



Faculteit Bedrijf en Organisatie

Mag Webpack zich nog steeds de koning der module bundlers noemen?

Maxim Vansteenkiste

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Heidi Roobrouck
Co-promotor:
Cedric Vanhaeverbeke

Instelling: —

Academiejaar: 2021-2022

Eerste examenperiode

Faculteit Bedrijf en Organisatie

Mag Webpack zich nog steeds de koning der module bundlers noemen?

Maxim Vansteenkiste

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Heidi Roobrouck
Co-promotor:
Cedric Vanhaeverbeke

Instelling: —

Academiejaar: 2021-2022

Eerste examenperiode

Deze bachelorproef werd geschreven in functie van het afronden van de opleiding Toegepaste Informatica aan de Hogeschool van Gent. Ook al volg ik de afstudeerrichting mobiele applicaties, waar webapplicaties een deel van zijn, toch is de module bundler nooit aan bod gekomen. Hoewel velen erop vertrouwen, weten weinigen wat ze voor essentiële taken verrichten. Door deze proef hoop ik dat er meer wordt stilgestaan bij het kiezen van een module bundler.

Ik wil graag mijn promotor, Heidi Roobrouck, bedanken om deze scriptie in goede banen te leiden. Ook mijn co-promotor, Cedric Vanhaeverbeke, verdient een dankwoordje. Ik kon steeds rekenen op zijn uitgebreide feedback. Deze twee personen hebben ervoor gezorgd dat dit onderzoek tot een goed einde is gekomen.

Een module bundler of build tool is een essentieel onderdeel in het maken van moderne webapplicaties. Al jaren wordt Webpack gezien als de beste keuze hiervoor. Aangezien er genoeg andere opties zijn, vele met een nieuwe aanpak, is het wel eens tijd om die bewering nog eens na te gaan. In een literatuurstudie wordt de geschiedenis en werking van een module bundler geschetst. Daarna, aan de hand van een vergelijkende studie, worden drie populaire alternatieven tegenover Webpack geplaatst in rechtstreeks duel. Eerst wordt een nieuw project opgezet met de respectievelijke kandidaten. Daarna trachten we drie open-source projecten, elk met zijn eigen moeilijkheden, om te vormen van Webpack naar een tegenkandidaat. Doorheen dit gedocumenteerd proces, wordt er meer uitleg gegeven over de verschillende technologieën die aan bod komen. Tot slot wordt in de conclusie, rekening houdend met verschillende ob- en subjectieve factoren, de titel van deze proef beantwoord.

De wijze waarop een webapplicatie gemaakt wordt, verandert continu. Het lijkt wel of er elke week een nieuw framework uitkomt met een iets andere aanpak dan al degene die hem voorgingen. Eén ding echter blijft al enkele jaren hetzelfde: de nood aan een module bundler is er nog steeds, voor nu toch.

Dit onderzoek tracht de noodzaak, werking en toekomst van de module bundler te schetsen. Dit aan de hand van een paar veel voorkomende implementaties ervan. Het uiteindelijke doel is om te bepalen of de huidige koning van de module bundlers, Webpack, nog steeds het recht heeft om die positie op te eisen.

1.1 Probleemstelling

Dit onderzoek is in de eerste plaats gericht naar mijn co-promotor. Als webdeveloper komt hij vaak voor de keuze van welke module bundler te gebruiken. In de toekomst kan hij die beslissing dus maken aan de hand van deze vergelijkende studie. Daarnaast kan dit onderzoek ook een meerwaarde bieden voor andere webdevelopers, inclusief mezelf.

1.2 Onderzoeksvraag

De hoofdonderzoeksvraag werd al eerder aangehaald: is het nog steeds gewettigd dan Webpack de meest gebruikte module bundler is?

1.3 Onderzoeksdoelstelling

Hoewel de onderzoeksvraag beantwoorden aan de hand van een vergelijkende studie uiteraard het uiteindelijke doel is, hoop ik dat dit werk volgend doel ook verwezenlijkt.

De module bundler is voor velen, mijn co-promotor en mezelf erbij gerekend, iets wat ze gebruiken en nodig hebben maar niet zoveel over nadenken en weten. Vele frameworks om webapplicaties te maken komen met een module bundler ingebouwd. Zo wordt de keuze dus voor je gemaakt. Ideaal voor iemand die zich daar geen zorgen over wil maken maar aangezien de webapplicatie niet werkt zonder, is het de moeite om toch meer te weten erover. Het tweede, meer verborgen doel is dus om de module bundler en zijn werking te ontrafelen.

1.4 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk ?? wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie.

In Hoofdstuk ?? wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen.

In Hoofdstuk ??, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

Een moderne website wordt gemaakt met drie belangrijke technologieën: HTML voor inhoud, CSS voor vormgeving, en Javascript om de pagina interactief te maken. Deze drie kunnen allemaal aan elkaar worden gekoppeld, ergens op een server worden gezet en dan kan een gebruiker een werkende website laden. Ook al zijn de grote web platformen van vandaag, zoals Facebook en Youtube, allemaal gebaseerd op deze technologieën, er zijn veel meer ingrediënten nodig om ze te laten werken.

Voor het maken van een simpele, statische site zijn de drie basistechnologieën van het web alles wat we nodig hebben. Het is echter een heel ander verhaal voor een ontwikkelaar die een complexere site wil maken, een reactieve site, een site die gebruik maakt van open-source packages of die gewoon zijn eigen werk wat makkelijker wil maken. Er zijn duizenden bibliotheken, packages en frameworks die webontwikkeling mogelijk maken op een heel nieuw niveau en eenvoudiger dan ooit tevoren (**npm-no-date**). Dit stelt ons echter voor een nieuw probleem.

Als we een webapplicatie zouden maken met een enkel Javascript bestand zonder afhankelijkheden i.e. geen andere bestanden die gelinkt zijn aan dat Javascript bestand, het linken aan een index.html en tot slot wat CSS toevoegen, zou die site prima werken. Maar wat als we een tweede Javascript bestand introduceren en daar naar verwijzen in het andere bestand. Wat als we een open-source package willen installeren en gebruiken, gedownload van een package-manager zoals NPM? Wat als we SASS willen gebruiken om CSS uit te breiden? Dit zou allemaal niet werken. De browser weet gewoon niet hoe alle verschillende stukjes aan elkaar te plakken. We hebben iets nodig dat alle verschillende Javascript bestanden en zijn afhankelijkheden bundelt, iets dat de SASS bestanden begrijpt en correct laadt, of welk ander bestand dan ook. We hebben een module bundler nodig. In essentie, nemen zij alle verschillende bronbestanden in een project en bundelt die tot een enkel uitvoer bestand

dat de browser begrijpt.

