FH Salzburg
MultiMediaTechnology

# *Unterstützung von Tree Shaking durch statische Analyse*

## Bachelorarbeit 2

AutorIn: Thomas Siller
BetreuerIn: Mag Hannes Moser

Salzburg, Österreich, 26.02.2018

## Eidesstattliche Erklärung

Ich erkläre hiermit eidesstattlich, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, und keine anderen als die angegeben Quellen und Hilfsmittel benutzt habe. Weiter versichere ich hiermit, dass ich die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission weder im In- noch im Ausland vorgelegt und auch nicht veröffentlicht.

*Datum* *Unterschrift*

*Vorname* *Nachname*

# Kurzfassung

Mit *Tree Shaking* bieten *Static Bundler* wie Webpack und RollupJS einen Service, welcher hilft nicht benötigte Code Segmente automatisch aus den kompilierten JavaScript Dateien zu entfernen. Als Teil dieser Arbeit werden Schwierigkeiten und *Best Practices* im Zusammenhang mit *Tree Shaking* ermittelt. Dies geschieht mittels einer Literaturrecherche. Weitere Erkenntnisse werden durch die Untersuchung von JavaScript Projekten auf GitHub, welche *Tree Shaking* erfolgreich verwenden, gewonnen.

Die statische Code Analyse, oder auch *Linting*, hilft Entwicklern und Entwicklerinnen Fehler im Programmcode vor dem Kompilieren der Anwendung zu erkennen. In der Folge wird ein Plugin für den *JavaScript Linter* ESLint erstellt. Diese überprüft die ermittelten Anforderungen für ein erfolgreiches *Tree Shaking* und zeigt gegebenenfalls Warnungen oder Fehler für den Benutzer oder die Benutzerin an. Darauffolgend wird eine statische Code Analyse mit dem *Tree Shaking Plugin* bei sechs NPM Bibliotheken durchgeführt und die Ergebnisse dokumentiert.

Am Ende dieser Arbeit werden anhand der gewonnenen Erkenntnisse Schlussfolgerungen getroffen und ein Ausblick auf mögliche weiterführende Forschung gegeben.

# Abstract

With *Tree Shaking*, static bundlers such as Webpack and RollupJS provide a service that helps to automatically remove unused code segments from the compiled JavaScript files. As part of this thesis, difficulties and best practices related to *Tree Shaking* are identified. This is done with the assistance of a literature search. Further insights are gained by analysing JavaScript projects on GitHub that successfully implement *Tree Shaking*.

Static code analysis, or linting, helps developers to identify bugs in the program code before actually compiling the application. Subsequently, a plugin for the *JavaScript Linter* ESLint will be created. It checks the requirements for successful *Tree Shaking* and, if necessary, it displays warnings or errors for the user. A static code analysis with the *Tree Shaking Plugin* is performed on six NPM libraries and the results are documented.

At the end of this work, conclusions will be drawn from the findings and an outlook on possible further research will be given.

# Inhaltsverzeichnis

# 1 Einleitung

Für Javascript Webentwickler ist das NPM Registry eines der wichtigsten Werkzeuge. Es bietet eine Plattform für Entwickler und Entwicklerinnen, um Sourcecode mit anderen Entwicklern und Entwicklerinnen zu teilen, bestehende Module herunterzuladen und in die eigene Anwendung zu integrieren. Seit dem Release wurden über 600.000 verschiedene Software Pakete hinzugefügt. Mit mehr als einer Millarde Downloads pro Woche ist das NPM Registry zu einer zentralen Bibliothek von Frameworks, Libraries und Werkzeugen für die Javascript Entwicklung geworden. (NpmDocs 2018)

Die Anzahl der Module und deren komplexe Abhängigkeiten zueinander führten im Bereich der Frontendentwicklung zu fundamentalen Problemen. Neben der Dependency Hell, einem der wohl bekanntesten dieser Probleme, ist es entscheidend, die an den Browser zu übertragende Datenmenge möglichst gering zu halten. Um dies zu erreichen, müssen die Entwickler und die Entwicklerinnen unnötige Versions Duplikate und unerreichbare Zweige im Programmcode vermeiden.(Dominik Wilkowski 2018)

Die benötigte Datenmenge einer Website hat einen negativen Effekt auf die Antwortzeit für Benutzer und Benutzerinnen. Es existiert bereits reichlich Literatur zu den Auswirkungen der Downloadzeit zur Nutzerzufriedenheit. Bereits 1997 erkannte Nielson (1997) die Bedeutung der Lade Geschwindigkeit einer Website auf die *User Experience*. In den 90er Jahren waren die Größe von Bilder einer der Hauptgründe für eine schlechte Performance. Mehr als 10 Jahre später ist die Auswirkung der Bildergröße auf die Ladezeit aufgrund der allgemein schnelleren Internetverbindungen nicht mehr so schwerwiegend. Auch wenn sich die Gründe für eine langsame Websites verändert haben, bleiben die Limits für die Antwortzeit gleich. (Nielson 2010)

