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

Woord vooraf

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.

Samenvatting

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.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

1. Inleiding

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

In Hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen.

In Hoofdstuk 4, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

2. State of the art

A modern website is created with three main technologies: HTML for content, CSS for styling, and Javascript to make the page interactive. Those three can all be linked, put on a server somewhere and the end-user can load a working website. Even though the large web platforms of today, like Facebook and Youtube, are all based on these technologies, a lot more ingredients are needed to make them work.

If we were to create a static, basic site, the three main technologies of the web are all you need. But it's a whole different story for a developer that wants to create a more complex site, a reactive site, a site that uses open-source packages or just wants to make his own job a little bit easier. There are thousands of libraries, packages and frameworks that enable web development on a whole new level and easier than ever before. However this does pose a new problem.

If we were to create a web application with a single Javascript file and no dependencies i.e no other files that are linked to that Javascript file, link it to our index.html and throw in some CSS, our site would work fine. But what if we introduce a second Javascript file and reference it in our other file. What if we want to install and use an open-source package downloaded from a package-manager like NPM? What if we want to use SASS to extend CSS? This all wouldn't work out of the box. The browser just isn't able to figure out how to stitch all the different pieces together. We need something that will bundle all the different Javascript files and dependencies, something that understands and correctly loads the SASS files or any other file for that matter. We need a module bundler. In essence, they take all the different source files in a project and put them into a single output file that the browser understands.

2.1 History

To understand the rest of this paper, you'll need to know what a module is. Modular programming breaks a program up into chunks based on functionality and often in separate files. These chunks are called modules. Modules are then linked together to form an application. (Webpack, g.d.-a) (Mozilla, 2021b)

Node.js, a Javascript runtime for computers and servers, has supported modular programming since the beginning using CommonJS. However support for modules on the web has been slow to arrive. The first module bundlers only came around after 2014. To understand why module bundlers exist, we should first know how web applications were built before them and what problem they solved. (Webpack, g.d.-b)

2.1.1 Script tags

Javascript can be linked to, or written directly in, the HTML file of a site using script-tags. Knowing this, we could use a different tag and corresponding Javascript file for each use case. Let's say we have a file with all the code for authenticating a user and another one for general events (button clicks, ... This is fine when we only have two files that aren't that big, but introduces network bottlenecks when scaled to a larger application. The same is true if we were to put all our code in one large JS file and link it to the HTML. The global scope would also get polluted with our custom functions. As some, or all, the functions are available on the global scope, this could introduce security risks and name collisions.

```
<!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

An Immediately invoked function expression or IIFE is a function that runs as soon as it's defined (Mozilla, 2021a). Because each IIFE declares a local scope, it solves the problem of polluting the global one. The use of IFFEs led to so-called task runners: they concatenate all your project files together. The big drawback of task runners is that when one file is changed, the whole project has to be rebuilt. You are also required to manually define all the dependencies up front. They make it easier to reuse functions and whole scripts but do nothing for the build output. You can still end up with a very large Javascript file that the user has to download.

```
(function () {
  let firstVariable;
  let secondVariable;
```

```
  // ...
})();

// firstVariable and secondVariable will be discarded after the
   function is executed.
```

2.1.3 CommonJS

Before 2009, Javascript ran only in a browser. Node.js introduced a Javascript runtime that could run on computers and servers. This introduced a new set of challenges (Crutchfield, 2018). As Javascript wasn't run in the browser and therefore no HTML script-tags were around, how could those applications load new chunks of code?

CommonJS introduced the require function in Javascript. With it, everything that an external module exports, can be imported. Reusable code can now be imported from any other Javascript file in a project. It makes implementing dependency management easy to understand.

All this came with a big catch: It worked, and still works, great for Node.js applications but it isn't an official feature of Javascript and therefore browsers don't support it. As commonJS doesn't actually bundle the code, web browsers can't make sense of the different modules that are imported. Something has to do it for them.

