RUST: ÉÉN TAAL, EINDELOZE MOGELIJKHEDEN

Jarne Vercruysse

Graduaat in het Programmeren

Promotor: Toto De Brant

HOGENT
Academiejaar 2024–2025

Departement IT en Digitale Innovatie

HO GENT

Woord vooraf

Tijdens mijn stage bij **Alkmist** kreeg ik een concrete onderzoeksvraag voorgelegd. Het platform van het bedrijf is opgebouwd uit een robuuste Rust-backend met een frontend van HTMX en Alpine.js. De vraag was of het platform aan waarde zou winnen door de frontend te moderniseren met een framework als Svelte of Solid.js.

Dit bracht me echter al snel tot het inzicht dat een overstap naar een ander JavaScript-framework de kern van de zaak ongemoeid zou laten: de technologische en conceptuele kloof tussen de Rust-backend en de JavaScript-frontend. De echte winst, zo redeneerde ik, lag niet in het kiezen van een *ander* JavaScript-framework, maar in het verenigen van de volledige stack onder één taal.

Om deze hypothese te toetsen, heb ik een gerichte proof-of-concept (POC) ontwikkeld: de **„sharespace”**, een klein maar functioneel bestandsdeelplatform. Hierin zijn zowel de backend als de frontend volledig in Rust geschreven. Met het **Leptos**-framework ontstond een razendsnelle, reactieve frontend die dankzij **Cargo Leptos** naadloos samensmolt met de backend. Het resultaat was een project met één taal, één build-tool en perfect gedeelde datatypes, wat de ontwikkelervaring aanzienlijk vereenvoudigde.

Een belangrijke les uit dit traject was de kracht van focus. Door te beginnen met één afgebakende feature bleef het project beheersbaar en kon ik de technologie diepgaand doorgronden. Een gestructureerde Git-flow en goede documentatie bleken hierbij onmisbare hulpmiddelen.

Tot slot wil ik **Alkmist** bedanken voor de unieke kans en de vrijheid die ze me boden om dit onderzoek uit te voeren. Een speciaal woord van dank gaat uit naar mijn promotor, **Toto De Brant**, voor zijn deskundige begeleiding en waardevolle feedback. Ten slotte bedank ik mijn vriendin, **Ise**, voor haar onuitputtelijke steun en haar rol als onmisbaar klankbord.

Samenvatting

Deze graduaatsproef onderzoekt de haalbaarheid van een full-stack applicatie die volledig in Rust is ontwikkeld. Het doel was om te bepalen in welke mate Rust ingezet kan worden als enige programmeertaal voor zowel frontend- als backendontwikkeling. Als casus werd een bestandsdeelplatform ontwikkeld met het Leptos-framework voor de frontend, dat compileert naar WebAssembly en zich richt op fijnmazige reactiviteit. De backend werd eveneens in Rust gebouwd, waardoor de volledige applicatie vanuit één codebase functioneert.

De centrale onderzoeksvraag luidde: *Hoeveel ontwikkelwerk is vereist om een volledige applicatie in Rust te realiseren, en welke voordelen of beperkingen brengt deze aanpak met zich mee?*

De methodologie bestond uit de herimplementatie van de tool ShareSpace, uitgevoerd binnen de context van een stageproject bij Alkmist. De resultaten tonen aan dat de initiële ontwikkelingskost hoger ligt door de leercurve van de taal, maar dat de voordelen op het gebied van veiligheid, foutafhandeling, consistente tooling en onderhoudbaarheid significant zijn. Bovendien maakt het Rust-ecosysteem het mogelijk om met technologieën als Tauri dezelfde codebase in te zetten voor meerdere platformen. Deze bevindingen onderstrepen het potentieel van Rust als een volwaardige full-stack oplossing binnen moderne softwareontwikkeling.

Inhoudsopgave

Inleiding

De keuze van een programmeertaal beïnvloedt de kwaliteit, betrouwbaarheid en onderhoudbaarheid van softwaretoepassingen aanzienlijk. In veel moderne webprojecten wordt een combinatie van talen en frameworks gebruikt, zoals React in combinatie met Node.js. Dit leidt vaak tot een gefragmenteerde ontwikkelervaring, context-switching tussen verschillende technologieën en een verhoogde complexiteit in onderhoud.