Die folgende Liste zeigt die Auswirkung der Antwortzeiten:

- 0.1 Sekunden gibt das Gefühl einer sofortigen Antwort. Das Ergebnis fühlt sich an, als wäre es vom Benutzer verursacht worden, nicht vom Computer. Dieses Level der Reaktionsfähigkeit wird von den Benutzern oder Benutzerinnen als direkte Manipulation empfunden.
- 1 Sekunde hält den Gedankenfluss des Benutzers nahtlos. Benutzer können eine Verzögerung wahrnehmen und somit wissen sie, dass der Computer das Ergebnis erzeugt. Sie fühlen sich immer noch in der Kontrolle über das Gesamtergebnis und haben nicht das Gefühl auf den Computer zu warten. Dieser Grad an Reaktionsfähigkeit ist für eine gute Navigation erforderlich.
- 10 Sekunden hält die Aufmerksamkeit des Benutzers oder der Benutzerin. In dem Bereich von 1-10 Sekunden fühlen sich die Benutzer und Benutzerinnen dem Computer ausgeliefert und wünschen sich eine shcnellere Reaktion. Nach 10 Sekunden fangen sie an über andere Dinge nachzudenken, was es schwieriger macht, ihre Gehirne wieder in die Spur zu bringen, wenn der Computer reagiert.

Jedes eingesparte Bit verringert die Zeit bis der Nutzer oder die Nutzerin eine Antwort erhält und hat somit eine positive Auswirkung auf die Benutzerzufriedenheit. Im Bereich der JavaS-

cript Webentwicklung wird die Verrrigerung der Datenmenge vorallem durch die Minification des Programmcode erreicht. Der *Static Bundler* Webpack bietet, seit der Veröffentlichung von Version 2 die Möglichkeit, mit Hilfe von *Tree Shaking* nicht benutzten Code zu bereinigen und die Datenmenge intensiver zu verkleinern. (WebpackTreeShaking 2018)

# 2 Static Module Bundler

*Static Module Bundler* wie Webpack und RollupJS ermittelen alle für die Anwendung benötigten Module und bündeln diese anschließend in einer oder mehreren Dateien. Dafür wird ausgehend vom konfigurierten Einstiegspunkten ein Baum aus Abhängigkeiten erzeugt. Neben dem eigenen Programmcode beinhaltet dieser auch Npm Packete und statische Assets wie Bilder oder CSS Dateien. Die Abhängigkeiten werden aufgelöst, um daraus je nach Konfiguration eine oder mehrere *Bundle* Dateien zu erzeugen. Je nach Anwendungsfall kann es von Vorteil sein den Programmcode als eine größere Datei oder in mehreren Teilen an den Browser auszuliefern, um ein möglichst schnelles Laden der Website zu gewährleisten. (RollupJSDocs 2018; WebpackConcepts 2018)

*Bundler* wie Webpack umhüllen jedes Code Modul mit Funktionen. Damit wird eine browserfreundliche Implementierung und das Laden der Assets je nach Bedarf gewährleistet (Rich Harris 2018). Zu beachten ist der Unterschied zwischen Laden und Bündlen von Modulen. Werkzeuge wie SystemJS werden verwendet, um Module zur Laufzeit im Browser zu laden und zu übertragen. Im Gegensatz dazu werden bei Webpack Module durch sogenannte *Loader transpiliert* und gebündelt bevor sie den Browser erreichen. (WebpackComparison 2018)

Dabei hat jede Methode ihre Vorteile und Nachteile. Das Laden und Übertragen von Modulen zur Laufzeit kann viel Aufwand für größere Sites und Anwendungen mit vielen Modulen verursachen. Aus diesem Grund ist SystemJS sinnvoll für kleinere Projekte, bei denen weniger Module benötigt werden. (WebpackComparison 2018)

# 3 Modules in JavaScript

In traditionellen JavaScript Webprojekten werden alle Abhängigkeiten als Liste aus `<script>` tags definiert. Es ist die Aufgabe des Entwicklers oder der Entwicklerin sicherzustellen, dass diese in der richtigen Reihenfolge zur Verfügung gestellt werden. Je komplexer die Abhängigkeiten zwischen dem Programmcode, Frameworks und Libraries werden, desto schwieriger wird die Wartung und die richtige Platzierung neuer Einträge in der richtigen Reihenfolge. Es wurde bereits mit verschiedenen Modul Spezifikationen versucht diese Schwierigkeiten für Entwickler und Entwicklerinnen zu vereinfachen. (Sebastian Peyrott 2018)

