HO
GENT

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 basistechnologieen 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**). 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 willen installeren en gebruiken, gedownload van een packages 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. Dit wetende, zouden wij 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. 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 IFFEs 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. 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 in Javascript. 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. 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. 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 maar er zijn nog andere, of om het even welk ander bestand (een afbeelding bijvoorbeeld) in hun project importeren en het in 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

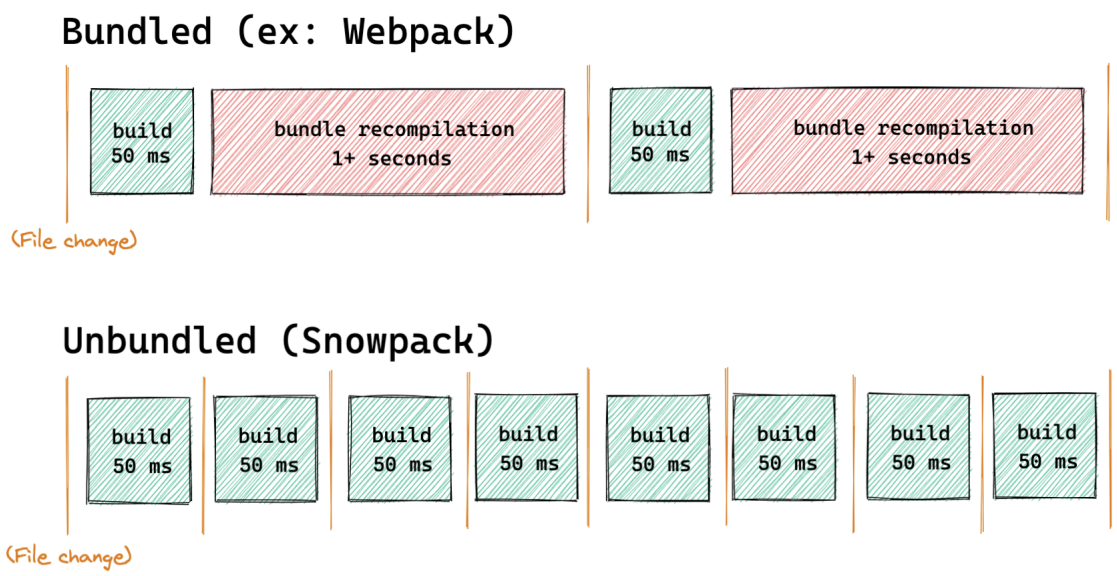
Ookal zorgt Tree shaking en het verwijderen van nieuwe lijnen, commentaar, ... voor aanzienlijk kleinere bundels, toch kan dit bestand te groot worden bij sommige zwaardere applicaties. Code splitting kan worden gebruikt om de uitvoer bundel die wordt aangemaakt op te delen in kleinere bestanden. De 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 questie, niet van heel de applicatie. Bij het laden van elke pagina, wordt de aparte bundel gedownload.

Het vervolg van dit hoofdstuk is opgedeeld in twee delen. De module bundlers die gaan besproken worden kunnen namelijk opgedeeld worden in twee categorieën. De module bundlers die besproken worden zijn gekozen aan de hand van populariteit (**stateofjs-2020**) binnen hun categorie. Voor elke categorie is het mogelijk om veel meer voorbeelden te vinden maar aangezien ze gebaseerd zijn op dezelfde principes, is het zinloos voor deze studie om die ook te vergelijken. Webpack en de categorie tot wie het behoort wordt vergeleken met een meer moderne manier van aanpak en voorbeelden daarvan.

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



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

Webpack. Dit alles betekent echter niet dat module bundlers voorbijgestreefd zijn. Sterker nog: ongebundelde build tools gebruiken ook module bundlers, de ene al meer dan de ander. Dit wordt gedaan omdat de voordelen van Tree-shaking, een enkel uitvoerbestand en andere pluspunten van module bundlers nog steeds gelden. In hoofdstuk 3 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 ook tools die een webapplicatie helpen 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. Eerst twee module bundlers, daarna twee ongebundelde build tools. Vooral in de categorie van module bundlers zijn er veel meer voorbeelden te vinden (Rollup, Fusebox, Browserify, ...), hoewel ze allemaal hetzelfde resultaat willen bekomen: een gebundelde webapplicatie aan een browser kunnen leveren. Aangezien de verschillen tussen de meeste helemaal niet groot zijn, is het vrij nutteloos een hele requirement analyse op te bouwen. De twee keuzes voor de module bundler categorie werden gemaakt aan de hand van populariteit en hoe veel ze van elkaar verschillen. De zoektocht naar ongebundelde build tools voor deze proef was een pak gemakkelijker. Er zijn er namelijk maar twee met een deftige user-base.