2.1 Geschiedenis

Om de rest van dit hoofdstuk te begrijpen, moet eerst een Javascript module uitgelegd worden. Modulair programmeren splitst een programma op in brokken gebaseerd op functionaliteit en vaak in aparte bestanden. Deze brokken worden modules genoemd. Modules worden dan aan elkaar gekoppeld om een applicatie te vormen. (**webpack-no-dateB**) (**mozilla-2021A**)

Node.js, een runtime voor computers en servers waardoor Javascript applicaties los van een browser kunnen draaien, ondersteunt modulair programmeren vanaf het begin met CommonJS. Ondersteuning voor modules in de browser daarentegen was niet bestaand. De eerste module bundlers kwamen pas na 2014. Om te begrijpen waarom module bundlers bestaan, moeten we eerst weten hoe webapplicaties voor hen werden gebouwd en welk probleem ze oplosten. (**webpack-no-dateA**)

2.1.1 Script tags

Javascript kan worden gekoppeld aan, of direct worden geschreven in, het HTML-bestand van een site met behulp van script-tags (**mozilla-2021**). Dit wetende, zouden we een verschillende tag en corresponderend Javascript bestand kunnen gebruiken voor elke toepassing van de applicatie. Laten we zeggen dat er een bestand is met alle code voor het aanmelden van een gebruiker en een ander voor algemene gebeurtenissen (op een knop klikken, ...). Dit is prima wanneer we slechts twee bestanden hebben die niet zo groot zijn, maar introduceert knelpunten voor het dataverkeer wanneer dit geschaald zou worden naar een grotere applicatie (**webpack-no-dateB**). Hetzelfde geldt als al de code in één groot bestand zou staan. Het globale domein van de browser zou ook vervuild raken met onze eigen functies. Aangezien sommige, of alle, functies beschikbaar zijn in het globale domein, zou dit veiligheidsrisico's en naam botsingen kunnen introduceren.

```
<!DOCTYPE html>

<html lang="en">
  <body>
    <h1>Hello world!</h1>
    <script src="js/authentication.js"></script>
    <script src="js/main.js"></script>
  </body>
</html>
```

2.1.2 Immediately invoked function expressions

Een Immediately invoked function expression of IIFE is een functie die wordt uitgevoerd zodra hij is gedefinieerd (**mozilla-2021B**). Omdat elke IIFE een lokaal domein declareert,

lost het het probleem van vervuiling van het globale domein op. Het gebruik van IFFE's heeft geleid tot zogenaamde task runners: ze voegen al je project bestanden samen. Het grote nadeel van task runners is dat wanneer één bestand wordt gewijzigd, het hele project opnieuw moet worden opgebouwd. Ook moet je alle afhankelijkheden zoals packages van tevoren handmatig definiëren. Ze maken het makkelijker om functies en hele scripts te hergebruiken, maar doen niets voor de build output te verkleinen. Je kunt nog steeds eindigen met een zeer groot Javascript bestand dat de gebruiker moet downloaden.

```
(function () {  
  let firstVariable;  
  let secondVariable;  
  // ...  
})();  
  
// firstVariable and secondVariable will be discarded after the  
// function is executed.
```

2.1.3 CommonJS

Vóór 2009 draaide Javascript alleen in een browser (**wikipedia-no-date**). Node.js introduceerde een Javascript runtime die op computers en servers kon draaien. Dit bracht een nieuwe reeks uitdagingen met zich mee (**crutchfield-2018**). Aangezien Javascript niet in de browser draaide en er dus geen HTML script-tags waren, hoe konden deze toepassingen nieuwe stukken code laden?

CommonJS introduceerde de require functie (**nodejs-2021**). Hiermee kan alles dat een externe module exporteert, worden geïmporteerd. Herbruikbare code kan nu worden geïmporteerd uit elk ander Javascript bestand in een project. Het maakt het implementeren van dependency management eenvoudig te begrijpen.

Dit alles kwam met een groot addertje onder het gras: Het werkte, en werkt nog steeds, voor Node.js applicaties maar het is geen officiële functie van Javascript en daarom ondersteunen browsers het niet (**crutchfield-2018**). Omdat CommonJS de code niet bundelt, kunnen webbrowsers niet overweg met de verschillende geïmporteerde en geëxporteerde modules. Iets moet dat voor hen doen.

```
//myFunctions.js  
  
const calculateTotal = (a, b) => a + b;  
  
module.exports = calculateTotal;
```

```
//main.js  
  
const calculateTotal = require("myFunctions.js");  
  
console.log(calculateTotal(1, 3)); /*Logs 4*/
```

2.1.4 ECMAScript Modules

CommonJS was geen officiële functie van Javascript. ECMAScript (=Javascript) heeft in versie 6 wel een eigen modulesysteem ingevoerd. ECMAScript Modules of ESM, bereiken dezelfde doelen als CommonJS, maar met een andere syntax. Moderne browsers kunnen nu applicaties draaien die opgebouwd zijn met ES Modules (**mozilla-2021**). Met de nadruk op moderne browsers.

```
//myFunctions.js

export const calculateTotal = (a, b) => a + b;
```

```
//main.js

import { calculateTotal } from "myFunctions.js";

console.log(calculateTotal(1, 3)); /*Logs 4*/
```

2.2 Module bundlers

Ontwikkelaars zijn altijd op zoek naar manieren om hun leven gemakkelijker te maken. Ze willen om het even welk type van module, CommonJS of ESM, of om het even welk ander bestand (een afbeelding bijvoorbeeld) in hun project importeren en het in een kleiner uitvoerbestand naar de eindgebruiker verzenden. En zo werd de module bundler geboren.

De meest essentiële functie van een module bundler is het bijhouden van alle modules die geïmporteerd worden in een project, los van uit hoeveel bestanden het bestaat, en deze te bundelen in een enkel bestand genaamd de bundel. Die bundel wordt ook geminificeerd om zo klein mogelijk te zijn, zonder de functionaliteit aan te tasten. Ze doen dit door commentaar, witruimtes, nieuwe regels, ... te verwijderen. Al dit resulteert in een kleiner bestand.

Ongebundeld bestand: 147 bytes

```
const array = ["Hello", "my", "name", "is", "Maxim"];

//Loop over all elements and print
for (const element of array) {
  console.log(element);
}
```

Gebundeld bestand: 97 bytes

```
const array=["Hello","my","name","is","Maxim"];for(const element of array){console.log(element);}
```

Alle module bundlers van vandaag delen deze functies samen met gemeenschappelijke concepten. We zullen kijken naar de 2 belangrijkste.

Tree shaking

Tree shaking is het proces van het verwijderen van ‘dead code’. Wanneer een module wordt geïmporteerd, is misschien slechts een deel van die module nodig. Misschien wordt slechts één functie van die module gebruikt in het project. Met tree shaking, wordt de rest van die module die niet gebruikt wordt, verwijderd. NPM packages kunnen vaak vele megabytes groot zijn. Tree shaking voorkomt dat al die bytes door de eindgebruiker moet gedownload worden, tenzij elke lijn code uit die packages wordt gebruikt.