## 3.1 CommonJS

Das Ziel von CommonJS ist eine Modul Spezifikation zur Erleichterung der Entwicklung von Serverseitigen JavaScript Anwendungen. NodeJS Entwickler und Entwicklerinnen verfolgten anfangs diesen Ansatz, entschieden sich im späteren Verlauf jedoch dagegen. Dennoch wurde die Implementierung stark von der CommonJS Spezifikation beeinflusst. (Sebastian Peyrott 2018)

```
1  // In circle.js
2  const PI = Math.PI;
3
4  exports.area = (r) => PI * r * r;
5
6  exports.circumference = (r) => 2 * PI * r;
7
8  // In some file
9  const circle = require('./circle.js');
10 console.log( `The area of a circle of radius 4 is ${circle.area(4)}`);
```

Listing 1: CommonJS Module

In Listing 1 erkennt man ein ComonJS JavaScript Modul und dessen Verwendung. Sowohl bei CommonJS als auch bei der NodeJS Implementierung gibt es zwei Elemente um mit dem Modul System zu interagieren: `require` und `exports`. `require` wird benötigt um ein anderes Modul in den aktuellen Scope zu importieren. Dabei wird als Parameter die Modul-id angegeben. Diese ist entweder der Name des Moduls innerhalb des `node_modules` Ordner oder der gesamte Pfad. `exports` dient zur Definition einer Modul Schnittstelle. Jede *Property* des `exports` Objekts wird als öffentliches Element exportiert. Der Unterschied zwischen NodeJS und CommonJS liegt vorallem im `module.exports` Objekt. Dies erfüllt dieselbe Aufgabe wie bei CommonJS das `exports` Objekt. Abschließend bleibt zu erwähnen, dass die Module synchron geladen werden, das heißt in dem Moment und in der Reihenfolge in der sie mit `require` angegeben wurden.(Sebastian Peyrott 2018)

## 3.2 Asynchronous Model Definition (AMD)

Das Entwickler Team von AMD spaltete sich wärend der Entwicklung von CommonJS ab. Der Hauptunterschied zwischen diese beiden Systemen liegt in dem asynchronen Laden von Ressourcen. (Sebastian Peyrott 2018)

```
1  //Calling define with a dependency array and a factory function
2  define(['dep1', 'dep2'], function (dep1, dep2) {
3
4      //Define the module value by returning a value.
5      return function () {};
6  });
7
8  // Or:
9  define(function (require) {
10     var dep1 = require('dep1'),
```

```
11          dep2 = require('dep2');
12
13      return function () {};
14  });
```

Listing 2: Asynchronous Model Definition

Listing 2 zeigt ein Beispiel für die Definition eines AMD Moduls. Das asynchrone Laden wird durch einen Funktionsaufruf nach dem Anfordern der Abhängigkeiten ermöglicht. Libraries, die nicht voneinander abhängig sind, können somit zur selben Zeit geladen werden. Dies ist besonders wichtig für die Frontend Entwicklung, da dort die Startzeit einer Anwendung essenziell für ein gutes Benutzererlebnis ist. (Sebastian Peyrott 2018)

## 3.3 ECMAScript 6

Ziel der ECMAScript 6 Modules Spezifikation war es, ein Format zu kreieren, welches sowohl AMD, als auch CommonJS Benutzer und Benutzerinnen zufriedenstellt. Die Spezifikation besitzt eine kompaktere Syntax als CommonJS und ähnlich zu AMD wird asynchrones Nachladen direkt unterstützt. Neben diesen Vorteilen wurde auch die Möglichkeit der statischen Code Analyse für Optimierungen geschaffen.

ES6 bietet 2 Arten von Exporten: *Named Exports* (mehrere Exporte pro Modul) und *Default Exports* (einem Export pro Modul). (Rauschmayer 2014)

### 3.3.1 Named Exports

Mit Hilfe des Präfixes `export` ist es möglich mehrere Elemente aus einem Modul zu exportieren. Dies gilt sowohl für Variablen als auch für Funktionen und Klassen. Sie werden anhand ihres Namens unterschieden und werden deshalb auch als *Named Exports* bezeichnet. (Rauschmayer 2014)

```
1  //------ lib.js ------
2  export const sqrt = Math.sqrt;
3  export function square(x) {
4      return x * x;
5  }
6  export function diag(x, y) {
7      return sqrt(square(x) + square(y));
8  }
9
10 //------ main.js ------
11 import { square, diag } from 'lib';
12 console.log(square(11)); // 121
13 console.log(diag(4, 3)); // 5
```

Listing 3: Named Exports

Neben dem gezeigt Weg in Listing 3, gibt es noch weitere Möglichkeiten um *Named Exports* zu erzeugen. Eine Liste der verschiedenen Schreibweisen von ES6 Modul Importe[1] und Exporte[2] kann auf der MDN Web Docs Website gefunden werden.