Hoewel de modernere aanpak van ongebundelde build tools in opmars is, zijn module bundlers zeker nog niet voorbijgestreefd. De verschillen en voorbeelden zullen worden

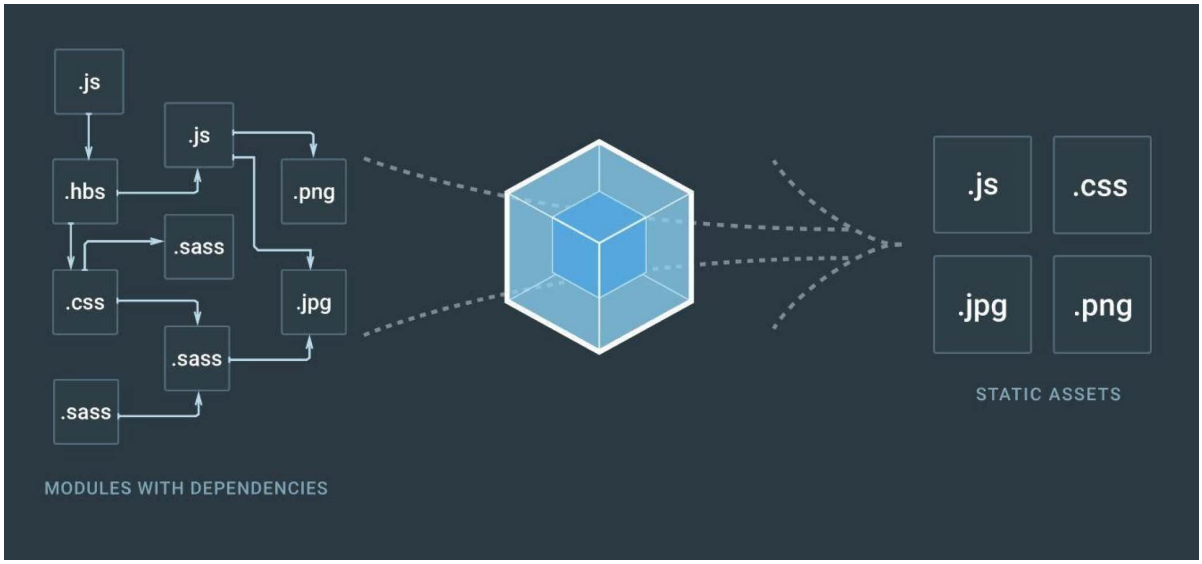
uitgelegd in de volgende sectie.

Webpack

Webpack is de meest populaire module bundler in de wereld (). Dit heeft veel te maken met het feit dat het één van de oudste is. Het wordt voorgeïnstalleerd in veel zoals create-react-app en Next.js (). Op deze manier wordt het door velen gebruikt zonder dat ze het weten. Het wordt geleverd met 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.



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

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 programmeertaal. Zonder al te veel in detail te treden, betekent dit dat Rust code direct naar machine code 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. Het pakt uit met de slogan 'The faster build tool'. Of dat effectief waar is, zal in het volgend hoofdstuk besproken worden. Het grootste pluspunt van Parcel was dat die zijn doel probeert te bereiken met zo min mogelijk configuratie. Deze filosophie 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.

In de stand van zaken werden de verschillende te vergelijken module bundlers toegelicht. Deze zullen in dit hoofdstuk naast elkaar gelegd en gequoteerd worden aan de hand van verscheidene factoren. 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. In het volgend hoofdstuk volgt de conclusie.

Eerst en vooral is er nood aan bestaande projecten om de build tools te kunnen testen. Gelukkig zijn er online genoeg open-source voorbeelden hiervan te vinden. Drie projecten werden gekozen aan de hand van hun grootte en technologieën die ze gebruiken. Eén eigenschap hebben ze allemaal gemeen: het zijn Javascript projecten die React als UI library gebruiken. De reden dat geen projecten gekozen zijn die andere UI libraries zoals Vue gebruiken, is omdat React veel meer gebruikt wordt, ook door de co-promotor. Desondanks zijn de gekozen projecten ook representatief voor de andere UI libraries omdat ze uiteindelijk allemaal Javascript zijn.

Volgende factoren worden gebruikt om een eenduidige vergelijking te maken. Daarnaast zullen meer subjectieve factoren, gelijk de gemakkelijkheid om het werkende te krijgen, aan bod komen.