Rust is een systeemtaal die ontworpen is met veiligheid, snelheid en expressiviteit als kernwaarden. Dankzij recente ontwikkelingen in het ecosysteem, zoals de frontendbibliotheek Leptos, het backendframework Axum en de native toolkit Tauri, is het mogelijk geworden om volledige applicaties in één taal te bouwen. Dit opent nieuwe perspectieven voor organisaties die hun technologiestack willen vereenvoudigen zonder in te boeten aan prestaties of veiligheid.

In deze graduaatsproef wordt onderzocht of het haalbaar is om een volledige webapplicatie – zowel frontend als backend – in Rust te bouwen. Als praktijkcase werd de functionaliteit van de ShareSpace-module, een component uit de SaaS-applicatie van Alkmist, herwerkt in het kader van een stageopdracht. De centrale vraag is of het gebruik van één taal leidt tot voordelen op het gebied van onderhoudbaarheid en ontwikkelervaring, en of de initiële leercurve opweegt tegen deze structurele voordelen.

1.1. Probleemstelling

Binnen softwareontwikkeling is het gebruik van meerdere programmeertalen voor één applicatie standaardpraktijk. Frontend en backend zijn vaak losgekoppeld, draaien in aparte omgevingen en vereisen specifieke kennisdomeinen. Dit verhoogt de instapdrempel voor nieuwe ontwikkelaars en kan de communicatie tussen teams complexer maken of duplicatie van logica introduceren. Voor organisaties met een beperkte teamgrootte of een focus op stabiliteit en eenvoud, is het waardevol om een eenduidige technologiestack te hanteren. De ShareSpace-module, een feature voor bestandendeling binnen de applicatie, vormt een representatieve casus om de meerwaarde van een full-stack Rust-aanpak te onderzoeken.

1.2. Onderzoeksvraag

De centrale onderzoeksvraag luidt:

> Is het haalbaar en zinvol om een volledige applicatie – zowel frontend als backend – in Rust te bouwen, en welke impact heeft deze keuze op de ontwikkelervaring, onderhoudbaarheid en technische kwaliteit van het eindproduct?

Deze vraag wordt verder opgesplitst in de volgende deelvragen:

- Welke tooling is beschikbaar binnen het Rust-ecosysteem voor full-stack ontwikkeling?
- Wat is de impact van de leercurve voor ontwikkelaars die de overstap maken van JavaScript naar Rust?
- Welke invloed heeft het gebruik van een gedeelde taal op de codekwaliteit en het onderhoud?
- Zijn er structurele of praktische beperkingen verbonden aan het gebruik van Rust als full-stack oplossing?

1.3. Onderzoeksdoelstelling

Het doel van deze graduaatsproef is om op basis van de herimplementatie van de ShareSpace-functionaliteit te evalueren of het technisch en praktisch haalbaar is om een volledige applicatie in Rust te ontwikkelen. Het eindresultaat is een werkend prototype met de volgende kenmerken:

- Een backend geschreven in Rust met het Axum-framework.
- Een frontend geschreven in Leptos, gebruikmakend van server-side rendering (SSR).
- Eén gedeelde codebase voor de frontend en backend.

De evaluatie gebeurt op basis van criteria als ontwikkeltijd, codecomplexiteit, foutgevoeligheid en de mate waarin een modulaire architectuur bereikt kan worden.

1.4. Opzet van deze graduaatsproef

De rest van deze graduaatsproef is als volgt opgebouwd:

- Het hoofdstuk **Stand van zaken** bespreekt het huidige Rust-ecosysteem, met een focus op tooling, frontend-backendintegratie en de rol van WebAssembly.
- Het hoofdstuk **Methodologie** licht de gevolgde werkwijze toe: hoe de proof-of-concept werd opgebouwd, welke tools werden gebruikt en hoe de evaluatie plaatsvond.
- Het hoofdstuk **Conclusie** brengt de bevindingen samen, formuleert een antwoord op de onderzoeksvragen en doet suggesties voor toekomstig onderzoek.