Code splitting

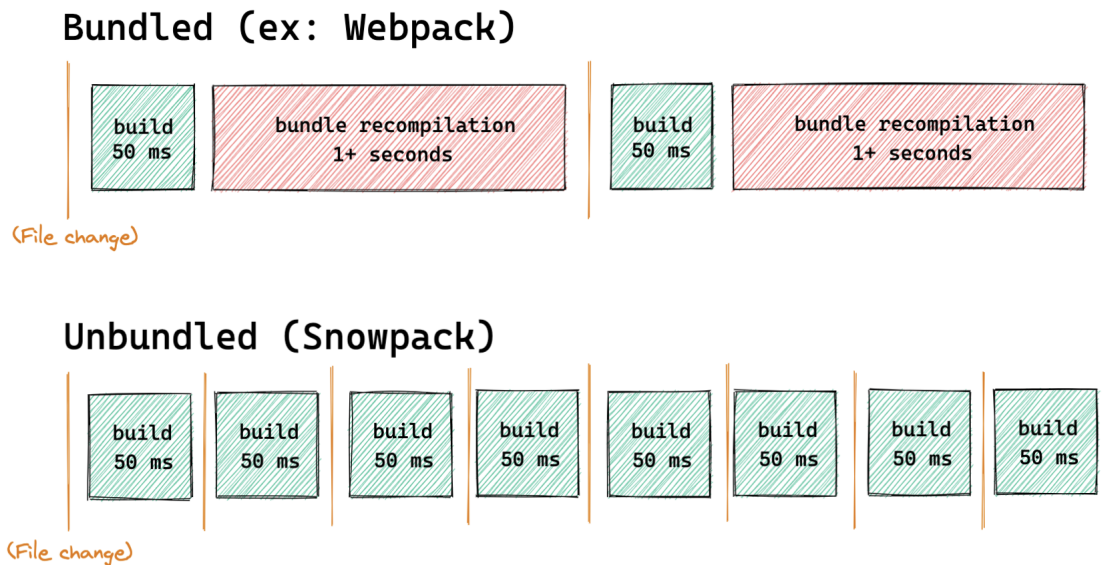
Ook al zorgt Tree shaking en het verwijderen van nieuwe lijnen, commentaar, ... voor aanzienlijk kleinere bundels, toch kan dit bestand te groot worden bij sommige zwaardere applicaties. Code splitting kan worden gebruikt om de uitvoer bundel die wordt aangemaakt op te delen in kleinere bestanden. De gedeeltelijke bundels worden dan parallel geladen of wanneer nodig. Bijvoorbeeld: neem een website met meerdere pagina's. Als de code niet wordt gesplitst, zal alle code van alle pagina's in een enkel bestand staan en door de gebruiker worden gedownload wanneer de site wordt opgevraagd. In veel gevallen, zoals bij een klein project, is dit prima. Dat is waarom het optioneel is. Voor grotere toepassingen echter, is het slimmer om voor elke pagina een aparte bundel aan te maken. Die bundel bevat dan enkel de code nodig voor de pagina in kwestie, niet van heel de applicatie. Bij het laden van elke pagina, wordt de aparte bundel gedownload.

2.2.1 Ongebundelde build tools

De meest recente ontwikkeling in het bouwen van een webapplicatie is de ongebundelde build tool. Ze vervullen dezelfde rol als de module bundler maar dan met een andere werkwijze. Omdat de meeste recente browsers nu ESModules ondersteunen, is het bundelen van alle Javascript modules van een project vaak niet nodig. Als een module bundler zijn development server opstart, moeten alle bestanden van het project worden gebouwd en vervolgens gebundeld. Als een bestand wordt gewijzigd, zal dat hele proces weer doorlopen worden. Bij ongebundelde build tools wordt een bestand alleen gebouwd als het wordt opgevraagd, wat een zeer snelle opstarttijd van de server betekent. Wanneer een bestand is gebouwd, wordt het voor onbepaalde tijd gecached. De browser zal een bestand nooit twee keer hoeven te downloaden totdat het verandert. Wanneer dat toch gebeurt, hoeft alleen dat ene bestand opnieuw te worden opgebouwd.

Dit alles resulteert in zeer snelle prestaties in vergelijking met module bundlers zoals Webpack. Wat echter niet betekent dat module bundlers voorbijgestreefd zijn. Sterker nog: ongebundelde build tools gebruiken ook module bundlers, de ene al meer dan de ander. Dit omdat de voordelen van Tree-shaking, een enkel uitvoerbestand en andere pluspunten van module bundlers nog steeds gelden. In hoofdstuk 3 en 4 zal hier dieper op ingegaan worden.

Voor de rest van deze proef zal de verzamelnaam “build tools” gebruikt worden om naar module bundlers en ongebundelde build tools te verwijzen. Module bundlers zijn namelijk



Figuur 2.1: Gebundeld vs ongebundeld (**snowpack-no-date**)

ook tools die een webapplicatie helpen bouwen of builden, maar dan gebundeld. Waar mogelijk zal natuurlijk de meest specifieke term gebruikt worden.

2.2.2 Keuze build tools

In deze paragraaf zullen voorbeelden van build tools worden besproken. De keuze welke te bespreken gebeurt aan de hand van een requirement analyse. Die is vrij beknopt aangezien de requirements voor zowel module bundlers als ongebundelde build tools dezelfde is met uiteraard het verschil dat ze tot de bepaalde categorie behoren.

- Functionele requirements
 - Een Javascript webapplicatie kunnen bouwen
- Niet-functionele requirements
 - Typescript ondersteunen
 - SASS ondersteunen
 - Statische bestanden ondersteunen
 - ...

Hieruit volgt de long list voor module bundlers:

- Webpack
- Parcel
- Rollup
- Fusebox
- Browserify

- Gulp
- ...

Module bundlers werken fundamenteel op dezelfde manier. Ze allemaal vergelijken is dus overbodig. We kiezen voor Webpack en Parcel, dit is dus de short list. Zij verschillen het meest van elkaar zoals verder in dit hoofdstuk nog besproken zal worden en zijn daarenboven twee van de meest populaire opties (**stateofjs-2020**). Rollup is ook een goede kandidaat maar komt al aan bod bij een ongebundelde build tool.

De zoektocht naar ongebundelde build tools voor deze proef was een pak gemakkelijker. Er bestaan namelijk maar twee met een deftige user-base. Volgende opsomming fungeert dus zowel als long- en short list.

- Snowpack
- Vite

Webpack

Webpack is de meest populaire module bundler in de wereld (**stateofjs-2020**). Dit heeft veel te maken met het feit dat het één van de oudste is. Het wordt voorgeïnstalleerd in veel web frameworks zoals create-react-app (**facebook-2021**) en Next.js (**vercel-no-date**). Op deze manier wordt het door velen gebruikt zonder dat ze het weten. Webpack bevat een optionele ingebouwde ontwikkelserver die het opzetten van een lokale ontwikkelomgeving eenvoudiger maakt.