- Output bundle grootte
- Output bundle snelheid
- Snelheid bij wijziging
- Opstartsnelheid development server

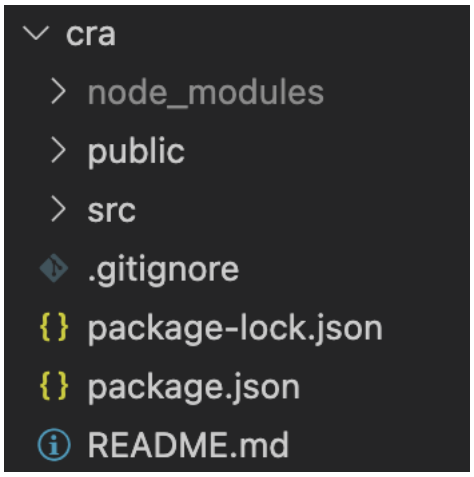
3.1 Nieuw project

3.1.1 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

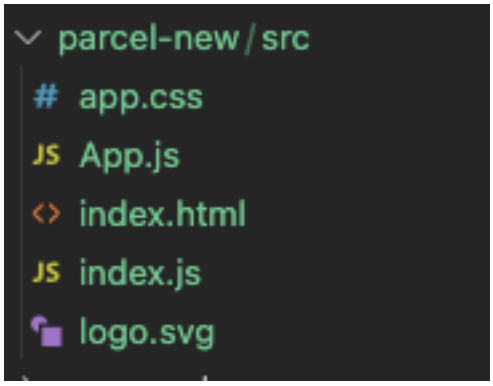
3.1.2 Parcel

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.

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.

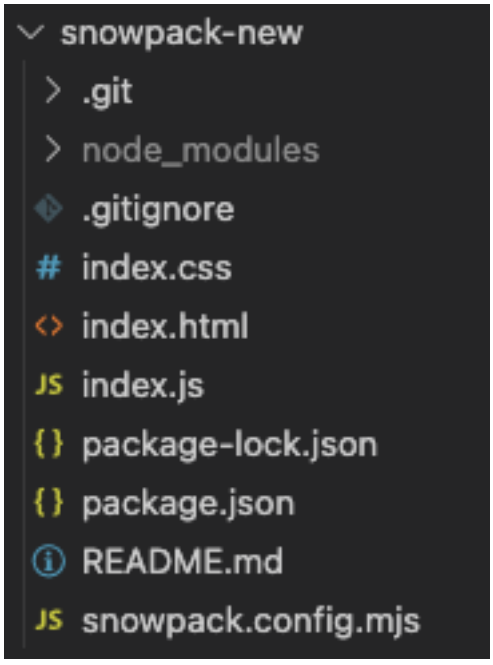


Figuur 3.2: Aan te maken bestanden voor nieuw Parcel project

3.1.3 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.

3.1.4 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 geïnitialiseerd. 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.

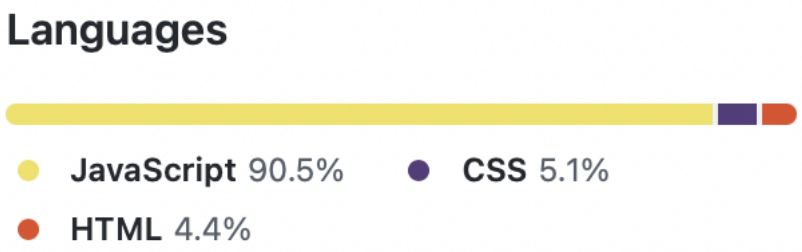
3.2 Bestaand project

In het vorige deel, werden nieuwe projecten opgezet voor de respectievelijke module bundlers. In dit deel worden bestaande projecten, opgezet met Webpack, omgevormd zodat ze gebruik maken van de andere opties. Per project wordt eerst wat toelichting gegeven en vervolgens per module bundler wat er nodig is om ze werkende te krijgen.

3.2.1 Mortage

Het eerste project dat we gaan proberen omvormen is de Mortage Overpayment Calculator (**houghton-2019**). Het is een zeer simpel en klein project van 17 KB groot dat is opgezet met CRA. Daarnaast maakt het nog gebruik van andere open-source packages die verzameld worden in de nodemodules map. Die map is 214 MB groot.

Mortage is een ideaal project om mee te starten aangezien het geen speciale technologieën gebruikt. CSS voor stijl, wat de browser begrijpt zonder enige omvorming en voor de rest Javascript en HTML. Normaal zouden we dus niet in de problemen mogen komen. Een potentiële moeilijkheid wat dit project ook vermijdt is statische bestanden.



Figuur 3.5: Overzicht gebruikte technologieën van het Mortage project

Parcel

Zoals in de literatuurstudie vermeld, probeert Parcel hetzelfde als Webpack te bereiken maar met veel minder tot geen configuratie. Die bewering zal nu nagegaan worden. Mortage is opgezet met CRA en gebruikt dus het react-scripts package die onder andere alle Webpack configuratie op zich neemt. Om Webpack in te ruilen voor Parcel moeten we dus eerst react-scripts bij het grofvuil zetten. Het package.json bestand bevat alle info over een project: de naam, versie, welke andere packages het gebruikt, hoe het wordt opgestart en nog veel meer. Ook Mortage heeft zo'n bestand en dat ziet er als volgt uit:

```
{
  "name": "mortgage",
  "version": "0.1.0",
  "dependencies": {
    "d3-axis": "^1.0.12",
    "d3-scale": "^2.2.2",
    "d3-selection": "^1.4.0",
    "d3-shape": "^1.3.4",
    "d3-transition": "^1.2.0",
    "react": "^16.8.1",
    "react-dom": "^16.8.1",
    "react-scripts": "2.1.3"
  },
  "scripts": {
    "start": "react-scripts␣start",
    "build": "react-scripts␣build",
    "test": "react-scripts␣test",
    "eject": "react-scripts␣eject"
```

```
  },
  "eslintConfig": {
    "extends": "react-app"
  },
  "browserslist": [
    ">0.2%",
    "not␣dead",
    "not␣ie␣<=␣11",
    "not␣op_mini␣all"
  ]
}
```

