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

Branco Bruyneel

18 mei 2022

Woord vooraf

Ter afsluiting van mijn opleiding New Media & Creative Technologies aan de Hogeschool West-Vlaanderen te Kortrijk, schreef ik deze bachelorproef. In het 4e 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 test applicatie, waar een gebruiker zo snel mogelijk een random 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 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

Samenvatting of abstract (mag in het Engels): MAX 1 halve A4-pagina:

Je beantwoordt in de samenvatting kort en bondig een viertal vragen:

- Wat is de onderzoeksvraag?
- Wat was jouw onderzoek?
- Welke elementen spelen een grote rol (zowel positief als negatief) bij de evaluatie van het onderzoek?
- Welke elementen zijn belangrijk bij jouw advies?
- Het besluit wordt kort samengevat.

Inhoudsopgave

Lijst van figuren

Lijst met afkortingen

Verklarende woordenlijst

Hoofdstuk 1

Inleiding

Hoofdstuk 2

Research

2.1 Wat is Rust?

Rust is een gecompileerde multi-paradigma programmeertaal bedacht door Graydon Hoare en is begonnen als een project van Mozilla Research. Geïnspireerd door de programmeertalen C en C++, maar kent weinig syntactische en semantische gelijkenissen tegenover deze talen. Rust focust zich met name op snelheid, veiligheid, betrouwbaarheid en productiviteit. 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 systeem 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.

```
1  fn main() {
2    println!("Hello, world!");
3  }
```

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 als 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 null pointer wil dereferentieren (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;
  // If name was not None, it would print here.
  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 wil je niet alle gevallen expliciet afhandelen en wil je slechts een patroon afhandelen terwijl je de rest negeert. In dat geval kan je `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 hoef je 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 loopt. 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.

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.

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.

De heap is minder georganiseerd: als je gegevens op de heap zet, vraag je 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, kun je de pointer op de stack opslaan, maar als je de eigenlijke gegevens wilt hebben, moet je de pointer volgen.

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 kan je niet elke variabele wegschrijven naar de stack, daarom is er een onderscheid tussen simpele en complexe types. Bij simpele types ken je 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:

- Integer
- Floating-point
- Boolean
- Character
- Tuple
- Array (ze hebben dus een vaste grootte in Rust)

Nu de begrippen zijn opgeklaard kunnen we kijken naar het ownership systeem. Het systeem 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. 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

Als we de regels volgen, wordt de variabele s owner over de string literal `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 systeem werkt in Rust. Dit was een zeer simpel voorbeeld en in de realiteit komen er natuurlijk nog wat eigenaardigheden bij kijken.

2.1.3 Ecosysteem

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2.2 Wat is WebAssembly & Hoe werkt het?

Om interactieve webapps te creëren gebruik je tegenwoordig javascript. Ondanks de succesvolle inspanningen van browsermakers om hun javascript-engines in elke versie weer wat efficiënter te maken, is dat voor veel toepassingen nog niet genoeg. Google dan kwam in 2011 met Native Client (NaCI), een sandbox voor het efficiënt en veilig uitvoeren van gecompileerde C en C++ code in de browser, onafhankelijk van het besturingssysteem van de gebruiker. Het bracht prestaties en low-level controle van native code naar moderne webbrowsers, zonder de veiligheid en portabiliteit van het web op te offeren. [1]

Mozilla wilde de platformonafhankelijkheid van javascript echter niet verlaten, en begon daarom in 2013 aan een andere aanpak: asm.js, een subset van javascript die browsers heel efficiënt kunnen uitvoeren. Je compileert dan een webapp uit een taal zoals C naar asm.js, en je browser voert dit dan als gewone javascript uit.

Het voordeel van asm.js is dat het gewoon al in alle webbrowsers werkte, maar Mozilla botste tegen de snelheidsgrenzen van javascript aan. Omdat javascript een tekstformaat heeft, vraagt het parsen veel rekenkracht, zeker op mobiele toestellen met een wat zwakkere processor. En zo werd in 2015 WebAssembly (afgekort Wasm) geboren, een binair instructieformaat voor een stack-gebaseerde virtuele machine in je webbrowser. Het is ontworpen als een overdraagbaar compilatiedoel voor programmeertalen, waardoor het gebruik op het web mogelijk wordt voor client- en servertoepassingen. [2] [3]

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 W3C WebAssembly Community Group 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.