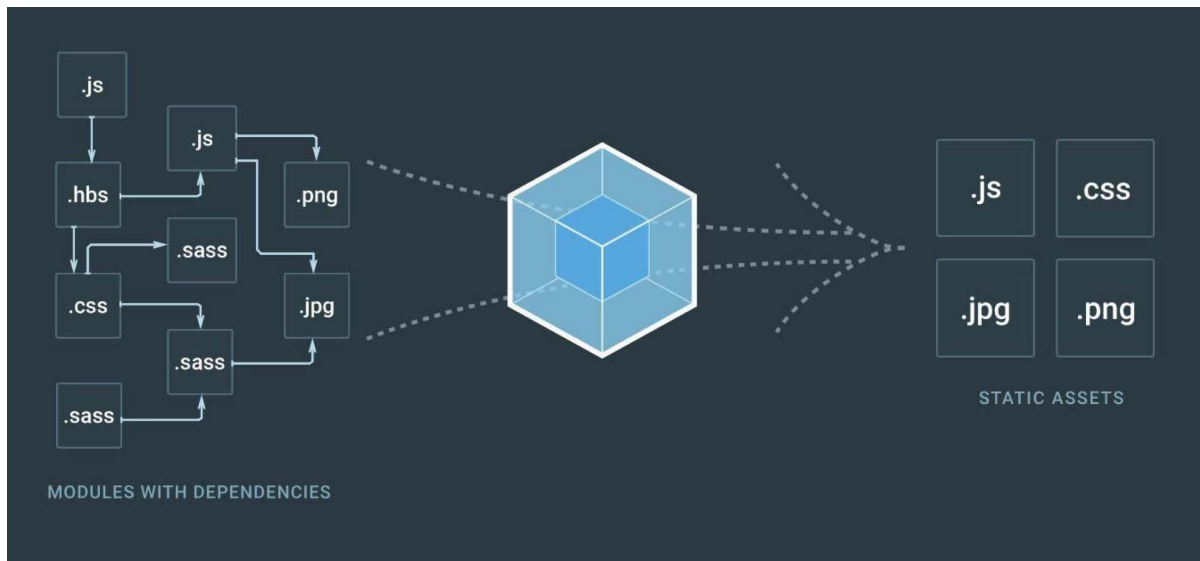
Webpack draait op Node.js. Het kan zijn werk doen zonder enige configuratie, maar is zeer configureerbaar indien nodig. Het ondersteunt de hierboven besproken module types en meer. Omdat Webpack standaard alleen Javascript en JSON bestanden begrijpt, worden Loaders gebruikt om de verwerking van andere bestandstypen mogelijk te maken door ze om te zetten in geldige modules. Met behulp van Loaders kunnen andere typen modules of zelfs assets zoals afbeeldingen worden geïmporteerd en verwerkt door Webpack. Bovendien kan Webpack ook worden uitgebreid met plugins. Deze maken een brede waaier aan extra functionaliteit mogelijk, zoals bundel optimalisatie.

De functie van een module bundler is al besproken. Maar hoe bereikt Webpack dit? Wanneer een bestand afhankelijk is van een ander in een project, ziet Webpack dat en zet in zijn dependency graph.

Deze graaf wordt recursief opgebouwd. Wanneer het tijd is om de applicatie te bouwen, gebruikt het de dependency graph om alle bestanden samen te voegen tot één uitvoerbestand dat dan naar de browser wordt verzonden. Module bundlers en dus Webpack ook doen dit zowel in ontwikkeling als in productie.

Parcel

Parcel is een andere module bundler. Het draait echter niet op Node.js, in plaats daarvan is de compiler die het gebruikt gebouwd met Rust. Rust is een gecompileerde program-



Figuur 2.2: Essentiële functie module bundler (**webpack-no-date**)

meertaal. Zonder al te veel in detail te treden, betekent dit dat Rust code direct naar machinecode wordt gecompileerd, wat resulteert in snellere prestaties. Het doet veel van dezelfde dingen die Webpack doet zonder enige configuratie. Toen het werd uitgebracht, was het belangrijkste kenmerk dat het geen configuratiebestand nodig had en Webpack wel. Nu kan Webpack echter ook zijn werk doen zonder een configuratiebestand. Hoewel een no-configuratie aanpak geweldig is voor kleinere projecten, is het vaak niet haalbaar voor een grote applicatie.

Snowpack

Snowpack is het eerste voorbeeld van een ongebundelde build tool. Het is momenteel de meest populaire in zijn categorie (**stateofjs-2020**). Het pakt uit met de slogan ‘The faster build tool’. Of dat effectief waar is, zal in het volgende hoofdstuk besproken worden. Het grootste pluspunt van Parcel was dat die zijn doel probeert te bereiken met zo min mogelijk configuratie. Deze filosofie volgt Snowpack niet. Een configuratie bestand is vereist en kan al snel aanzienlijke grootte bereiken.

Zoals hierboven vermeldt, gebruikt een build tool vaak toch een module bundler in productie. Bij Snowpack is dit echter optioneel. Standaard is de productie build die Snowpack aanmaakt dus ook ongebundeld, iets waar enkel moderne browsers mee overweg kunnen. De configuratie kan wel aangepast worden om gebruik te maken van Webpack of Rollup, nog een andere module bundler, zodat dat laatste geen probleem meer vormt. Voor dit onderzoek voegen we zo’n module bundler niet toe aan Snowpack. Bij de volgende build tool die aan bod komt, Vite, is het geen optie om de module bundler in productie weg te laten. Snowpack zal dus de volledige ongebundelde applicatie representeren, terwijl Vite hetzelfde zal doen voor half-ongebundelde.

Vite

Vite is de jongste build tool die in deze studie aan bod komt. Het combineert een ongebundelde development omgeving met een gebundelde in productie. Voor dit laatste gebruikt het Rollup (**vite-no-date**), een alternatief van Webpack. Hoewel het ook een configuratie bestand bevat, is die veel minder groot en ingewikkeld dan Snowpack.

Zoals de naam Vite doet vermoeden, beweren de makers ervan bliksemsnelle opstart- en buildtijden in vergelijking met de competitie.

In de twee volgende hoofdstukken zullen de besproken build tools getest en vergeleken worden. Eerst bij het opzetten van een nieuw project en daarna bij het omvormen van een bestaand. Elk hoofdstuk bevat een eigen conclusie, als laatste volgt de algemene.

De traditionele methodologie is in deze proef opgedeeld in twee hoofdstukken. In de stand van zaken werden de verschillende te vergelijken build tools toegelicht. Deze zullen in de twee hoofdstukken naast elkaar gelegd en gequoteerd worden. Eerst zal gekeken worden hoe ze verschillen bij het opzetten van een nieuw project. Daarna trachten we een bestaand project dat met Webpack opgezet is, om te vormen naar een van de andere opties. Elk hoofdstuk wordt afgesloten met een eigen besluit. Tot slot volgt in het laatste hoofdstuk een algemene, samenvattende conclusie.