In het gedeelte “scripts” staan de verschillende commando’s. Het start en build commando starten het project in development en production modus respectievelijk op. Beide voeren op hun beurt een commando van react-scripts uit. Dit gaat dus vervangen moeten worden. In de documentatie van Parcel valt te lezen dat we die commando’s moeten vervangen met de volgende:

```
{
  "scripts": {
    "start": "parcel␣src/index.html",
    "build": "parcel␣build␣src/index.html"
  }
}
```

Vervolgens moet het index.html bestand aangepast worden. Voor de wijzigingen zag het er als volgt uit:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link
      rel="stylesheet"
      href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/
         bootstrap.min.css"
    />
    <meta
      name="viewport"
      content="width=device-width,␣initial-scale=1,␣shrink-to-fit=
         no"
    />
    <title>Mortgage Calculator</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</
       noscript>
    <div id="root"></div>
  </body>
</html>
```

Merk op dat er nergens een verwijzing is naar een Javascript bestand. React-scripts voegt dat automatisch toe bij het bouwen van het project. Parcel doet dat niet dus er moet nog een expliciete verwijzing komen.

```
<!-- ... -->
<script type="module" src="index.js"></script>
<!-- ... -->
```

Nu moet Parcel uiteraard nog gedownload worden. Nadien werkt het project.

Snowpack

Snowpack gebruikt de principes van unbundled development uitgelegd in de literatuurstudie. Wat dat betekent in de werkelijkheid volgt later. Eerst kijken we hoe Mortage kan omgevormd worden van Webpack naar Snowpack.

Bij Parcel lag de focus op zo weinig mogelijk configuratie, niet bij Snowpack. Snowpack is veel moeilijker te configureren dan Parcel en Vite en voelt in dat opzicht aan als Webpack. We ondernemen dezelfde stappen als hierboven om react-scripts te verwijderen en die te vervangen door de nieuwe commando's. Daarna voegen we Snowpack toe samen met twee andere plugins voor React. We doen net dezelfde wijzigingen aan index.html, voegen een configuratie bestand toe en proberen de app te runnen. Een foutmelding verschijnt. Om die te kunnen begrijpen, is er eerst wat meer uitleg nodig.

Een React component retourneert JSX. JSX is, zonder te veel in details te gaan, een extensie bovenop Javascript die een manier biedt om de weergave van componenten te structureren en lijkt heel hard op HTML. Het belangrijke om hiervan mee te nemen is dat het een Javascript extensie is. In CRA en sommige andere frameworks is het mogelijk om JSX in een bestand te schrijven met de standaard Javascript extensie .js . Snowpack ondersteunt dit niet. Als een bestand JSX bevat, moet die de extensie .jsx krijgen. Dus moeten we alle bestanden die JSX gebruiken van bestandsextensie veranderen. Hierna werkt de app naar behoren.

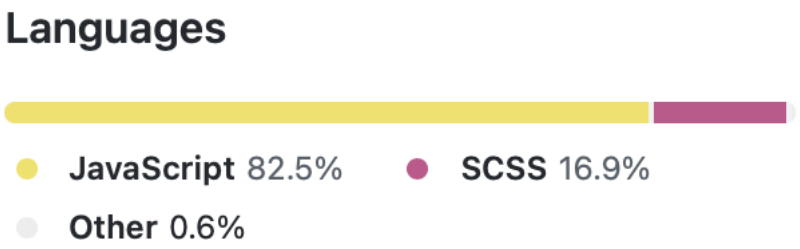
Vite

Vite combineert het beste van de twee voorgaande build tools. Het gebruikt de principes van unbundled development en dat met weinig configuratie. Om Mortage op te zetten met Vite overlopen we stappen die eerder al aan bod kwamen: react-scripts verwijderen, index.html en package.json naar de nieuwe commando's aanpassen en Vite zelf installeren. Net zoals Snowpack ondersteunt Vite geen JSX in een .js bestand en is er een plugin voor React. In tegenstelling tot Snowpack is de plugin niet nodig om het project werkend te krijgen. Na het aanpassen van de bestandsextensies en eventueel downloaden van de plugin, kan dit project zonder problemen opgestart worden.