Stand van zaken

Dit hoofdstuk bouwt voort op de inleiding en biedt een overzicht van de huidige mogelijkheden voor full-stack webontwikkeling in Rust. Er wordt stilgestaan bij de rol van WebAssembly, de voordelen van een eentalige technologiestack en de tooling die deze aanpak ondersteunt. Daarnaast worden ook de uitdagingen en de maturiteit van het ecosysteem besproken om de methodologische keuzes in deze proef te kaderen.

2.1. Rust als fundament

Rust is een systeemtaal die ontworpen werd met veiligheid, snelheid en controle als kernwaarden. Een centraal kenmerk van de taal is het *ownershipmodel*, waarmee de compiler veelvoorkomende programmeerfouten – zoals geheugenfouten en *data races* – al tijdens het compileren detecteert, nog voor de code wordt uitgevoerd. Deze eigenschappen maken Rust bijzonder geschikt voor toepassingen waar betrouwbaarheid en prestaties cruciaal zijn, zoals backendservers en infrastructuurcomponenten.

2.2. Frontend en backend in één taal

Dankzij de opkomst van WebAssembly (WASM) is het mogelijk geworden om frontendcode te schrijven in talen zoals Rust en deze te compileren naar een binair formaat dat direct in de browser kan worden uitgevoerd. Dit opent de deur naar een uniforme technologiestack, waarbij niet langer gewisseld hoeft te worden tussen bijvoorbeeld JavaScript voor de frontend en een andere taal voor de backend.

Een sleutelmechanisme dat deze naadloze integratie mogelijk maakt, zijn de *server functions*. Door een functie te annoteren met het `#[server]`-macro, zorgt de Leptos-toolchain ervoor dat deze code enkel op de server bestaat. Op de client-side wordt de functieaanroep automatisch omgezet naar een type-veilige netwerkaanroep naar een API-endpoint dat eveneens automatisch wordt gegenereerd. Dit abstraheert de noodzaak om manueel een API-laag te bouwen, data te serialiseren (bv. naar JSON) en `fetch`-requests te schrijven. De ontwikkelaar kan serverlogica aanroepen alsof het een lokale functie is, wat de frictie tussen client en server na-

genoeg volledig wegneemt.

2.2.1. De Brug tussen Rust en de Browser: Bindings

WebAssembly op zichzelf is een zeer beperkte omgeving. Het heeft geen ingebouwde toegang tot de browser-API's zoals het Document Object Model (DOM) of de `fetch`-API. Om deze kloof te overbruggen, is een "bindings-laag nodig die als vertaler fungeert tussen de Rust/WASM-wereld en de JavaScript-wereld van de browser.

In het Rust-ecosysteem wordt deze functionaliteit geleverd door een gelaagde architectuur van crates:

- **`wasm-bindgen`**: Dit is de fundamentele toolchain die Rust-code analyseert en de nodige JavaScript-"lijmcode"genereert om communicatie tussen WASM en JS mogelijk te maken.
- **`web-sys`**: Bovenop `wasm-bindgen` biedt deze crate directe bindings naar nagenoeg alle web-API's. Omdat interactie met JavaScript door de Rust-compiler als inherent onveilig wordt beschouwd, is deze crate intern gebouwd met `unsafe` code. Dit maakt de API's krachtig, maar in sommige gevallen ook complexer of minder ergonomisch in gebruik dan veiligere alternatieven.
- **`gloo`**: De `gloo`-familie van crates bouwt een veilige en „Rusty" abstractielaag bovenop `web-sys`. Het biedt handige, veilige wrappers voor veelvoorkomende taken zoals het afhandelen van events, timers of HTTP-requests, wat de ontwikkelervaring aanzienlijk verbetert.

Een framework als Leptos focust zich vervolgens primair op het bieden van een performant reactiviteitsmodel. Hoewel Leptos veelvoorkomende taken intern afhandelt, zullen ontwikkelaars voor specifieke interacties met browser-API's die buiten het reactiviteitsmodel vallen, vaak teruggrijpen naar de veilige hulpmiddelen van `gloo`.