Build tools hebben een enkel doel: een werkende webapplicatie afleveren. Hoe snel ze dat kunnen is de belangrijkste vergelijkende factor. Hieronder staan de twee factoren waar het meest rekening mee gehouden zal worden. Daarnaast zullen meer subjectieve factoren, zoals de gemakkelijkerheid om ze werkende te krijgen, aan bod komen.

- Snelheid creatie uitvoer voor productie
- Opstartsnelheid development server

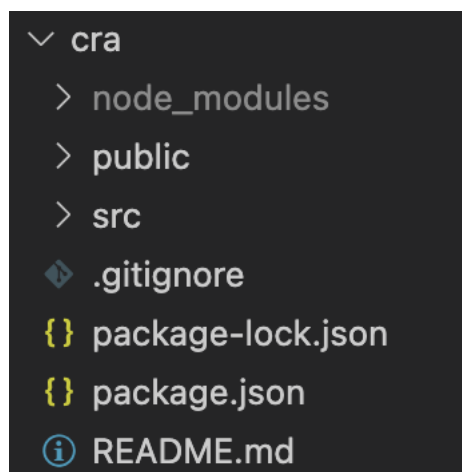
Alle metingen werden gedaan op een M1 MacBook Pro en zijn reproduceerbaar aan de hand van deze repository vansteenkiste-2021A. De snelheidsmetingen werden gedaan aan de hand van een schermopname waardoor achteraf de snelheid accuraat kon waargenomen worden zonder ruimte voor menselijke fouten. Alle metingen werden driemaal uitgevoerd, met het verwijderen van cache bestanden waar mogelijk.

3.1 Webpack

Een nieuw project opzetten met Webpack kan op verschillende manieren: zelf een project opzetten en de configuratie volledig manueel schrijven of gebruik maken van een framework waarin het al geconfigureerd voor ons is. Aangezien niet velen het eerste pad bewandelen, zullen we gebruik maken van de meest populaire manier om een React project op te zetten. Create-react-app of CRA is een minimale framework gemaakt door de makers van React zelf om gemakkelijk een React omgeving op te zetten. Het is één van de vele frameworks die Webpack als module bundler gebruikt. Om te beginnen, voeren we volgende commando's uit.

```
npx create-react-app my-app
```

Bovenstaande code maakt een project aan met create-react-app. Daarna ziet ons project er als volgt uit. Merk op dat er geen config bestand voor Webpack is.



Figuur 3.1: Bestandsstructuur nieuw CRA project

Alles wat in de public map staat, zijn statische assets. Die zullen dus via een url bereikbaar zijn. Als we kijken in het index.html bestand, merken we op dat er geen enkele script-tag aanwezig is. Hierover later meer. In de src map staat alles wat door Webpack zal gebundeld worden. Standaard worden er CSS bestanden aangemaakt voor styling en een logo in svg formaat. Dit wordt gedaan om aan te tonen dat we deze assets gewoon in de Javascript code kunnen importeren, de bundler kan hiermee overweg.

```
import logo from "./logo.svg";  
import "./App.css";  
  
// ...
```

In het package.json bestand staat er allemaal info over dit project. In het dependencies gedeelte staan alle externe packages die dit project gebruikt. Nieuwe packages worden gedownload aan de hand van Node Package Manager of NPM. Bij de dependencies staat een package genaamd “react-scripts”. React-scripts (**facebook-2018**) is een package gemaakt door de makers van React en is de motor achter CRA. Wat voor dit onderzoek

relevant is, is dat het de Webpack.config bevat. Dit bestand bestaat uit maar liefst 700+ lijnen code (**facebook-2021**). Nu is de reden dat we CRA gebruiken en niet van nul beginnen, duidelijk.

De volgende stap is om het project lokaal op te starten. React-scripts gebruikt de ingebouwde development server van Webpack. Na het uitvoeren van volgend commando, wordt die server opgestart en opent de webapplicatie in een browser.

```
npm start
```

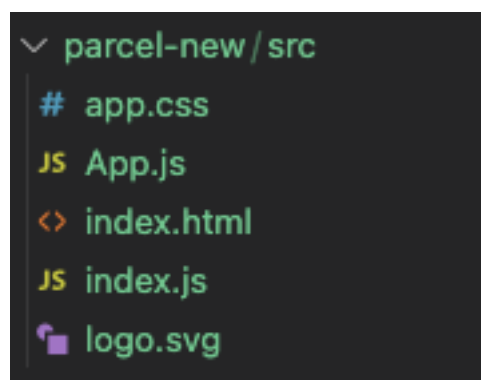
Grootte project (MB)	0,037
Grootte node_modules (MB)	214,1
Grootte uitvoer (MB)	0,514
Snelheid creatie uitvoer (s)	3,67
	5
	3
	3
Snelheid opstarten development server (s)	2,67
	4
	2
	2

Tabel 3.1: Overzicht nieuw project met Webpack

3.2 Parcel

Bij Webpack gebruikten we een framework om het vele configuratie werk te omzeilen. Bij Parcel is dit niet nodig. Zoals in de literatuurstudie vermeld werkt Parcel op een gelijkaardige manier als Webpack, maar dan met zo min mogelijk configuratie. Om een nieuw project op te zetten, gaan we dus geen framework gebruiken.

Maak een nieuwe map aan waar het project zal leven. Aangezien we van nul beginnen, moeten de bestanden uit figuur 3.2 zelf aangemaakt worden.



Figuur 3.2: Aan te maken bestanden voor nieuw Parcel project

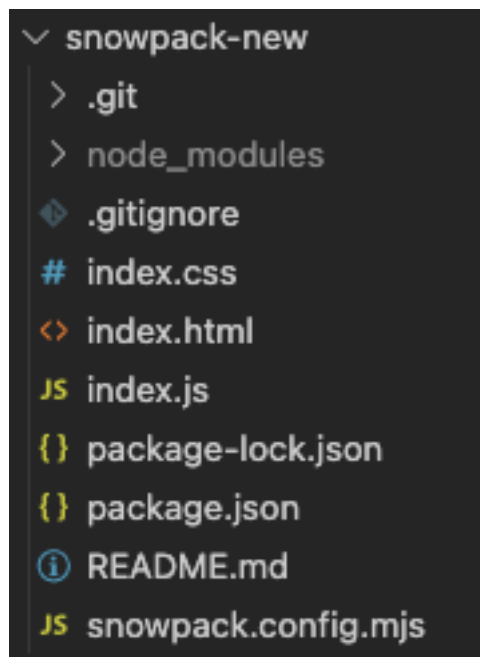
Hierna moeten er nog enkele packages geïnstalleerd worden, namelijk: React, React-DOM en natuurlijk Parcel. In de package.json moet er ook nog meegegeven worden waar de index.html zich bevindt. Merk op dat er geen apart configuratiebestand voor Parcel is. Nu kan het project opgestart worden met hetzelfde commando als bij Webpack.

Merk op dat er geen public folder aanwezig is zoals in CRA. Er is momenteel geen folder waar statische bestanden kunnen leven. Als dit een vereiste is, heeft Parcel een plugin nodig. Gelukkig zijn die gemakkelijk te installeren.