3.2.2 Todoist

Todoist (**hadwen-2021**) is een simpele to-do webapplicatie. Een gebruiker kan taken toevoegen die nog moeten gedaan worden en aanduiden wanneer die voltooid zijn. Het is een groter project dan Mortage met een grootte van 106KB en 377MB nodemodules

maar kan nog steeds beschouwd worden als een kleine tot medium grootte app. Het heeft een database nodig om de taken te kunnen opslaan en zoals hieronder te zien is, gebruikt het een andere taal voor stijl: SCSS. SCSS is een superset i.e. een uitbreiding van gewone CSS. Aangezien een webbrowser die uitbreiding niet ondersteunt, moet het SCSS bestand omgevormd worden naar gewone CSS wanneer de applicatie start. Dit kan eventueel tot extra configuratie leiden bij het instellen van een nieuwe build tool.



Figuur 3.6: Overzicht gebruikte technologieën van het Todoist project

Parcel

Om dit project om te vormen naar Parcel worden eerst dezelfde stappen overlopen als in het vorige Parcel project. Er zijn twee mogelijke extra struikelblokken bij Todoist: het gebruik van SCSS en statische bestanden. Het eerste overkomt Parcel met glans: we moeten niets zelf configureren. De eerste keer dat Todoist wordt gestart, detecteert Parcel dat er SCCS aanwezig is en download de bijhorende plugin. Het tweede probleem vereist ook een plugin en daarenboven nog wat configuratie. In de package.json duiden we met volgende lijnen aan waar het mapje met statische bestanden zich bevindt. Nadien moeten de verwijzingen naar de afbeeldingen nog veranderd worden naar de nieuwe locatie. Hierna is ook dit project bijna klaar voor gebruik. Enkel nog het configuratie bestand voor de database moet nog aangepast worden.

Snowpack

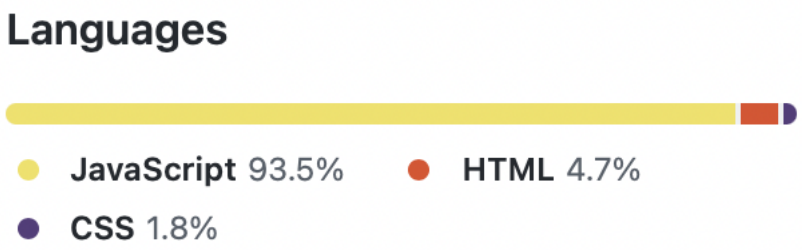
Ook hier zijn de stappen gelijkaardig aan het vorig project. Na die te doorlopen volgen nog twee potentiële moeilijkheden. In het vorig deel over Snowpack werd al vermeld dat statistische bestanden ondersteunt worden zonder extra plugin. Het tweede probleem, SCSS, kan eveneens opgelost worden door een plugin. In tegenstelling tot Parcel, moet die wel manueel toegevoegd worden.

Vite

Voor Vite kunnen we heel kort blijven. Na het uitvoeren van dezelfde stappen als vorig project, werkt alles. Zowel statische bestanden als SCCS zijn ondersteunt zonder extra plugins of configuratie.

3.2.3 Bar

Als laatste nemen we het grootste en meest complexe project van deze studie onder de loep. De BarApp (**vansteenkiste-2021**) is ontworpen om barlijsten bij te houden zonder zelf rekening te houden met de effectieve betaling. In vergelijking met de voorgaande projecten is het veel groter: 27MB voor het project zelf en dan nog eens 442MB voor de nodemodules. SCCS valt weg als moeilijkheid en wordt vervangen door twee nieuwe: TypeScript en Tailwind CSS.



Figuur 3.7: Overzicht gebruikte technologieën van het BarApp project

TypeScript is net als SCCS een superset of uitbreiding. Deze keer niet van CSS maar van Javascript. De browser begrijpt geen TypeScript dus moet het eerst vertaald worden naar normale Javascript. De build tools zijn hier verantwoordelijk voor.

De tweede moeilijkheid, Tailwind CSS, staat niet in het gebruikte-talen-overzicht van hierboven. Toch is het iets waar we rekening mee moeten houden. Het is een collectie van allemaal CSS klassen die de gebruiker direct in een JSX element kan gebruiken. Tailwind zelf bevat dus hele grote CSS bestanden om die klassen allemaal te definiëren. Aangezien we enkel de klassen willen behouden die effectief gebruikt worden in het project, worden de ongebruikte gewist bij het bouwen van de applicatie voor productie. Hoewel dit alles geen taak is van de build tool zelf, kunnen er moeilijkheden optreden bij de configuratie.