```
//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 wasn't an official feature of Javascript. ECMAscript (=Javascript) did introduce its own module system in version 6. ECMAScript Modules or ESM, accomplish the same goals as CommonJS, but with a different syntax. Now modern browsers can make sense of modular applications that only use ESMs.

```
//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

Developers are always seeking ways to make their lives easier. They wanted to import whatever type of module or any asset for that matter into their project and ship it in a smaller output file than the source to the end-user. This is why the module bundler was born.

The most essential function of a module bundler is following all imports of a project, that can contain many files, and bundling them into a single file called the bundle. They also minify that bundle to be as small as possible, without affecting its functionality. It does this by removing comments, white-spaces, new lines, ...

Unbundled file: 147 bytes

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

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

Bundled file: 97 bytes

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

All module bundlers around today share these functions along with common concepts. We'll look at the 2 most important ones.

Tree shaking

Tree shaking is the process of removing dead code. When a module is imported, perhaps only a part of that module is needed. Maybe only one function of that module is used in the project. With tree shaking, the rest of that module that isn't used, is removed.

Code splitting

Code splitting can be used to divide up the output bundle that is created into smaller files. The partial bundles are then loaded in parallel or when needed. For example: take a website with multiple pages. If the code isn't split, all the code of all the pages will be contained in a single file and downloaded by the user when the site is requested. In many cases, like a small project, this is fine. That's why it's optional. For larger applications however, it could introduce network bottlenecks. If that's the case, the code can be split into a single