2.3. De Kracht van Server-Side Rendering (SSR)

Een kernfunctionaliteit die de full-stack aanpak van Leptos verder versterkt, is Server-Side Rendering (SSR). Dit betekent dat de initiële paginaweergave op de server wordt gerenderd en als HTML naar de browser wordt gestuurd. Dit zorgt voor een zeer snelle eerste laadtijd (*First Contentful Paint*) en is gunstig voor zoekmachine-optimalisatie (SEO). Nadat de HTML is geladen, neemt de WebAssembly-code het over om de pagina interactief te maken, een proces dat bekendstaat als *hydration*.

Leptos biedt hierin geavanceerde controle door verschillende SSR-modi:

- **Synchrone (In-Order) Rendering:** De server wacht tot alle data voor de pagina is geladen en stuurt vervolgens de volledige HTML in één keer.
- **Asynchrone (In-Order) Rendering:** De server stuurt onmiddellijk een HTML-omhulselën streamt de data-afhankelijke content zodra deze beschikbaar komt, in de volgorde van de code.
- **Out-of-Order Rendering:** Deze modus streamt content zodra deze beschikbaar is, onafhankelijk van de volgorde, wat de snelste weergave oplevert.

2.4. Uitdagingen en Overwegingen

Ondanks de voordelen brengt het gebruik van Rust ook uitdagingen met zich mee. Een van de meest significante nadelen is de **lange compileertijd**. De strenge analyses die de Rust-compiler uitvoert om veiligheid te garanderen, kosten tijd. Dit staat in schril contrast met de instantane reactiviteit van geïnterpreteerde talen zoals JavaScript.

Voor frontendontwikkeling wordt deze uitdaging echter grotendeels gemitigeerd. Tools zoals `cargo-leptos` bieden een *hot reloading*-functionaliteit. Wanneer een aanpassing in de view-laag wordt gemaakt, wordt enkel die component opnieuw gecompileerd. Dit verkort de feedback-loop drastisch.

2.5. Tooling en Ecosysteem

Het Rust-ecosysteem voor webontwikkeling is in volle groei. Hoewel het nog niet dezelfde maturiteit heeft als die van gevestigde JavaScript-frameworks, bewijst het zich als stabiel voor interne tools en kleinere projecten. Voor grootschalige, bedrijfskritische SaaS-applicaties is mogelijk verder onderzoek nodig naar de schaalbaarheid op lange termijn.

De leercurve is een andere belangrijke factor. Hoewel het leren van Rust zelf tijdrovend kan zijn, probeert een framework als Leptos de drempel voor webontwikkelaars te verlagen. Een kernaspect van de filosofie van Leptos is dat het bouwt op bestaande webstandaarden, in plaats van ze te vervangen. Zo is de router ontworpen om te werken met fundamentele HTML-elementen zoals links (`<a>`) en formulieren (`<form>`), die vervolgens progressief worden uitgebreid met reactiviteit. Deze aanpak maakt het framework niet alleen herkenbaar voor ontwikkelaars, maar zorgt er ook voor dat de applicatie de basisprincipes van het web respecteert.

2.6. Vooruitblik

De combinatie van Rust en WebAssembly vertegenwoordigt een veelbelovend alternatief voor traditionele webstacks. De verdere maturiteit van het ecosysteem zal zich waarschijnlijk uiten in een toename van gespecialiseerde UI-bibliotheken, verbeterde ontwikkelaarstools en een breder aanbod aan leermateriaal.

Naarmate de voordelen – zoals type-veiligheid over de gehele stack, de abstractie van API-lagen via server functions en de eenvoud van een *single binary deployment* – bekender worden, zullen meer organisaties de overstap overwegen. Deze graduaatsproef draagt bij aan dit opkomende veld door een praktische evaluatie van de huidige mogelijkheden en uitdagingen. Het potentieel om de robuustheid van backend-systemen te verenigen met een moderne, performante frontend-ervaring, vormt de drijfveer achter het onderzoek dat in de volgende hoofdstukken wordt gepresenteerd.

Methodologie

Dit hoofdstuk beschrijft de werkwijze die gevolgd werd tijdens de ontwikkeling van de proof-of-concept, van de initiële technologiekeuze en de evaluatiecriteria tot de beperkingen van het project.

