." (2022), adres: `https://doc.rust-lang.org/cargo/`.

[27] L. Clark. „Standardizing WASI: A system interface to run WebAssembly outside the web." (27 mei 2019), adres: `https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/`.

[28] (), adres: `https://bytecodealliance.org/`.

Bijlage A

Benchmarks

Name Duration for...	vanillajs	solid-v1.3.3	vue-v3.2.26	yew-v0.19.3	angular-v13.0.0	react-v17.0.2	blazor-wasm-v6.0.1
Implementation notes	772						
create rows creating 1,000 rows	85.1 ± 4.5 (1.00)	88.8 ± 6.7 (1.04)	104.3 ± 8.6 (1.23)	140.9 ± 3.1 (1.65)	123.6 ± 11.1 (1.45)	127.6 ± 5.7 (1.50)	256.9 ± 1.6 (3.02)
replace all rows updating all 1,000 rows (5 warmup runs).	82.1 ± 1.0 (1.00)	83.6 ± 0.7 (1.02)	91.0 ± 0.9 (1.11)	138.4 ± 1.2 (1.69)	97.2 ± 2.2 (1.18)	105.9 ± 1.3 (1.29)	212.3 ± 3.3 (2.59)
partial update updating every 10th row for 1,000 rows (3 warmup runs). 16x CPU slowdown.	159.3 ± 1.7 (1.00)	162.8 ± 3.7 (1.02)	179.1 ± 4.1 (1.12)	176.6 ± 1.7 (1.11)	169.8 ± 1.7 (1.07)	206.9 ± 3.0 (1.30)	638.5 ± 2.7 (4.01)
select row highlighting a selected row. (no warmup runs). 16x CPU slowdown.	19.7 ± 1.0 (1.00)	23.3 ± 2.2 (1.18)	32.7 ± 1.8 (1.66)	29.6 ± 1.1 (1.50)	49.6 ± 1.8 (2.52)	93.3 ± 2.3 (4.74)	527.8 ± 1.8 (26.80)
swap rows swap 2 rows for table with 1,000 rows. (5 warmup runs). 4x CPU slowdown.	48.2 ± 0.8 (1.00)	48.9 ± 1.0 (1.01)	49.4 ± 0.7 (1.02)	52.1 ± 1.1 (1.08)	335.8 ± 4.4 (6.96)	339.5 ± 4.7 (7.04)	169.0 ± 0.9 (3.50)
remove row removing one row. (5 warmup runs).	24.4 ± 0.7 (1.00)	25.5 ± 0.8 (1.05)	25.6 ± 0.6 (1.05)	25.7 ± 0.6 (1.05)	25.1 ± 0.8 (1.03)	26.8 ± 0.8 (1.10)	54.1 ± 0.3 (2.22)
create many rows creating 10,000 rows	861.5 ± 48.4 (1.00)	950.4 ± 20.3 (1.10)	1,021.6 ± 17.1 (1.19)	1,964.7 ± 20.7 (2.28)	1,104.6 ± 7.9 (1.28)	1,378.0 ± 15.6 (1.60)	2,242.6 ± 34.2 (2.60)
append rows to large table appending 1,000 to a table of 10,000 rows. 2x CPU slowdown.	182.4 ± 2.9 (1.00)	185.8 ± 1.4 (1.02)	208.4 ± 4.6 (1.14)	297.6 ± 5.7 (1.63)	253.2 ± 4.1 (1.39)	260.9 ± 5.5 (1.43)	497.6 ± 2.8 (2.73)
clear rows clearing a table with 1,000 rows. 8x CPU slowdown.	48.5 ± 1.3 (1.00)	54.5 ± 1.7 (1.12)	65.2 ± 1.2 (1.34)	115.4 ± 2.3 (2.38)	141.9 ± 2.9 (2.92)	75.7 ± 1.8 (1.56)	144.0 ± 2.3 (2.97)
geometric mean of all factors in the table	1.00	1.06	1.19	1.53	1.77	1.90	3.72
compare: Green means significantly faster, red significantly slower	compare	compare	compare	compare	compare	compare	compare

Figuur A.1: Confidence interval

Name	vanillajs	solid-v1.3.3	vue-v3.2.26	yew-v0.19.3	angular-v13.0.0	react-v17.0.2	blazor-wasm-v6.0.1
consistently interactive a pessimistic TTI - when the CPU and network are both definitely very idle. (no more CPU tasks over 50ms)	1,955.6 ± 1.4 (1.04)	1,955.7 ± 0.1 (1.04)	2,105.5 ± 1.0 (1.12)	1,880.0 ± 0.5 (1.00)	2,817.2 ± 23.7 (1.50)	2,555.6 ± 3.0 (1.36)	2,795.9 ± 16.5 (1.49)
main thread work cost total amount of time spent doing work on the main thread. includes style/layout/etc.	156.0 ± 13.2 (1.05)	148.4 ± 18.1 (1.00)	188.0 ± 9.3 (1.27)	193.6 ± 103.1 (1.30)	324.3 ± 5.0 (2.18)	201.8 ± 17.2 (1.36)	1,950.6 ± 25.0 (13.14)
total kilobyte weight network transfer cost (post-compression) of all the resources loaded into the page.	150.3 ± 0.0 (1.00)	149.7 ± 0.0 (1.00)	195.3 ± 0.0 (1.30)	331.2 ± 0.0 (2.21)	294.5 ± 0.0 (1.97)	274.4 ± 0.0 (1.83)	1,154.8 ± 0.0 (7.71)
geometric mean of all factors in the table	1.03	1.01	1.23	1.42	1.86	1.50	5.32

Figuur A.2: Startup metrics

Name	vanillajs	solid-v1.3.3	vue-v3.2.26	yew-v0.19.3	angular-v13.0.0	react-v17.0.2	blazor-wasm-v6.0.1
ready memory Memory usage after page load.	1.5 (1.00)	1.5 (1.02)	1.7 (1.14)	1.6 (1.05)	2.2 (1.51)	1.8 (1.19)	2.1 (1.42)
run memory Memory usage after adding 1000 rows.	1.4 (1.00)	2.2 (1.55)	3.4 (2.35)	2.0 (1.35)	4.2 (2.94)	4.5 (3.14)	4.6 (3.15)
update every 10th row for 1k rows (5 cycles) Memory usage after clicking update every 10th row 5 times	1.5 (1.00)	2.3 (1.55)	3.4 (2.36)	2.0 (1.34)	4.3 (2.98)	5.1 (3.52)	4.6 (3.14)
replace 1k rows (5 cycles) Memory usage after clicking create 1000 rows 5 times	1.6 (1.00)	2.5 (1.57)	3.5 (2.24)	2.0 (1.25)	4.8 (3.05)	5.0 (3.18)	4.7 (3.01)
creating/clearing 1k rows (5 cycles) Memory usage after creating and clearing 1000 rows 5 times	1.2 (1.00)	1.5 (1.20)	1.7 (1.36)	1.4 (1.15)	2.7 (2.21)	2.2 (1.82)	2.5 (2.02)
geometric mean of all factors in the table	1.00	1.36	1.81	1.22	2.46	2.38	2.44

Figuur A.3: Memory