or multiple files per page. Then the smaller files are only loaded when the corresponding page is requested.

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 Bundled development

In this section, examples of module bundlers that use bundled development will be discussed. Almost every Javascript web bundler is based on the concepts of bundled development. The alternative and more modern approach is unbundled development. The differences and examples will be explained in the next section.

Webpack

Webpack is the most popular module bundler in the world according to NPM downloads. This has much to do with the fact that it's been around the longest. It comes pre-installed in many web frameworks like create-react-app and Next.js. This way, it's used by many without even knowing it. It comes with an optional built in development server that makes setting up a local development environment easier.

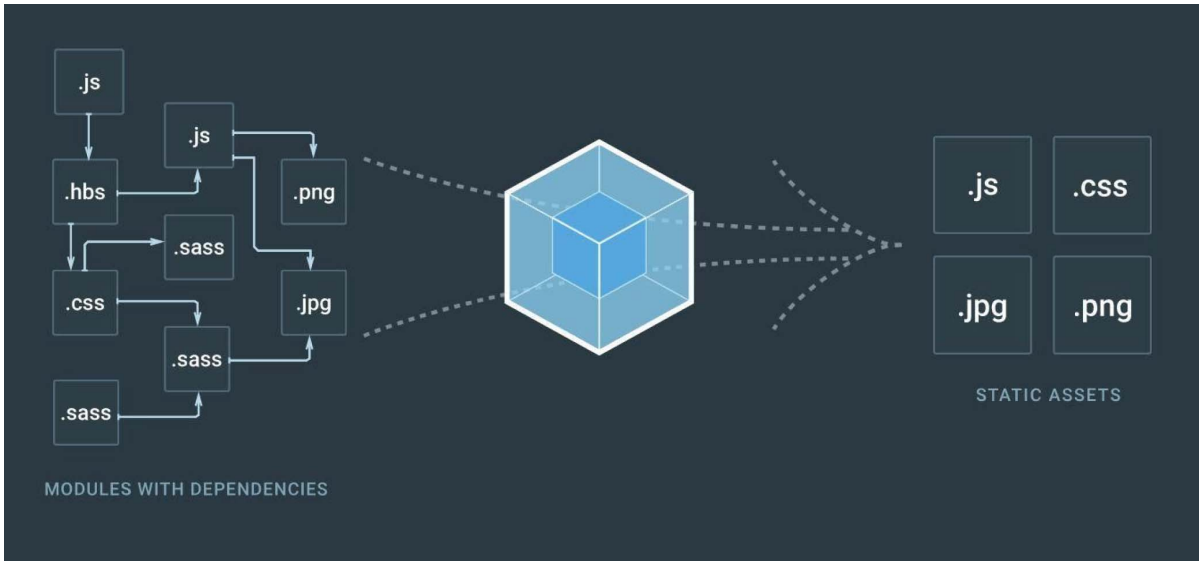
Webpack runs on Node.js. It can do its job without any configuration, however is very configurable if needed. It supports the module types discussed above and more. Because Webpack only understands Javascript and JSON files out of the box, Loaders are used to allow processing of other file types and convert them into valid modules. Using Loaders, other types of modules or even assets like images can be imported and processed by WebPack. On top of that, Webpack can also be extended with plugins. They allow for a wide range of extra functionality like bundle optimization.

The function of a module bundler has already been discussed. But how does Webpack achieve this? Whenever one file depends on another in a project, Webpack sees this dependency and puts it into something called a dependency graph.