### 3.3.2 Default Exports

Es gibt den Fall, dass ein Modul nur ein Objekt exportiert. In der Frontendentwicklung tritt dies häufig auf, wenn nur eine Klasse oder ein Konstruktor implementiert wird. Dabei wird der *Default Export* verwendet. Er ist das zentrale Export Element eines ECMAScript 6 Moduls und daher leichter als *Named Exports* zu importieren. (Rauschmayer 2014)

```
1  //------ myFunc.js ------
2  export default function () { ... };
3
4  //------ main1.js ------
5  import myFunc from 'myFunc';
6  myFunc();
7
8  //------ MyClass.js ------
9  export default class { ... };
10
11 //------ main2.js ------
12 import MyClass from 'MyClass';
13 let inst = new MyClass();
```

Listing 4: Default Exports

Wie Listing 4 veranschaulicht, besitzt der *Default Export* keinen Namen. Beim Importieren wird daher meist der Modul Name für die Identifizierung verwendet.

```
1  import { default as foo } from 'lib';
2  import foo from 'lib';
```

Listing 5: Import Deklaration

```
1  //------ module1.js ------
2  export default 123;
3
4  //------ module2.js ------
5  const D = 123;
6  export { D as default };
```

Listing 6: Export Deklaration

In Wirklichkeit ist der *Default Export* auch ein *Named Export* mit dem speziellen Namen `default`. Somit sind die Import und Export Deklarationen in Listing 5 und Listing 6 gleichbedeutend. (Rauschmayer 2014)

1. https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Statements/import - besucht am 18.07.2018
2. https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Statements/export - besucht am 18.07.2018

### 3.3.3 Static Module Structure

Bei anderen Module Systemen wie CommonJS muss man den Programmcode ausführen, um herauszufinden, welche Importe und Exporte verwendet werden. Die Spezifikation von ES6 zwingen den Entwickler oder die Entwicklerin zu einer statischen Modul Struktur. In diesem Kapitel wird erläutert, was dies bedeutet und welche Vorteile dadurch entstehen. (Rauschmayer 2014)

Eine statische Modul Struktur ermöglicht durch Kompilieren des Codes, die darin befindlichen Importe und Exporte zu ermitteln. Es ist nicht nötig den Sourcecode auszuführen. In denm folgenden Listing 7 wird gezeigt warum dies bei CommonJS nicht möglich ist. (Rauschmayer 2014)

```
1  var mylib;
2  if (Math.random()) {
3      mylib = require('foo');
4  } else {
5      mylib = require('bar');
6  }
7
8  if (Math.random()) {
9      exports.baz = ...;
10 }
```

Listing 7: Flexible Struktur bei CommonJS

Bei der CommonJS Deklaration aus Listing 7 wird erst bei der Ausführung des Codes entschieden welcher Import beziehungsweise Export verwendet wird. Es6 Module besitzen nicht diese Flexibilität. Sie zwingen den Entwickler oder die Entwicklerin dazu, eine flache und statische Modul Struktur zu verwenden. Neben dem Verlust der Flexibilität bringt dies jedoch einige Vorteile mit sich. (Rauschmayer 2014)

### 3.3.4 Vorteile

#### 1. Schneller Lookup

CommonJS liefert beim Anfordern von abhängigen Modulen ein Objekt zurück. Beim Aufruf von zum Beispiel `lib.someFunc();` muss ein langsamer *Property Lookup* durchgeführt werden.

Im Gegensatz dazu sind die Inhalte bei dem Import eines ES6 Moduls bekannt. Der Zugriff auf die *Properties* kann daher optimiert werden. (Rauschmayer 2014)

#### 2. Variablen Überprüfung

Dank der statischen Modul Struktur sind die verfügbaren Variablen innerhalb eines Moduls immer bekannt. Dazu zählen:

- Globale Variablen

- Modul Importe
- Modul Variablen

Dies bietet eine große Hilfe für Entwickler und Entwicklerinnen. Durch eine statische Überprüfung mit einem *Linter* werden frühzeitige Tippfehler oder auch nicht verfügbare Variablen erkannt.(Rauschmayer 2014)

**3. Macro Support**

Wenn eine JavaScript Engine Macros unterstützt, können Entwickler und Entwicklerinnen neue Syntaxen der Sprache hinzufügen.

**4. Type Support**

Eine statische Typprüfung ist nur möglich, wenn die Typdefinitionen statisch gefunden werden. In weitere Folge können Typen aus Modulen importiert werden, wenn diese eine statische Struktur besitzen.

**5. Unterstützung anderer Sprachen**

Um das Kompilieren von Sprachen mit Macros und statischen Typen zu JavaScript zu ermöglichen, sollten die Module aus den zwei zuvor genannten Gründen eine statische Struktur aufweisen.