3.1. Een pragmatische en organische aanpak

In tegenstelling tot een traditioneel project met een gedetailleerd stappenplan, werd voor deze proef een meer organische aanpak gehanteerd. De steile leercurve van zowel Rust als het full-stack ecosysteem maakte het moeilijk om vooraf een volledig en accuraat mentaal model van de applicatie te vormen. Daarom werd gekozen voor een proces dat ruimte bood voor exploratie en leren.

De ontwikkeling verliep in drie globale fasen:

1. **Frontend-ontwikkeling**: Eerst werd de focus gelegd op het bouwen van een visueel en functioneel werkende frontend in Leptos. Hierdoor kon de volledige gebruikersflow worden uitgewerkt en konden de reactieve principes van het framework worden verkend. Voor de styling werd gebruikgemaakt van `TailwindCSS` en `DaisyUI`.
2. **Backend-logica**: Vervolgens werd de server-side logica geïmplementeerd. Dit omvatte de functies voor het effectief verwerken van bestand-uploads en -downloads.
3. **Integratie**: Ten slotte werden beide delen naadloos met elkaar verbonden met behulp van Leptos' *server functions*, waardoor de frontend direct en type-veilig met de backend kon communiceren.

3.2. Evaluatiecriteria

Om de haalbaarheid en de voordelen van de full-stack Rust-aanpak te beoordelen, werd de **ontwikkelervaring** als primair, kwalitatief criterium gehanteerd. De evaluatie focuste op de volgende vragen:

- **Efficiëntie van de workflow**: De mogelijkheid om in één enkele codebase te werken, bood een zeer geïntegreerde ervaring. Aanpassingen aan de backend (bv. een datastructuur) en de frontend (de weergave daarvan) konden tegelijkertijd en in dezelfde context worden doorgevoerd.
- **Code-organisatie**: De full-stack opzet maakte het mogelijk om de code per feature te organiseren. Alle logica, types, en componenten voor een specifieke functionaliteit konden worden gegroepeerd, wat de structuur overzichtelijk en onderhoudbaar maakte.
- **Preventie van synchronisatiefouten**: Een specifiek aandachtspunt was hoe de gedeelde datatypes tussen de frontend en de backend synchronisatiefouten tussen de API en de client voorkomen. In een traditionele stack kan een kleine aanpassing aan de API-respons leiden tot een runtime-fout in de frontend. Er werd geëvalueerd hoe dit probleem in de full-stack architectuur per definitie wordt geëlimineerd, omdat een wijziging in een gedeeld type direct een compileerfout veroorzaakt als de andere laag niet wordt aangepast.

3.3. Technische uitdagingen en inzichten

Het vormen van een mentaal model voor een full-stack Rust-applicatie was de grootste conceptuele uitdaging. De reis om tot een werkend prototype te komen, bracht verschillende concrete technische en inzichtelijke leermomenten met zich mee.

3.3.1. Reactiviteit, Closures en Ownership

Een specifiek struikelblok was het werken met asynchrone operaties en *closures* (anonieme functies) als callbacks in de reactieve context van Leptos. Een essentieel inzicht was dat een callback-functie, die pas later wordt uitgevoerd (bijvoorbeeld na een klik), expliciet moet worden opgeslagen in een reactieve primitief, zoals een *signal*. Als dit niet gebeurt, beschouwt de Rust-compiler de closure als ongebruikt en wordt deze onmiddellijk uit het geheugen verwijderd (*dropped*), waardoor de callback nooit wordt uitgevoerd. Dit illustreert hoe Rust's ownershipmodel de manier van denken over UI-events fundamenteel beïnvloedt.

3.3.2. Geheugenbeheer in de praktijk

Het project bood diepgaand inzicht in Rust's geheugenbeheer. De realisatie dat een type als `String`, dat op de *heap* wordt opgeslagen, de `Copy`-trait niet kan implementeren, had directe praktische gevolgen voor hoe data door de applicatie werd doorgegeven. Dit toonde echter ook de kracht van de Rust-compiler. In plaats van

enkel een foutmelding te geven, biedt de compiler vaak gedetailleerde suggesties en uitleg, waardoor het fungeert als een gids die de ontwikkelaar helpt om tot een correcte en veilige oplossing te komen.