This graph is built recursively. When it is time to build the application, it uses the graph to piece all the files together into one output file that is then shipped to the browser. Webpack does this both in development as in production.

Parcel

Parcel is another Javascript bundler. However it doesn't run on Node.js, instead the compiler it uses is built with Rust. Rust is a compiled programming language. Without



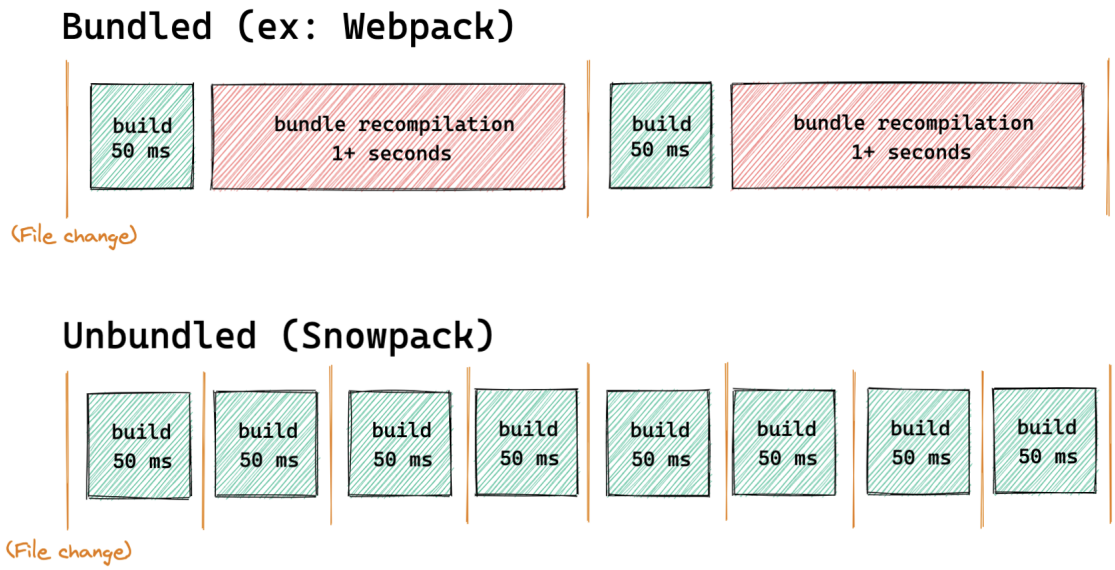
Figuur 2.1: Essentiele functie module bundler (Webapck, g.d.)

going into too much detail, this means that Rust code is compiled directly to machine code resulting in faster performance. It does many of the same things that Webpack does without any configuration. When it was released, the main feature was that it didn't need a configuration file and Webpack did. Now however, Webpack can also do its job without one. While a no-configuration approach is great for smaller projects, it isn't feasible in a large application.

2.2.2 Unbundled development

The examples we looked at so far are all module bundlers that use the concepts of bundled development. One very important thing to note is the word development. The differences between both approaches lies only in the development phase. The advantages of unbundled development are only noticeable when the developer or the development team is actually developing the application. When it is time to make a production build that the end-user will be able to use, the following unbundled development build tools still use the same module bundlers discussed in the previous section. So in essence, we can't call the unbundled development build tools module bundlers, because they don't bundle anything. They do use module bundlers.