### 3.3.5 Schreibgeschützte Imports

Importe in CommonJS Modulen sind Kopien der exportierten Werte. Module Importe in ECMAScript 6 hingegen, sind schreibgeschützte Views auf die exportierten Entitäten. Dies bedeutet, dass Variablen die innerhalb eines Modules deklariert wurden, während der gesamten Programmlaufzeit bestehen bleiben. (Rauschmayer 2018)

```
1  //------ lib.js ------
2  export let counter = 3;
3  export function incCounter() {
4      counter++;
5  }
6
7  //------ main1.js ------
8  import { counter, incCounter } from './lib';
9
10 // The imported value `counter` is live
11 console.log(counter); // 3
12 incCounter();
13 console.log(counter); // 4
14
15 // The imported value can't be changed.
16 counter++; // TypeError
```

Listing 8: Importe sind Views auf exportierte Entitäten

In Listing 8 ist die importierte Variable `counter` schreibgeschützt. Sie kann jedoch durch den Aufruf der Funktion `incCounter` verändert werden.

In anderen Worten jeder Import ist eine Live-Verbindung zu den exportierten Daten. Rauschmayer (2018) zeigt die Unterschiede zwischen verschiedenen Importen in ES6:

- Unqualified imports wie zum Beispiel `import x from 'foo'` verhalten sich wie mit `const` deklarierte Variablen.
- Wird ein gesamtes Module Objekt importiert `import * as foo from 'foo'`, so verhält es sich wie ein mit `Object.freeze` geschütztes Objekt.

Gemäß ECMAScript (2015) erzeugt die Methode `CreateImportBinding(N, M, N2)` eine unveränderbare indirekte Bindung zu den Exporten eines anderen Moduls. Dabei wird überprüft, ob bereits ein Binding mit dem Namen **N** existiert. **M** ist die Bezeichnung des Moduls und **N2** ist der Name des von **M** exportierten Objekts. Im Weiteren ist zu beachten, dass es nicht möglich ist die Werte von Importen zu ändern, es können jedoch die Objekte geändert werden, auf welche verwiesen wird. (Rauschmayer 2018)

```
1 //------ lib.js ------
2 export let obj = {};
3
4 //------ main.js ------
5 import { obj } from './lib';
6
7 obj.prop = 123; // OK
8 obj = {}; // TypeError
```

Listing 9: Importe sind schreibgeschützt

Listing 9 zeigt, dass importierte Objekt wie `obj` nicht überschrieben werden können. Es ist jedoch möglich *Properties* hinzuzufügen oder zu verändern.

Exporte in ES6 werden über den Export Eintrag verwaltet. Bis auf reexportierte Module besitzten alle Einträgen die folgenden zwei Namen:

- Local Name: Ist der Name unter dem der Export im Module gespeichert ist.
- Export Name: Importierende Module verwenden diesen Namen, um Zugang zu dem Export zu erhalten. Nachdem Importieren einer Entität, wird diese stets mittels des Pointers erreicht. Der Pointer besteht dabei aus den zwei Teilen Modul und Local Name und referenziert auf ein Binding innerhalb des Moduls.

(Rauschmayer 2018)

Die nachfolgende angepasste Tabelle 1 aus ECMAScript (2015) gibt einen Überblick über die Local und Export Namen verschiedener Export Varianten:

Tabelle 1: Local und Export Namen verschiedene Export Varianten

| Export Statement Form | Export Name | Local Name |
|---|---|---|
| export var v; | v | v |
| export default function f(){}; | default | f |
| export default function(){}; | default | *default* |
| export default 42; | default | *default* |
| export {x}; | x | x |
| export {v as x}; | x | v |
| export {x} from ”mod”; | x | null |
| export {v as x} from ”mod”; | x | null |
| export * from ”mod”; | null | null |

# 4 Tree Shaking

*Tree Shaking* bedeutet im Javascript Umfeld Dead-Code Elimination. Der Begriff und das dahinterliegende Konzept wurde durch den *Static Bundler* RollupJS populär. Das Konzept des *Tree Shaking* beruht auf der statischen Struktur von ES6 Modulen. Diese werden im Zusammenhang mit *Bundlern* wie Webpack oder RollupJS in der Literatur auch als *Pure Modules* oder *Modules* oder *Harmony Modules* bezeichnet. (WebpackTreeShaking 2018)

Wie bereits in Kapitel 3 gezeigt wurde ermöglicht die ES6 Spezifikation, sowohl die statische Analyse von Modulen als auch die Verwendung einer Teilmenge der zur Verfügung gestellten Exporte. Für das nachfolgende Listing 10 wurde Webpack verwendet.

```
1  export function hello() {
2    return 'hello';
3  }
4  export function world() {
5    return 'world';
6  }
7
8  export default {
9    hello,
10   world
11 }
```

Listing 10: helpers.js

```
1  import {hello} from './helpers';
2
3  let elem = document.getElementById('output');
4  elem.innerHTML = `Output: ${hello()}`;
```