3.3 Snowpack

Voor Snowpack gaan we net zoals bij Parcel te werk zonder framework. In tegenstelling tot Parcel heeft het Snowpack team al een speciaal react-template gemaakt met een kant en klaar commando om het te initialiseren. Zelf de bestanden aanmaken en dependencies toevoegen is dus niet nodig.

```
npx create-snowpack-app react-snowpack --template @snowpack/app-template-minimal
```



Figuur 3.3: Aangemaakte bestanden door commado

Deze structuur is identiek aan die van CRA. In tegenstelling tot Parcel is een public folder al geconfigureerd waar statische bestanden, zoals afbeeldingen, beschikbaar staan. Een configuratiebestand voor Snowpack is ook aangemaakt. Hierin staan al 25 lijnen configuratie voor ons geschreven. Meer dan Parcel maar aanzienlijk minder dan CRA. Merk op dat sommige bestanden nu de extensie .jsx in plaats van .js hebben. Dit komt doordat Snowpack geen JSX tolereert in .js bestanden. JSX is wat een React component retourneert i.e elk React component moet een .jsx extensie hebben. Geen probleem bij

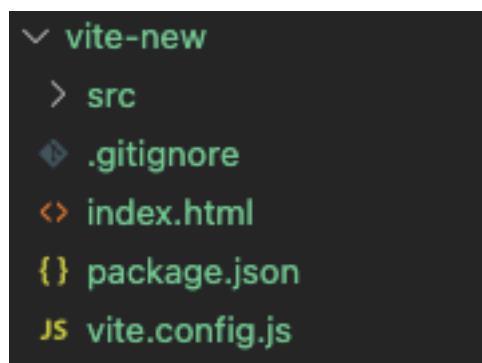
een nieuw project maar bij een oud kan het nodig zijn om vele bestanden van extensie te veranderen, zie later.

Zoals in de literatuurstudie vermeld, is Snowpack geen module bundler aangezien het de verschillende bestanden in een project niet bundelt. In productie kan dat optioneel nog gedaan worden door Webpack of Rollup maar dat is niet standaard. In development heeft dit het grote voordeel dat de bundel niet telkens opnieuw opgebouwd moet worden als een bestand veranderd.

3.4 Vite

Vite is nog een voorbeeld van een unbundled build tool, net zoals Snowpack. Een nieuw project opzetten is heel gemakkelijk aan de hand van een simpel commando dat ze voorzien hebben, net zoals Snowpack.

```
npm init vite@latest vite-new --template react
```



Figuur 3.4: Aangemaakte bestanden door commado

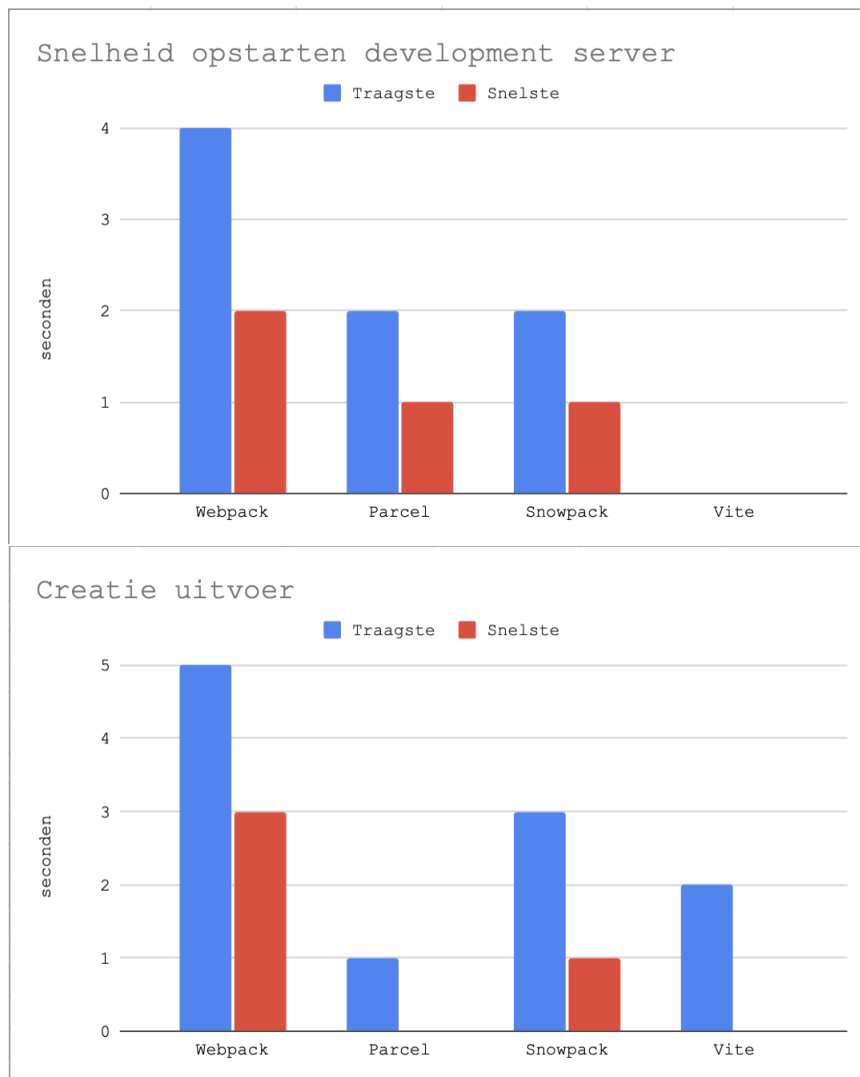
Er is een klein config bestand aanwezig van 7 lijnen code waar de react plugin is geïnitieerd. Voor de rest ziet het project in grote lijnen er uit als dat van Snowpack. Er is geen public map aanwezig maar dat kan aangemaakt worden en wordt automatisch geconfigureerd.

3.5 Conclusie

Al de build tools hebben niet gezweet bij het opzetten van een nieuw project, wat maar normaal is. Voor Webpack is er gebruik gemaakt van een framework, CRA, omdat het configuratie werk anders te veel zou zijn. Ook al is het een nieuw project en dus relatief klein, toch kunnen we al verschillen waarnemen tussen de vier kandidaten.

Op onderstaande figuur zien we al een trend verschijnen die doorheen deze conclusie zal gelden: Webpack is aanzienlijk trager dan de competitie, ondanks dat Snowpack niet bundelt, is het toch even traag of trager dan Parcel. Die laatste zijn Rust compiler, zie

literatuurstudie, zal zijn vruchten afwerpen. De drie balkjes die ontbreken bij Vite en het ene bij Parcel zijn geen fout: ze waren gewoon sneller dan een volledige seconde.