Because most browsers now support ESM, bundling all Javascript modules of a project into one output file isn't necessary. All non-Javascript modules do have to be translated or built to a valid Javascript module using a built tool (in WebPack this is called a Loader). In bundled development, when the development server is started, the whole project first has to be built and bundled; When a file is changed that file is rebuilt and then the whole project is again rebundled. In unbundled development a file is only built when it's requested, meaning very fast startup time of the server. When a file is built, it is cached indefinitely. The browser will never have to download a file twice until it changes. When a file does change, only that single file has to be rebuilt.



Figuur 2.2: Gebundeld vs ongebundeld (Snowpack, g.d.)

All this results in very fast development performance compared to bundled development tools like WebPack. However, as mentioned before, unbundled development tools still use traditional module bundlers to make a production build. It does this because the advantages of Tree-shaking, a single output file and other features of module bundlers still hold up in the production phase: Tree-shaking results in a smaller output file; Not all modules are written with ESM; and the list goes on. Omdat volgende voorbeelden eigenlijk geen module bundlers zijn (want ze bundelen zelf niets), worden ze build tools genoemd. In het volgende hoofdstuk worden de twee populairste besproken.

Snowpack

Snowpack is het eerste voorbeeld van een unbundeld 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 unbundled build tool vaak toch een module bundler in productie. Bij Snowpack is dit 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 werking representeren, terwijl Vite hetzelfde zal doen voor de half-ongebundelde werking.

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, g.d.), een alternatief aan Webpack. Hoewel het ook een configuratie bestand bevat, voelt die minder zwaar aan als bij Snowpack.