Parcel

Nog maar eens overkomt Parcel de mogelijke struikelblokken met zo goed als geen configuratie. Het ondersteunt Typescript zonder iets extra te moeten doen. Naast de standaard configuratie van Tailwind zelf, is er niets anders nodig om het werkend te krijgen. Na het overlopen van de stappen van de vorige Parcel projecten, met in het bijzonder de stap voor statische bestanden, is ook deze applicatie bijna klaar voor gebruik. Voor het laatste detail nemen we nog een kijkje naar de index.html. Aangezien CRA op een andere manier verwijst naar de afbeeldingen die hier staan, moeten de href attributen aangepast worden.

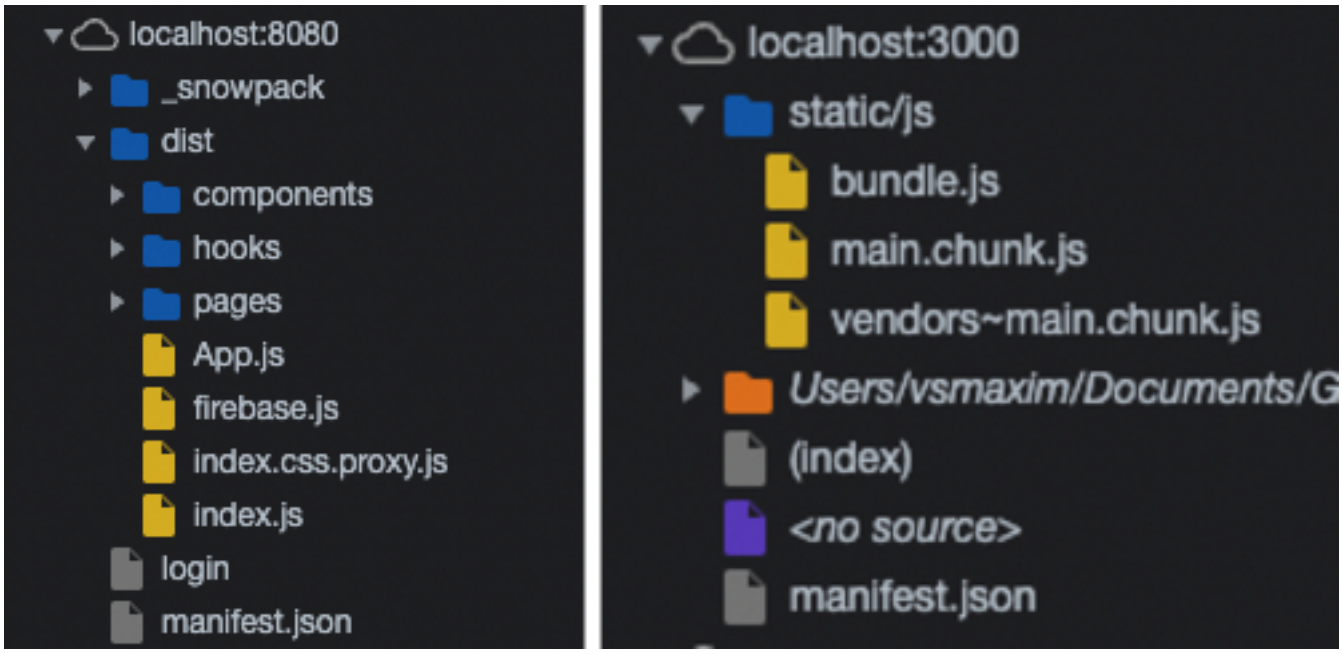
Snowpack

Om al een tipje van de sluier op te lichten: dit project was veel lastiger om op te zetten met Snowpack dan met de andere twee opties. Eerst het goede nieuws: Typescript is ook

ondersteund zonder extra configuratie. Daar stopt de vreugde al. De overige problemen worden door hun complexiteit in paragrafen onderverdeeld.

Voor Tailwind CSS valt het nog wel mee. Naast dezelfde configuratie en installatie stappen die moeten doorlopen worden als bij Parcel (en Vite), moet er nog een plugin geïnstalleerd worden voor Snowpack zelf. Hierna moet die zijn configuratie bestand aangepast worden zodat de plugin kan werken. Dat doen we door op twee verschillende plaatsen een lijn code toe te voegen.

Zoals in de literatuurstudie uitgelegd, ondersteunen moderne browsers enkel ESM. Aangezien Snowpack een unbundled build tool is, stuurt het de bestanden van het project naar de browser zonder ze te bundelen.



Figuur 3.8: Bestandsstructuur unbundled vs bundled

Merk op dat bij de unbundled versie van dit project, de bestandsstructuur dezelfde is als hoe we het project maken. Een bundler zoals Webpack, steekt alle bestanden die naar elkaar verwijzen samen in één bestand dat dan naar de browser gestuurd wordt. Unbundled build tools zoals Snowpack laten de bestandsstructuur met rust omdat ze vertrouwen op het feit dat moderne browsers ESM ondersteunen. Dus zolang er in het project enkel ESM wordt gebruikt, is er geen probleem. Jammer genoeg staat er in de App.jsx een require functie, een CommonJS module dus. Omdat Snowpack niets doet om dit te vertalen zal de CommonJS module manueel omgevormd moeten worden naar ESM. Gelukkig wordt die vertaling wel uitgevoerd voor de nodemodules, want die bevatten ook vaak CommonJS.

Voor het volgend probleem, wordt het index.jsx bestand onder de loep genomen.

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import App from "./App";
```

```
import reportWebVitals from "./reportWebVitals";
import * as serviceWorkerRegistration from "./
    serviceWorkerRegistration";

if (process.env.NODE_ENV === "production") {
  console.log(‘SINT-POL BAR‘);

  window.console = {
    log: () => {},
    info: () => {},
    warn: () => {},
    error: () => {},
  };
}

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById("root")
);

//PWA
serviceWorkerRegistration.register();

// If you want to start measuring performance in your app, pass a
    function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA
    -vitals
reportWebVitals();
```