Listing 11: index.js

Listing 11 zeigt die Verwendung des zuvor Implementierten `helpers.js` Moduls. Dabei ist zu beachten, dass lediglich die Methode `hello` importiert und verwendet wird. `world` gilt somit als Dead Code oder *Unused Export* und wird in der gebündelten Anwendung nicht benötigt.

```
79 /* 1 */
80 /***/ (function(module, __webpack_exports__, __webpack_require__) {
81
82 "use strict";
83 /* harmony export (immutable) */ __webpack_exports__["a"] = hello;
84 /* unused harmony export world */
85 function hello() {
86   return 'hello';
87 }
88 function world() {
89   return 'world';
90 }
91
92 /* unused harmony default export */ var _unused_webpack_default_export = ({
93   hello: hello,
94   world: world
95 });
```

Listing 12: bundle.js

Um die unnötignen Codezeilen zu entfernen werden diese zuerst von Webpack markiert. In Listing 12 wurde sowohl der *Default Export* als auch die `world` Funktion mit `unused harmony` markiert.

Webpack entfernt den unerwünschten Code nicht, erst der darauf folgendn Minifier erkennt die `unused harmony` Annotationen und entfernt diese Segmente.

Die daraus resultierende *Bundle* Datei beinhaltet eine stark verkleinerte `hello` Funktion und die nicht benötigte Methode `world` wurde aus der Datei entfernt. Auch wenn in diesem Beispiel der Gewinn nicht groß ist, kann *Tree Shaking* dazu beitragen die *Bundle* Datei in größeren Projekten signifikant zu verkleinern. (WebpackTreeShaking 2018)

## 4.1 Webpack

Um *Tree Shaking* mit Webpack zu ermöglichen sind einige Konfigurationen nötig. Zuvor ist es wichtig einen genaueren Blick darauf zu werfen wie Webpack mit ES6 Modulen arbeitet.

Eine große Schwierigkeit bei der Verwendung von ES6 ist die Browser Inkompatibilität. Erstens dauert es eine gewisse Zeit bis die neuen Spezifikationen in den Browsern von den Herstellern implementiert werden können (CanIUse 2018b). Zweitens benutzt ein beachtlicher Teil der Clients nicht die aktuellste verfügbare Version (CanIUse 2018a).

Um das Problem mit der Inkompatibilität zu umgehen werden JavaScript Compiler wie Babel verwendet. Diese verwandeln Es6 in einen für die meisten Browser kompatiblen JavaScript Code. Somit können Entwickler und Entwicklerinnen bereits die neusten Funktionen verwenden, ohne dass diese von dem Großteil der Browser unterstützt werden muss.(Babel 2018)

Per Default wird von Babel jedes JavaScript Modul in CommonJS konvertiert. Durch die dynamische Implementierung von CommonJS Modulen können diese jedoch nicht statisch analysiert werden. Um ein erfolgreiches *Tree Shaking* zu ermöglichen, ist es daher entscheidend, dies zu ändern.(Babel 2018)

```
{
  entry: './index.js',
  output: {
    filename: 'dist/es6-modules/bundle.js'
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel-loader',
        query: {
          presets: [
            ['es2015', { modules: false }]
          ]
        }
      }
    ]
  }
}
});
```

Listing 13: webpack.config.js

Listing 13 zeigt eine typische *Tree Shaking* Konfiguration von Babel und Webpack. Besonders wichtig ist die `modules: false` Option von Babel. Mit dieser kann man die automatischen Modul Transformationen verändern. Neben dem `false` Wert, der diese komplett deaktiviert, ist es ebenfalls möglich andere Optionen anzugeben (`"amd"| "umd"| "systemjs"| "commonjs"`). (Babel 2018)

## 4.2 RollupJS

Neben Webpack wurde das *Tree Shaking* Konzept vor allem durch den *Module Bundler* RollupJS bekannt (WebpackTreeShaking 2018). Dieser wird vor allem empfohlen beim Entwickeln von Libaries. Webpack wiederum bietet eine bessere Unterstützung für Assets wie Bilder oder CSS und gilt daher als besseres geeignet für die Erstellung von Apps. (Rich Harris 2018)

Der Unterschied zwischen den *Bundlern* liegt in der Art und weise wie die einzelnen Module in der *Bundle* Datei verpackt sind. Webpack hüllt jedes Modul in eine Funktion mit einer browserfreundlichen Implementierung von `require` (Rich Harris 2018). Features wie On-demand Loading werden dadurch erst möglich. Nolan Lawson (2018) konnte jedoch zeigen das dies im weiteren auch dazu führt, dass je mehr Module verwendet werden auch der damit einhergehende Overhead wächst.

RollupJS verwendet im Gegensatz die Möglichkeiten von ES6. Der gesamte Code landet an einem Platz und wird in einem Durchgang verarbeitet. Damit wird der Code schlanker und kann schneller starten. (Rich Harris 2018)

## 4.3 Side Effects

In dieser Arbeit wird *Tree Shaking* aus der Perspektive zweier Entwicklergruppen betrachtet.