Zoals de naam Vite doet vermoeden, beweren de makers ervan bliksemsnelle opstarttijden in vergelijking met de competitie.

3. Methodologie

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

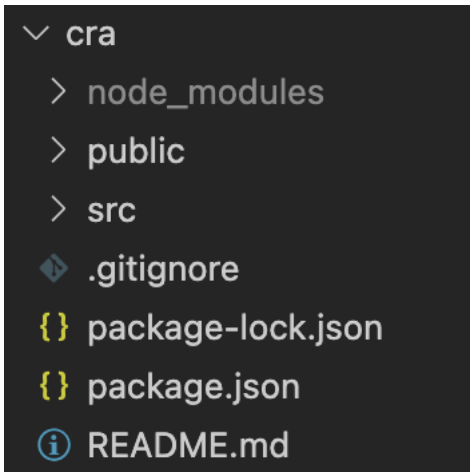
3.1 Nieuw project

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

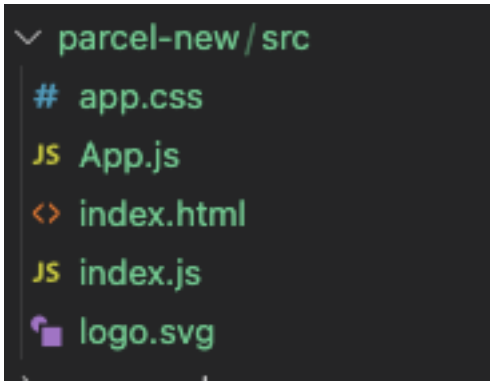
3.1.2 Parcel

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.

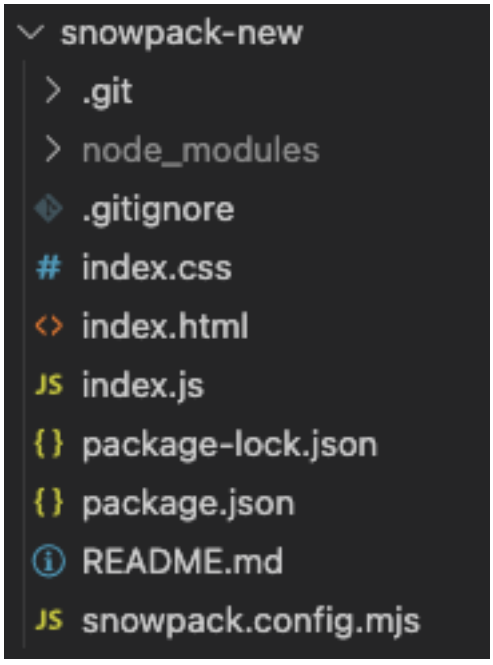


Figuur 3.2: Aan te maken bestanden voor nieuw Parcel project

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.

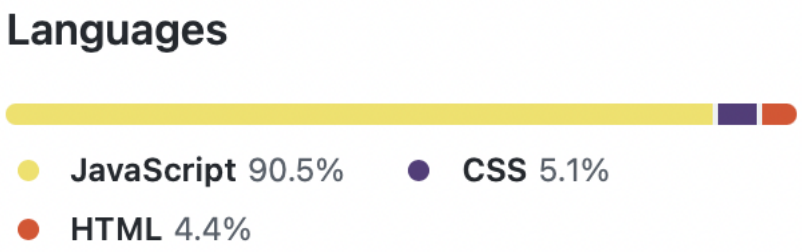
3.2 Bestaand project

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.

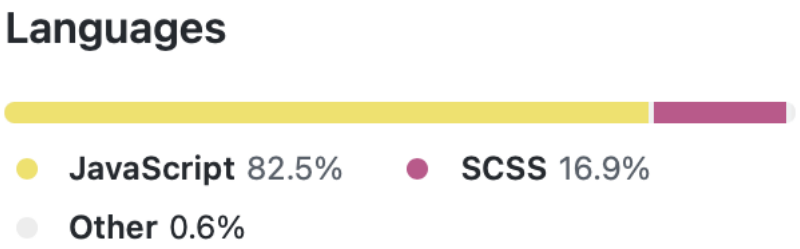
Vite

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

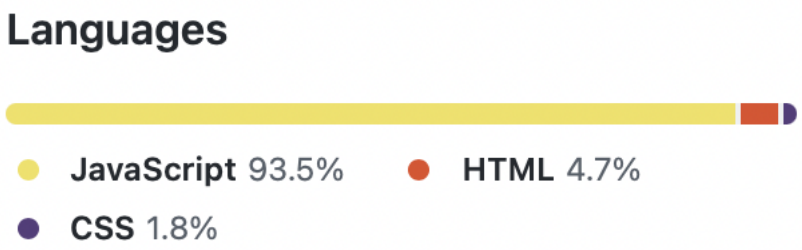
Ook hier zijn de stappen gelijkaardig aan het vorig project. Na die te doorlopen volgen nog twee potentiële moeilijkheden. In het vorig deel over Snowpack werd al vermeld dat statistische bestanden ondersteunt worden zonder extra plugin. Het tweede probleem, SCSS, kan eveneens opgelost worden door een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

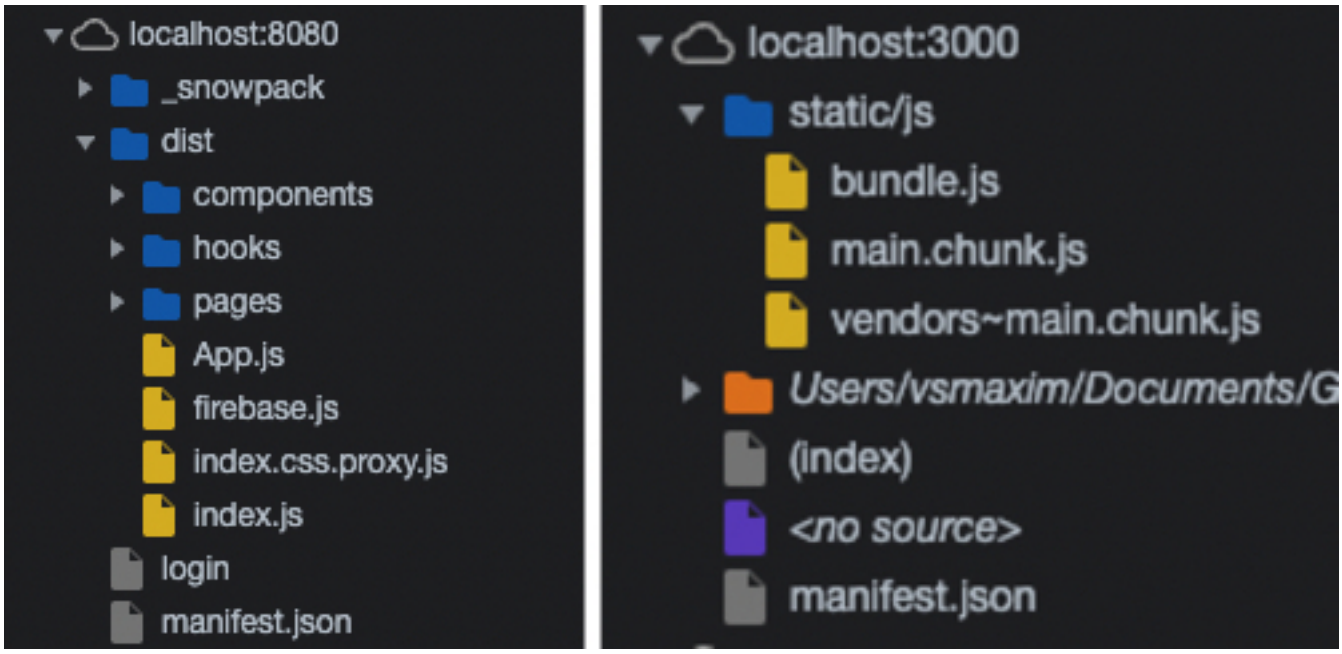
Snowpack

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.

4. Conclusie

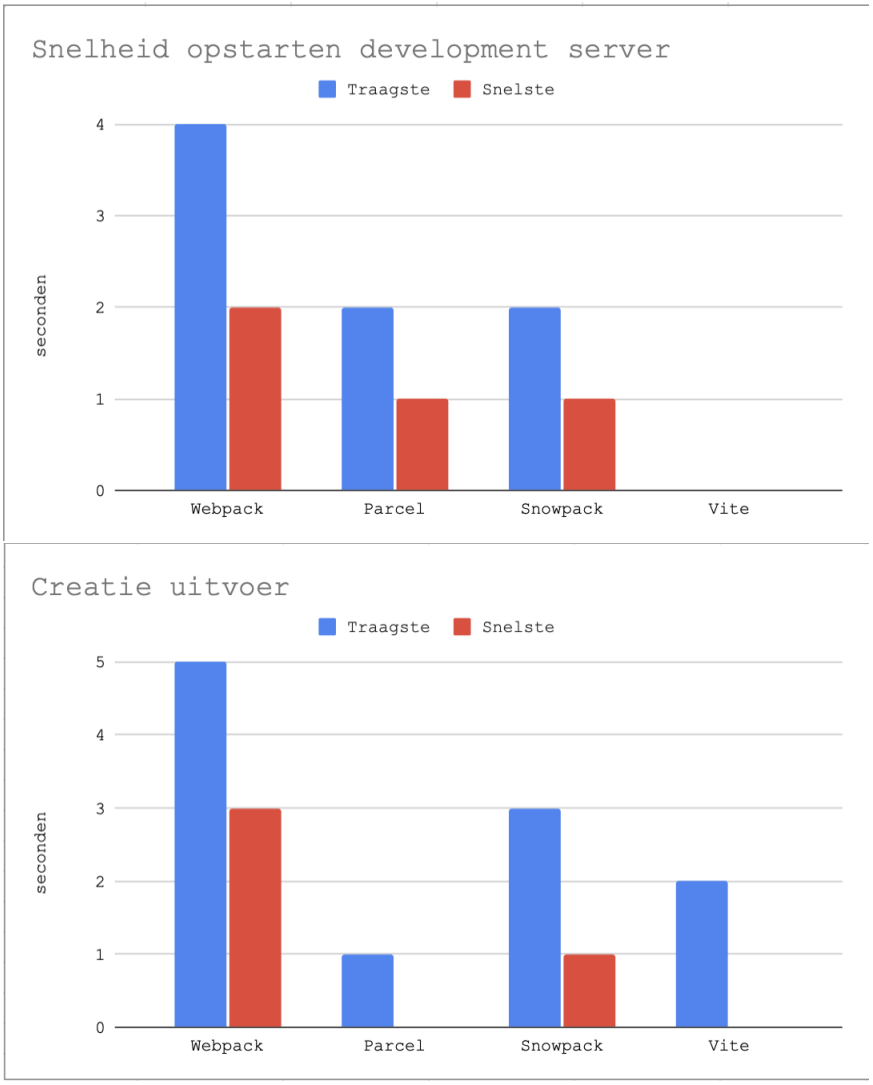
In het vorig hoofdstuk werden de gekozen module bundlers op de proef gesteld aan de hand van drie open-source projecten en een van nul te beginnen. De verschillende stappen om ze werkende te krijgen werden overlopen, bij de een waren dat er al meer dan de ander. In dit hoofdstuk wordt gekeken naar de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

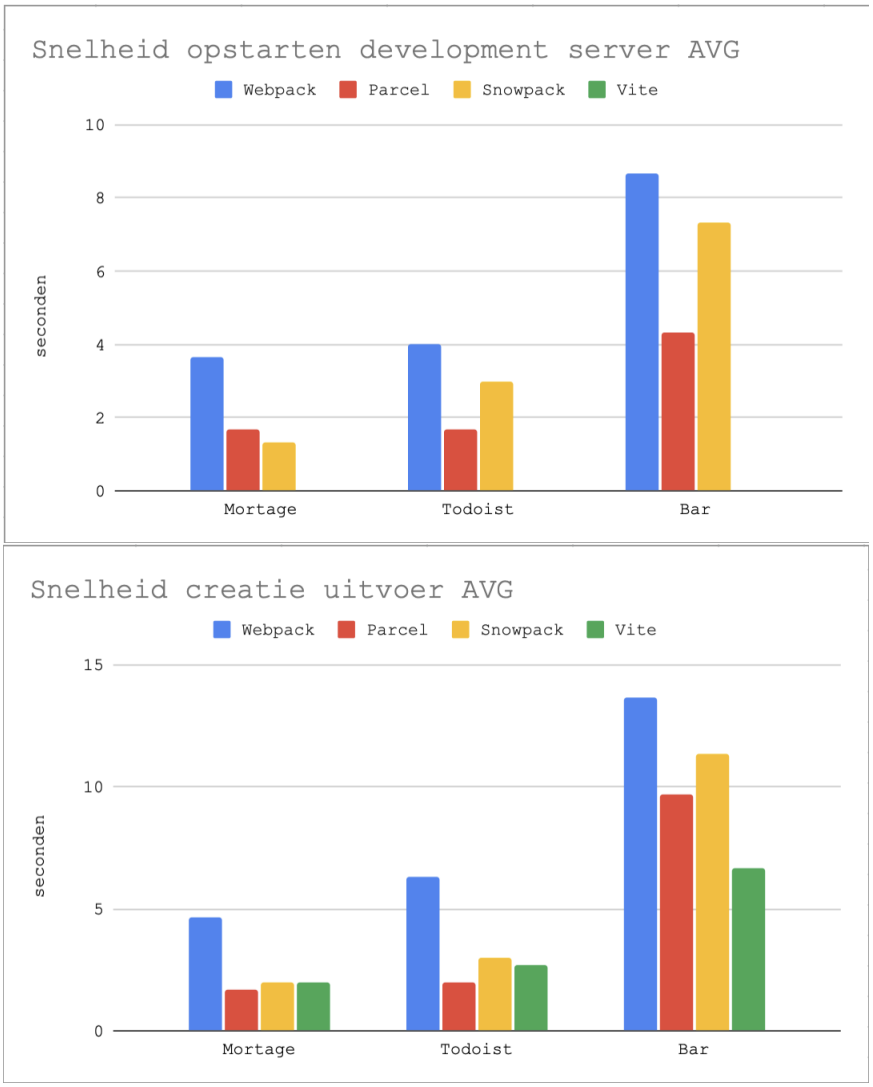


Figuur 4.1: Resulaten nieuw project

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.

A. Onderzoeksvoorstel

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/

Bibliografie