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 Typscript 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 WebAssembly. 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 WebAssembly is om verschillende soorten waarden in en uit functies te krijgen. Dat komt omdat WebAssembly momenteel slechts twee types kent: integers en floating point getallen.

Dit betekent dat je niet zomaar een string in een WebAssembly functie kunt stoppen. In plaats daarvan moet je een aantal stappen doorlopen om een string voor te stellen als getallen. Als je complexere types hebt, zul je 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 je Rust code, zal het automatisch de code maken die nodig is (aan beide kanten) om complexere types te laten werken.

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

Om efficiënt webapplicaties te bouwen heb je natuurlijk 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 over 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 offloaden 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 bouw je een web app 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 bouw je een Webapplicatie 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.

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 je Rust al hebt staan maak dan zeker dat je de laatste versie hebt door rustup update uit te voeren.

```
1 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 je development environment.

```
1 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, voer je het initiele project uit met de cargo build tool. Na de output van de het build process, zou je 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

```rust
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.

```html
<!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 je 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 je 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 deze tutorial 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.

```rust
#[function_component]
pub fn App() -> Html {
    let tasks = vec![
        html! {
          <Task
            id={"1"}
            title={"Take dog out for a walk"}
            completed={true}
          />
        },
        html! {
          <Task
            id={"2"}
            title={"Feed the cats"}
            completed={false}
          />
        },
        html! {
          <Task
            id={"3"}
            title={"Water the plants"}
            completed={false}
          />
        },
    ];
    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 bouw je een API in Rust?

Volgend op de Todo applicatie die we gebouwd hebben in "Hoe bouw je een web app 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.

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 je project te helpen beheren. Omdat het een standalone binary is, en geen directe invloed heeft op de code van je 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 je nu de API kunnen opstarten met `cargo run` en bij een GET request naar `localhost:8080/hello` `"Hello world!"` als repons te zien krijgen.