De aangeduide code vormt een probleem. De lijn process.env.NODE env === “production” wordt gebruikt om te kijken of de app in productie draait of lokaal, in development. Naast dit voorbeeld wordt deze methode vaak toegepast in dit project. Het probleem is dat Snowpack het niet ondersteunt. Omdat dit voorbeeld niet cruciaal is, laten we het weg om Snowpack opgezet te krijgen. Jammer genoeg gebruikt deze app Workbox. Workbox is een Javascript library die een PWA opzetten, gemakkelijker maakt. Een PWA is een webapplicatie die installeerbaar is op een Smartphone of computer. Het probleem is dat Workbox ook gebruikt maakt van bovenstaande methode. Als we deze library nog steeds zouden willen gebruiken, zou dit alles ook moeten herschreven worden. Voorlopig gaan we verder zonder aangezien het veel tijd in beslag zou nemen. Op naar het laatste probleem.

React kan al even gebruikt worden zonder het te moeten importeren in een bestand. Dat wordt automatisch voor de developer gedaan. Jammer maar helaas: Snowpack ondersteunt dit niet. Aan elk Javascript bestand waar iets van de React methodes gebruikt worden, moeten een import statement toevoegen als volgt:

```
import React from "react";

// ...
```

Na dit alles te overlopen, werkt het project. Eindelijk.

Vite

Na de hele waslijst problemen die bij Snowpack opdoken, doet het deugd om weer met Vite aan de slag te mogen gaan. Van de struikelblokken die Snowpack tegenkwam, blijft voor Vite enkel nog het probleem van de CommonJS module over. Na die te herschrijven en dezelfde stappen te doorlopen als bij de vorige Vite secties, werkt dit project volledig, ook Workbox. Typescript wordt ook ondersteunt zonder extra gedoe en voor Tailwind is de gebruikelijke installatie en configuratie nodig, geen extra plugin. Net zoals bij Parcel moeten de verwijzingen naar de afbeeldingen in de index.html bijgeschaafd worden.

De conclusie over de bevindingen die in dit hoofdstuk waargenomen zijn, volgt in het volgend hoofdstuk.

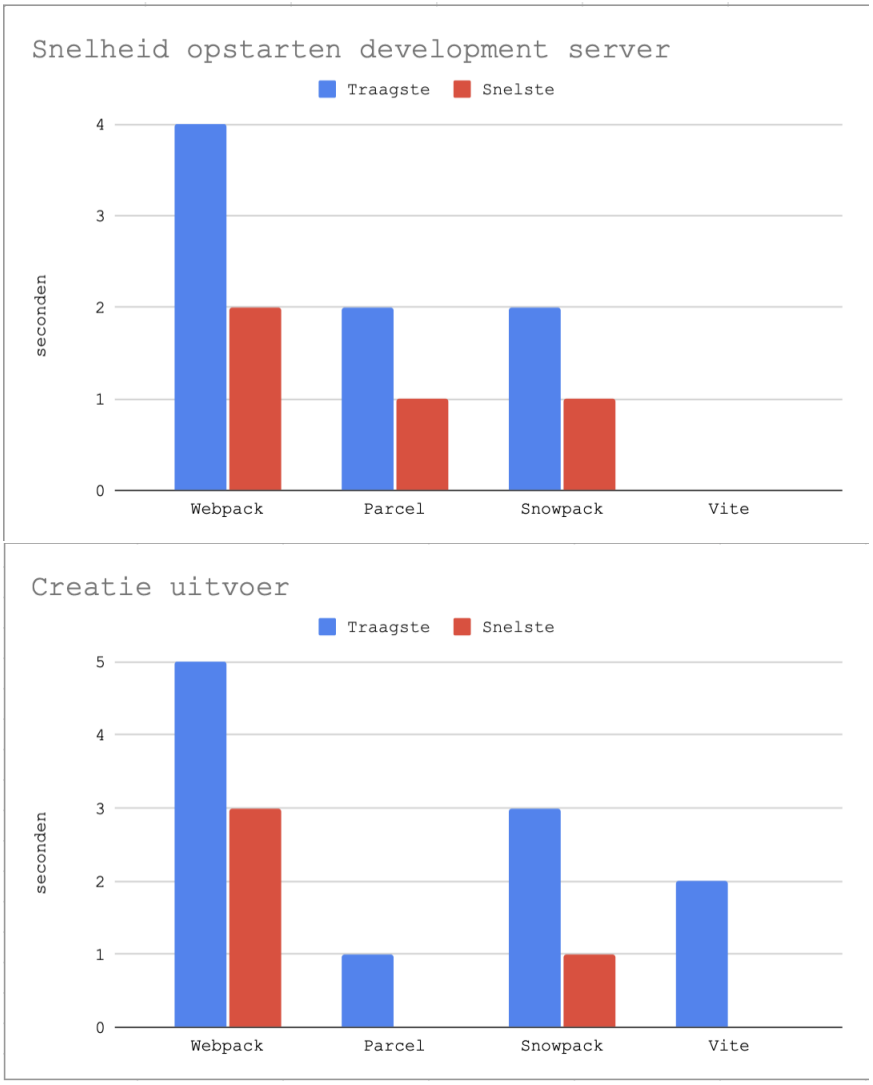
In het vorig hoofdstuk werden de gekozen module bundlers op de proef gesteld aan de hand van drie open-source projecten en een van nul te beginnen. De verschillende stappen om ze werkende te krijgen werden overlopen, bij de een waren dat er al meer dan de ander. In dit hoofdstuk wordt gekeken naar de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