Crutchfield, C. . (2018, oktober 22). *CommonJS ... what, why and how*. Verkregen 2 december 2021, van https://medium.com/@cgcrutch18/commonjs-what-why-and-how-64ed9f31aa46

Facebook. (2018, december 6). *create-react-app repository*. Verkregen 21 december 2021, van https://github.com/facebook/create-react-app/tree/main/packages/react-scripts

Facebook. (2021, september 22). *create-react-app/webpack.config.js*. Verkregen 21 december 2021, van https://github.com/facebook/create-react-app/blob/main/packages/react-scripts/config/webpack.config.js

Hadwen, K. . (2021, september 14). *Karlhadwen/todoist*. Verkregen 21 december 2021, van https://github.com/karlhadwen/todoist

Houghton, P. . (2019, februari 8). *GitHub - paulhoughton/mortgage: Mortgage overpayment calculator*. Verkregen 21 december 2021, van https://github.com/paulhoughton/mortgage

Mozilla. (2021a, december 3). *IIFE - MDN Web Docs Glossary*. Verkregen 2 december 2021, van https://developer.mozilla.org/en-US/docs/Glossary/IIFE

Mozilla. (2021b, oktober 15). *JavaScript modules*. Verkregen 2 december 2021, van https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

Snowpack. (g.d.). *Unbundled vs bundled development*. https://www.snowpack.dev/concepts/how-snowpack-works#unbundled-development

StateofJS. (2020). *State of JS 2020: Build Tools*. Verkregen 15 december 2021, van https://2020.stateofjs.com/en-us/technologies/build-tools/

Vansteenkiste. (2021). *MaximVansteenkiste/bar-react*. Verkregen 21 december 2021, van https://github.com/MaximVansteenkiste/bar-react

Vite. (g.d.). *Building for Production | Vite*. Verkregen 8 december 2021, van https://vitejs.dev/guide/build.html

Webapck. (g.d.). *Webpack overzicht module bundler*. https://webpack.js.org

Webpack. (g.d.-a). *Modules history*. Verkregen 2 december 2021, van https://webpack.js.org/concepts/modules/

Webpack. (g.d.-b). *Why Webpack?* Verkregen 2 december 2021, van https://webpack.js.org/concepts/why-webpack/