Figuur 3.5: Resultaten nieuw project

De developer experience was vooral de build tools zo goed als hetzelfde bij het opzetten van dit nieuw project. De conclusie kunnen we daarop dus niet baseren. Maar de data hierboven is duidelijk genoeg: bij het opzetten van een project is Vite de beste optie. Het gebruikt het beste van beide werelden: ongebundelde code in development en gebundelde code in productie. De andere opties hebben niet direct een afknapper, maar de snelheid van Vite valt niet te negeren.

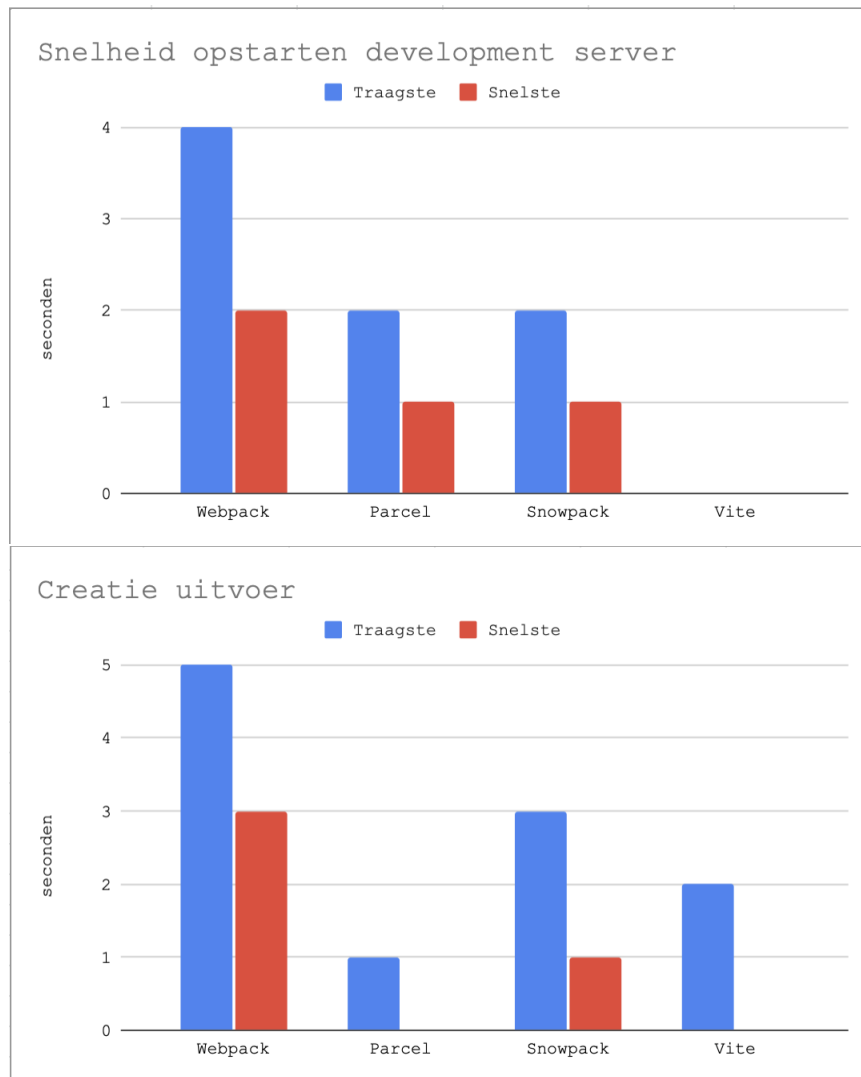
In het vorig hoofdstuk werden de gekozen module bundlers op de proef gesteld aan de hand van drie open-source projecten en een van nul te beginnen. De verschillende stappen om ze werkende te krijgen werden overlopen, bij de een waren dat er al meer dan de ander. In dit hoofdstuk wordt gekeken naar de gemeten en waargenomen resultaten, objectief en subjectief, om tot een conclusie te komen.

4.1 Nieuw project

Al de module bundlers hebben niet gezweet bij het opzetten van een nieuw project, wat maar normaal is. Voor Webpack is er gebruik gemaakt van een framework, CRA, omdat het configuratie werk anders te veel zou zijn. Ookal is het een nieuw project en dus relatief klein, toch kunnen we al verschillen waarnemen tussen de vier kandidaten.

Op onderstaande figuur zien we al een trend verschijnen die doorheen deze conclusie zal gelden: Webpack is aanzienlijk trager dan de competitie, ondanks dat Snowpack niet bundelt, is het toch even traag of trager dan Parcel. Die laatste zijn Rust compiler, zie literatuurstudie, zal zijn vruchten afwerpen. De drie balkjes die ontbreken bij Vite en het ene bij Parcel zijn geen fout: ze waren gewoon sneller dan een volledige seconde.

De developer experience was voor al de build tools zo goed als hetzelfde bij het opzetten van dit nieuw project. De conclusie kunnen we daarop dus niet baseren. Maar de data hierboven is duidelijk genoeg: bij het opzetten van een project is Vite de beste optie. Het gebruikt het beste van beide werelden: ongebundelde code in development en gebundelde code in productie. De andere opties hebben niet direct een afknapper, maar de snelheid van Vite valt niet te negeren.