4.1 Nieuw project

Al latex 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.

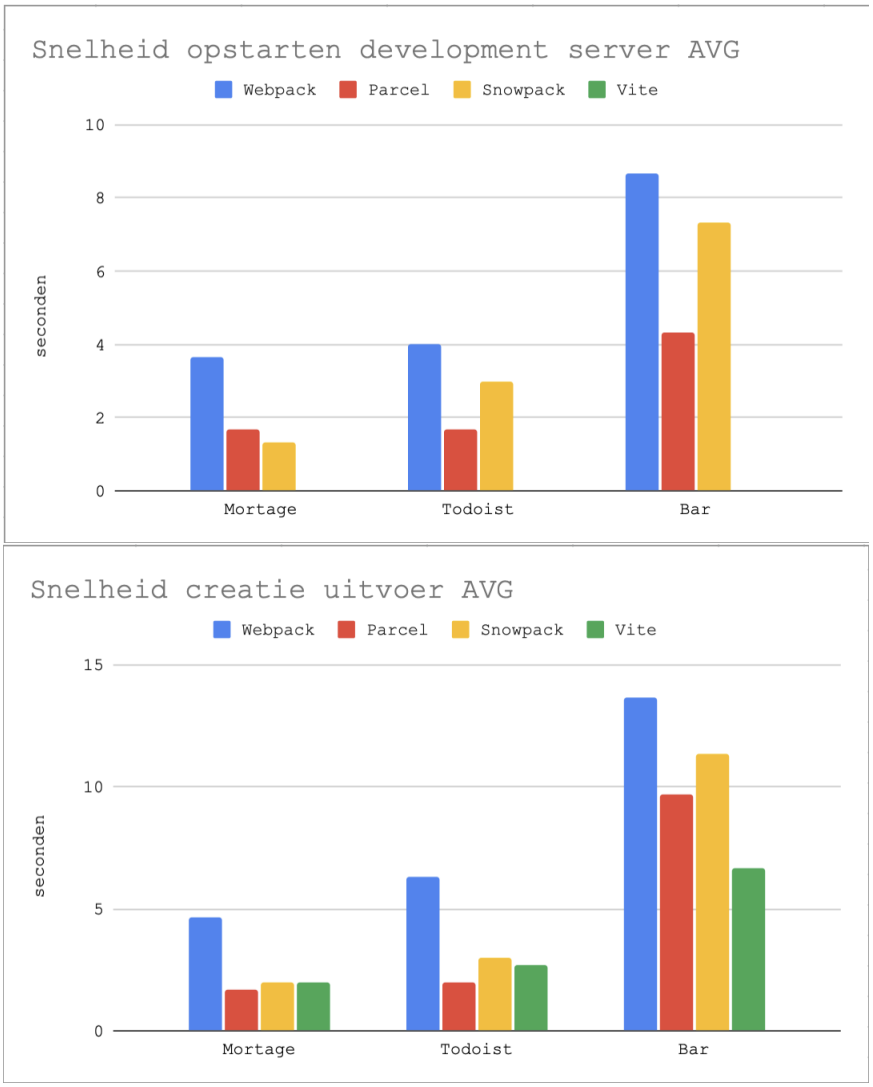


Figuur 4.1: Resulaten nieuw project

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, afhangend 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: Resulaten 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 onderzoeksvoorstel 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/