```
1  #[get("/task")]
2  async fn get_tasks() -> Result<HttpResponse, Error> {
3    todo!()
4  }
5
6  #[post("/task")]
7  async fn add_task() -> Result<HttpResponse, Error> {
8    todo!()
9  }
```

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, Google, Microsoft kunnen dit beamen sinds ze ondertussen al een aantal jaar Rust gebruiken. Zo heeft Amazon in 2018 Firecracker gelanceerd. 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 developen 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 onderzoeksvraag "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 heb je 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 gebruik je gewoonlijk een web framework. Het huidige rust ecosysteem heeft een aantal stabiele web frameworks ter beschikking. Uit de onderzoeksvraag "Hoe bouw je een API in Rust?" hebben we gekozen voor Actix-web te gebruiken als framework. Actix-web voorziet alles wat je 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).

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 gedeelde 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 je Javascript blijven 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:

- 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. Toch kan een bedrijf er een voordeel uithalen. Bijvoorbeeld als hun volledige stack geschreven is in Rust, kunnen ze delen van hun codebase zoals database entiteiten hergebruiken in de frontend. Nog een voordeel is dat beide teams maar een taal hoeven te kennen en zo elkaar kunnen helpen bij veelvoorkomende fouten.

Het heeft natuurlijk ook zijn nadelen. De compileer tijd kan soms op lopen tot enkele minuten.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Hoofdstuk 3

Technisch onderzoek

Mijn 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 volgde ik de kleine code voorbeelden mee in een eigen Github repository. 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 kon ik mijn kennis over structs, ownership, enums en pattern matching, error handling, lifetimes en tests praktisch toepassen.

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 je rekening moet houden met het geheugen.

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

Vervolgens heb ik een keuze moeten maken welke frameworks ik zou gebruiken voor zowel front als backend. Na wat onderzoek dat beschreven staat in “Welke front- end backend frameworks zijn er ter beschikking?” heb ik gekozen voor Yew als frontend en Actix.web samen met diesel.rs als backend framework.

3.1 Omschrijving

Om de huidige status van Full Stack web development in te onderzoeken heb ik gekozen voor een speed typing test webapp te maken.

Het is een simpele applicatie waarbij de gebruiker een random code snippet krijgt die hij zo snel mogelijk moet overtypen. Tijdens het typen wordt er live zijn tijd en words per minute (WPM) bijgehouden. Als de gebruiker de volledige snippet 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.

Ik had voor ogen om bij dit project nog een aantal features 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 ben ik daar niet toe tot geraakt. Het leren van Rust en WebAssembly nam meer tijd in beslag dan verwacht. Sinds ik geen voorkennis had van Rust of een gelijkaardige systeem level programmeertaal was dit project al een hele uitdaging op zich.

3.2 Opbouw/structuur

De frontend van de applicatie is volledig geschreven in Rust. Dit is mogelijk gemaakt door het gebruik van Yew als framework. Daarbij heb ik 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 hoeft ik zelf geen CSS meer schrijven maar kan ik 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 heb ik 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 --serve` draait hij de lokale dev server.

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

3.3 Werking frontend

De startpagina is geïnspireerd door mijn favoriete editor vim. Zo krijg je een simpele versie van vim te zien als editor voor het typen. Hieronder zie je 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. Om de gebruiker het idee te geven dat hij tekst over typt heb ik vier html elementen gebruikt: een cursor met het huidige karkater, de correct getypte tekst, de foute getypte tekst en de resterende tekst.

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 waarmee ik aan de slag ging. De moeilijkheid van het bouwen lag vooral aan hoe ik efficiënt de variabelen van de tekst en de bijhorende statistieken in een state kan 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 je 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. Voor simpele variabelen op te slaan in een state zoals bijvoorbeeld een boolean gebruik je de `use_state` hook. Die zorgt ervoor dat de boolean doorheen het component lifecycle hetzelfde blijft tenzij je hem aanpast. Als je 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.

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 je een diep genest child component hebt die afhankelijk is van de state. Dan heb je "prop drilling" waarbij je 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 heb ik opgelost door de `use_context` hook te gebruiken samen met de `use_reducer`. Zo kon ik een `GameStateProvider` component definiëren die de context (state) consumeert en gebruikt kan worden door alle child components. Dit betekent dat ik 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.

Zo ben ik uiteindelijk geland op 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.

De werking van de backend verschilt niet veel met het voorbeeld "Todo" applicatie die we gemaakt hebben in "Hoe bouw je 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 je 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 heb ik 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, ik lijst ze hier even op:

- 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 random code snippet op te halen uit de database. Om ervoor te zorgen dat we niet eerst alle snippets op halen uit de database om dan een random snippet eruit te selecteren. Heb ik eerst via een SQL-query de snippets tabel random 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 je SQL functies kunt mappen naar Rust types.

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

3.4 Werking backend

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 reflecteer ik op het resultaat van het onderzoek en vergelijk de bevindingen uit de praktijk. Daarvoor heb ik contact opgenomen met twee core maintainers van het gebruikte web framework genaamd Yew. Zij hebben beide ook professionele ervaring 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 met 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"met een episode over Yew, waar ik interessante inzichten gekregen heb over dit onderzoek.

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

Naast de core maintainers heb ik ook gereflecteerd met Emiel Van Severen, een laatstejaars-student Industrieel Ingeineur in de Informatica. Hij heeft nog geen professionele ervaring met Rust maar is zich zeer bekend in het ecosysteem en heeft een aantal hobby projecten in Rust.

Ook was er tijdens het onderzoek ook aardig wat feedback van de Yew community. Yew heeft een relatief kleine maar zeer actieve community. Zo kreeg ik in hun discord instant feedback bij problemen of vragen.

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

4.1.1 Sterke

rust package management

sterk typesystem + compiler

rust verzekert dat alle edge cases worden afgehandeld

bv het unrwappen van een Result of option

performance gewijs hetzelfde

4.1.2 Zwakke

veel boiler plate om het zelfde te kunnen bereiken dan met react + typescript

ecosysteem nog niet ontwikkeld

de frameworks hebben vaak nog geen 1.0 versie

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

cargo workspace

er zijn geen testen

gzipped content

4.3 Wat zijn de mogelijke implementatiehindernissen voor een bedrijf?

4.4 Wat is de meerwaarde voor het bedrijf?

als het bedrijf al rust gebruikt bij hun andere services

4.5 Welke alternatieven/suggesties geven bedrijven en/of community?

4.6 Is er een economische meerwaarde aanwezig?

4.7 Wat zijn jouw suggesties voor een (eventueel) vervolgonderzoek?

ci/cd deployment

Hoofdstuk 5

Advies

Hoofdstuk 6

Conclusie

Hoofdstuk 7

Referenties

Hoofdstuk 8

Bijlages