3.4. Afbakening en Beperkingen

Het ontwikkelde prototype is een proof-of-concept en geen één-op-één-implementatie van de bestaande ShareSpace-module van Alkmist; hiervoor was meer ontwikkeltijd nodig geweest. De scope van de gerealiseerde functionaliteit omvat de kernfeatures: het uploaden van bestanden en het genereren van een deelbare link.

Daarnaast is het belangrijk te benadrukken dat de bevindingen, met name over ontwikkeltijd en de leercurve, gebaseerd zijn op het perspectief van één ontwikkelaar die tegelijkertijd de technologie aan het leren was. De resultaten zijn dus niet noodzakelijk generaliseerbaar naar een ervaren team van Rust-ontwikkelaars.

Conclusie

Het doel van deze graduaatsproef was om te onderzoeken of een full-stack webapplicatie, volledig ontwikkeld in Rust, een haalbare en zinvolle aanpak is. Op basis van de ontwikkeling van een werkend prototype kan deze vraag volmondig met ”ja”beantwoord worden, zij het met de belangrijke kanttekening dat het een aanzienlijke conceptuele leerinspanning vereist.

4.1. Beantwoording van de Onderzoeksvraag

De resultaten tonen aan dat diepgaande voorkennis van Rust geen strikte vereiste is, maar een sterke motivatie om te leren wel. Het vormen van een mentaal model voor de architectuur is de grootste uitdaging. Een specifiek voorbeeld hiervan is het werken met Server-Side Rendering (SSR). Dit vereist een duidelijk onderscheid tussen code die op de server draait (in de *binary*) en code voor de client (in de *library*). In de praktijk betekent dit dat men via conditionele compilatie moet zorgen dat server-specifieke dependencies niet worden meegecompileerd in de WebAssembly-bundel, om te voorkomen dat de client-applicatie onnodig groot wordt.

Hoewel dit soort concepten de leercurve steiler maken, werkt de befaamde strengheid van de Rust-compiler in het voordeel van de ontwikkelaar. Fouten worden vroeg in het proces opgevangen en de gedetailleerde feedback fungeert als een gids. Het vermogen om systeem-nabije functionaliteit te schrijven met de zekerheid dat de compiler de meest voorkomende fouten voorkomt, is een van de krachtigste en meest gewaardeerde aspecten van de taal.

4.2. De Voordelen van een Geïntegreerde Stack

Het meest significante voordeel van de full-stack aanpak is de eenduidige en geïntegreerde codebase. Het elimineert de noodzaak voor context-switching tussen verschillende talen en ecosystemen. Dit resulteerde in een sterk verbeterde ontwikkelervaring, waarbij aanpassingen aan de backend en frontend tegelijkertijd en in dezelfde context konden worden doorgevoerd.

De architectuur, met name door het gebruik van *server functions*, maakt een aparte API-laag overbodig. Frontend en backend communiceren naadloos via gedeelde, type-veilige datastructuren. Dit versnelt niet alleen de ontwikkelcyclus, maar voorkomt ook een hele klasse van synchronisatiefouten. Het stelt een ontwikkelaar in staat om code per feature te organiseren, wat de consistentie en onderhoudbaarheid van het project ten goede komt.

4.3. Aanbeveling en Toekomstig Werk

De minimalistische en modulaire aard van het Rust-ecosysteem plaatst de ontwikkelaar centraal en laat toe om slanke, performante applicaties te bouwen. Een concrete volgende stap voor dit project zou de integratie met **Tauri** kunnen zijn, waarmee dezelfde Rust-codebase met minimale aanpassingen kan worden verpakt als een native desktopapplicatie.

Voor een jonge ontwikkelaar is de brede horizon van Rust misschien wel het meest aantrekkelijke aspect. Het feit dat de taal deuren opent naar backend-, frontend-, desktop- en zelfs embedded ontwikkeling, betekent dat een investering in Rust een investering is in een toekomst vol mogelijkheden. Juist die onbegrensde potentie vormt de ultieme motivatie om zich volop in deze technologie te verdiepen en vormt de kern van de aanbeveling in deze proef: geef full-stack Rust een serieuze testrit.