Zum einen jene Entwickler und Entwicklerinnen welche viel projektinternen JavaScript Code in ihren Web Projekten verwenden. Nielson (1997) zeigt den Zusammenhang zwischen der Ladezeit einer Seite und dem Benutzererlebnis. Das Ziel ist es folglich, eine oder mehrere möglichst kleiner *Bundle* Dateien an den Client auszuliefern, für schnelle Seitenaufrufe und eine bessere *User Experience*.

Die zweite Gruppe ist jene der Javascript Libary und Framework Entwickler. Ihr Sourcecode wird von vielen Projekten in verschiedener Art und Weise verwendet. In vielen Fällen wird in den Javascript Projekten nur ein kleiner Teil der importierten Libaries verwendet. Wenn diese *Tree Shaking* unterstützt, können jedoch *Bundler* wie Webpack, den nicht benötigten Code beim kompilieren der *Bundle* Dateien entfernen.

In diesem Kapitel werden meherere Hindernisse für die beiden Benutzergruppen im Bezug auf Side Effects im Code veranschaulicht und in weitere Folge Maßnahmen zu Vermeidung dieser ermittelt.

Umfangreiche und weitverbreitete JavaScript Libaries wie zum Beispiel Lodash[3] und jQuery[4] bieten eine große Anzahl an nützlichen Tools. Sie werden oft als NPM Modul installiert und anschließend in den Projektdateien als Import eingebunden. Zu meist werden nur Teile von den zu Verfügung gestellten Funktionalitäten auch wirklich verwendet. Das kann dazuführen das ein erheblich Teil der von Webpack erzeugten *Bundle* Dateien aus nicht benötigten Code besteht. Plugins wie der Webpack Visualizer[5] bieten eine einfache Möglichkeit, um festzustellen, welche Module in der von Webpack generierten *Bundle* Datei am meisten Speicherplatz einnehmen.

Jedoch bietet unter anderem Webpack eine Möglichkeit für Library Entwickler und Entwicklerinnen ES6 Module so zu implementieren, dass der nicht benötigte Library Sourcecode beim Builden entfernt wird. Durch die Verwendung von `sideEffects: false` in der package.json Datei innerhalb der Library, wird Webpack mitgeteilt das *Tree Shaking* nicht nur auf den üblichen Projektcode anzuwenden, sondern ebenfalls auf den Code der sich im Libary Ordner `node_modules` befindet.

Dabei werden Importdeklarationen wie `import {a, b} from "big-module-with-flag"` von Webpack erkannt und beim Bau der Applikation zu `import a from "big-module-with-flag/a"import b from "big-module-with-flag/b"` umgeschrieben. Somit werden in der *Bundle* Datei auch nur die zwei von dem Entwickler oder der Entwicklerin verwendeten Methoden

3. https://lodash.com/ - besucht am 26.05.2018
4. https://jquery.com/ - besucht am 26.05.2018
5. https://chrisbateman.github.io/webpack-visualizer/ - besucht am 12.07.2018

importiert und der restliche Libary Code entfernt (WebpackTreeShaking 2018). Zur Veranschaulichung der Umwandlung von Importen existiert ein von Webpack zur Verfügung gestelltes Beispiel[6].

Wenn mit Hilfe von `sideEffects: false` ein Modul als *pure* bezeichnet wird, werden alle nicht verwendeten Methoden, Klassen oder Objekte aus dem Code entfernt. Dies kann jedoch zur Entstehung von *Side Effects* führen und somit die Funktionsweise stören. Ein *Side Effect* ist als Code definiert, der beim Import ein anderes Verhalten ausführt als einen oder mehrere Exporte freizugeben. Ein Beispiel hierfür sind *Polyfills*, die den globalen Geltungsbereich beeinflussen und normalerweise keinen Export bereitstellen. Dies wird im Weiteren in Listing 14 und Listing 15 näher erläutert.

```
1  import a from 'a'
2    import b from 'b'
3
4    console.log("das ist ein side effect")
5
6    export {
7        a,
8        b
9    }
```

Listing 14: Libary mit Side Effects

```
1  import { a } from 'lib'
2
3    console.log("a:", a)
```

Listing 15: Verwendung der Libary

Listing 14 zeigt eine Beispiel für einen Libary *Entry Point*. Dieser exportiert alle Funktionalitäten die zur Verfügung gestellt werden.