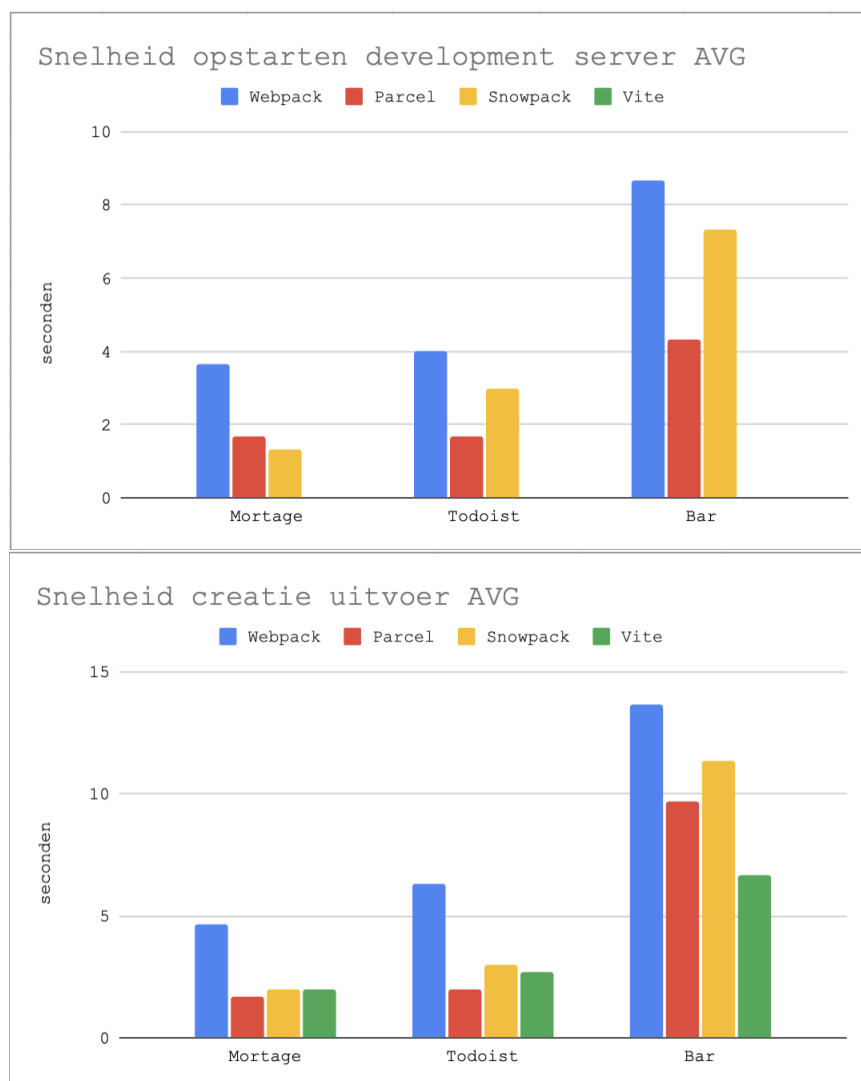


Figuur 4.1: Resultaten nieuw project

4.2 Bestaand project

Een bestaand project omvormen is het heel andere koek gepaard met andere overwegingen. In de methodologie is getracht projecten te kiezen met zoveel mogelijk uiteenlopende technologieën. Natuurlijk is lang niet alles aan bod gekomen. Een project kan bestaan uit talloze combinaties van verschillende, soms niche technologieën, packages of zelf geschreven bibliotheken. Wat in dit onderzoek geconcludeerd zal worden, is een overweging gebaseerd op de projecten die hier aan bod gekomen zijn. Het kan zijn dat voor een ander project, die keuze minder gemakkelijk of zelfs uitgesloten is door welke factor dan ook. Dat terzijde, de conclusie zal in het algemeen voor de meesten gelden aangezien veel voorkomende use-cases aan bod kwamen. Onderstaande resultaten zouden hoogstwaarschijnlijk nog kunnen bijgeschaafd worden door van elke build tool de configuratie te optimaliseren per project. Dit echter is niet het punt van deze studie. Welke build tool is in het algemeen de beste keuze, zonder uren de configuratie bij te schaven?

Op onderstaande figuur is te zien dat elke geteste build tool het beter doet dan Webpack en het verschil is vaak niet klein. Hoewel Snowpack ook aanzienlijk sneller maar aangezien uit de methodologie is gebleken dat de developer experience ver van optimaal is, zeker in vergelijking met de andere opties, kijk je beter elders bij het kiezen van een nieuwe build tool. Parcel en Vite daarentegen doen het heel goed op vlak van developer experience en snelheid. Parcel levert gebundelde code af, ook in development. Dit kan een voor- of nadeel zijn, afhankelijk hoe het bekeken wordt. Uit het vorig hoofdstuk blijkt dat Vite geen CommonJS ondersteunt, Parcel wel. Hoewel Parcel in sommige scenario's net iets sneller weet te zijn dan Vite, wint die laatste toch in de meeste gevallen. Merk op dat ook bij de tweede grafiek, het balkje bij Vite ontbreekt. Dit is weer geen fout: Vite slaagt erin om zijn development server gemiddeld in sneller dan een seconde op te starten. Dat is indrukwekkend, zeker als dit naast de resultaten van de competitie gelegd wordt.



Figuur 4.2: Resultaten bestaand project

De opzet van deze proef, of Webpack nog steeds mag gezien worden als de koning der module bundlers, kan dus beantwoord worden met een duidelijke 'nee'. Zelf binnen zijn eigen categorie van gebundelde code, blijkt Parcel, in de meeste gevallen, een betere optie

te zijn.

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

A.1 Introduction

A module bundler is one of the most essential technologies in modern web development. Nobody makes web applications with plain HTML, CSS and JS anymore. All the fancy new tools, frameworks and libraries for a web developer to use, like Angular or React, won't work in the browser without being bundled. So the choice of which bundler to use is very important. While Webpack is the most popular^{1,2,3}, there are many others that claim to do the job better. How does Webpack compare to its competitors? Are their claims valid? If so, why is Webpack still the most popular? Thus to summarize: does it make sense for development teams at companies, stand-alone developers or just hobby coders to bundle their code with the same tool as 5 years ago?

Many developers give much thought about which framework to use, which back-end and so forth. The module bundler however often doesn't get that much thought. This is because it's less attractive and in many frameworks comes pre-installed. But as your web app can't run without it, it's one of the most important choices you make. The goal of this paper is to demystify the module bundler and compare the most popular options. Not just in terms of speed and module size, but also its plugins, developer experience,

A.2 State-of-the-art

Much can already be found about module bundlers online. On the website or GitHub repository of the bundler itself, articles written by third-parties and video's on YouTube. While they contain very useful information, they lack an in-depth analysis of the differences between the major players and what that means for the developer and the output product. This is an important goal of the paper.

A.3 Methodology

Firstly, the most popular bundlers will be campered theoretically. How they work and how they may differ. To compare them in practice will require code-bases of many sizes. Luckily there are many open-source codebases on GitHub to chose from. Why many sizes, you ask? Because it could be interesting to see how module bundlers deal with smaller projects compared to larger ones. To see if the advantages of one fade when the project size in- or decreases. It could also be interesting to compare them based on the framework, like Angular or Create-React-App. Those frameworks ship with pre-configured bundlers so it remains to be seen if it's even possible to replace those with others. It's something that has to tested.

Secondly and maybe more important: what about already existing, operational projects? For example: a company that already has a site or some kind of application on the web bundled with WebPack. Is it possible to just switch to another module bundler? What hurdles have to be overcome to do so and what are the benefits?

Testing many code-bases and observing the differences in numerous categories is necessary. One important category is the developer experience: how easy is it to install? How simple or complex are the config file and plugins. There are a lot of things to consider.

A.4 Anticipated results

Many module bundlers claim speedier builds and smaller bundle sizes. So that's an easy prediction to make. Wether that remains true with projects of any size remains to be seen. Will those improvements be great enough to warrant the competitor to be used over Webpack? Developer experience and performance in development mode are defining factors as well. Obviously the former will be quite subjective so that's an important factor to note.

A.5 Anticipated conclusions

It's hard to predict what the conclusion will be. That's the most important reason why I want to conduct this research. Webpack has served us well but as the tech world shifts

more and more towards applications written with web technologies, the time of newer approaches may have come.

A.6 References

1. Search results module bundlers. (n.d.). Google Trends. Retrieved October 2021, from <https://trends.google.com>
2. Nalakath, N. (2021, February 2). Module Bundlers and their role in web development. | Better Programming. Medium. Retrieved October 2021, from <https://betterprogramming.pub/javascript-module-bundlers-2a1e9307d057>
3. The State of JavaScript 2018: Other Tools. (2018). StateOfJs. Retrieved October 2021, from <https://2018.stateofjs.com/other-tools/>