Beim *Tree Shaking* werden alle Imports umgeschrieben. Dabei wird `import { a } from 'lib'` in Listing 15 zu `import a from 'lib/a'` und somit wird jeglicher Programm Code in lib.js nicht ausgeführt. Die `console.log` Nachricht in Zeile 4 wird somit nie angezeigt. Dies führt vorallem dann zu Problemen, wenn sich *Side Effects* auf Exporte in der Datei auswirken.

```
1  import { a } from 'lib'
2
3  a.value = 'this value is dangerous'
4
5     export {
6        a
7    }
```

Listing 16: Side Effect mit Auswirkung auf Exporte

In Listing 16 wird wie in Listing 14 `a` exportiert. Davor wird in der Zeile 3 die `importantValue` Eigenschaft hinzugefügt. Bei aktivem *Tree Shaking* wird `a` jedoch direkt mit `import a from 'lib/a'` importiert. `a.value` wird somit nie angelegt und kann nicht verwendet werden. Diese

6. https://github.com/webpack/webpack/tree/master/examples/side-effects - besucht am 26.05.2018

Unterschiede sind in spätere Folge für Benutzer und Benutzerinnen nur schwer nachzuvoll ziehen und Entwickler und Entwicklerinnen sollten bei der Erstellung einer Libary unbedingt diese vermeiden.(WebpackTreeShaking 2018)

Mit `sideEffects: false` betrachtet Webpack den gesamten Projektcode als *Pure Module*. Sind für die ordnungsgemäße Ausführung jedoch einige Importe nötig, können Entwickler und Entwicklerinnen diese als String Array angeben. Dieses akzeptiert relative Pfade , absolute Pfade und *Glob Pattern* zu den relevaten Dateien. Webpack verwendet Micromatch [7] zur Auflösung der Pfade.(WebpackTreeShaking 2018)

CSS Dateien sind ein gutes Beispiel für Importe die in jedem Fall in dem gebündelten Programmcode enthalten sein müssen. Bei der Verwendung des *CSS-Loader* und dem Importieren in einer Projektdatei, wird der CSS Code analysiert und die darin befindlichen Klassen stehen zur Verfügung. Sollten diese aber durch *Tree Shaking* entfernt werden, fehlen die CSS Klassen und die dazugehörigen Styles in der Anwendung. (WebpackTreeShaking 2018)

```
1  import './styles.css'
2
3  console.log('sideeffectful file')
4
5  export const sideeffect = 'sideeffect'
```

Listing 17: sideeffectful.js mit CSS Side Effect

```
1  .myClass {
2        background-color: blue;
3  }
```

Listing 18: styles.css

Listing 17 zeigt ein JavaScript Modul, welches die CSS Datei `styles.css` importiert. Mit der `sideEffects: false` Webpack Konfiguration wird `import './styles.css'` in Zeile 1 als *Unused Export* erkannt und ist in der gebündelten Anwendung nicht mehr vorhanden. `console.log('sideeffectful file')` und `export const sideeffect = 'sideeffect'` werden jedoch noch ausgeführt. Im Gegensatz dazu wird `sideEffects: ["*.css"]` in der Konfigurationsdatei verwendet und der CSS Import mit in die *Bundle* Datei übernommen. Auch wenn dies wegen dem Minify Schritt nicht sofort erkenntlich ist. Ist folgendes Code Segment darin zu finden `function(e,n,o){(e.exports=o(2)(!1)).push([e.i,".myClass {\n background-color: blue;\n}","",""])}`.(WebpackTreeShaking 2018)

### 4.3.1 Untersuchung von JavaScript Libraries

Es existieren bereits Projekte auf NPM die *Tree Shaking* verwenden. Durch eine Sourcecode Analyse sollen zusätzliche Erkenntnisse gewonnen werden, über die zum Einsatz kommenden Design Patterns und Möglichkeiten die Entwickler und Entwicklerinnen mit *Linting* bei der Implementierung dieser zu unterstützen. In weiterer Folge werden am Ende dieser Arbeit die

7. https://github.com/micromatch/micromatch - besucht am 21.06.2018

ausgewählten Projekte mit dem ESLint *Tree Shaking Plugin* auf Fehler und Warnungen getestet. Dabei sollten keine Fehler angezeigt werden, um die richtige Funktionsweise des Plugins sicherzustellen.

**Auswahlkriterien**

Die wichtigsten Kriterien für die zur Analyse ausgewählten Projekte ist zum einen die Verwendung eines *Static Bundlers* und zum anderen eine bereits bestehende Implementierung von *Tree Shaking*. Um dies festzustellen, wird zuerst geprüft ob eine package.json Datei im Repository existiert.

Anschließend wird in dieser nach `sideEffects: false` gesucht. Wie bereits im Kapitel 4.3 veranschaulicht, dient die `sideEffects` Einstellung dazu, dem *Bundler* anzuzeigen, dass nur die verwendeten Exporte in dem gebuildeten Programmcode benötigt werden. Die ausgewählten JavaScript Repositories sollten für eine möglichst große Anzahl an Entwicklern und Entwicklerinnen von relevanter Bedeutung sein. Für die Auswahl werden daher jene NPM Module verwendet, welche am häufigsten in Webprojekten verwendet werden. Es existiert bereits eine generische Liste [8] die diese Module beinhaltet.

Auf NPM befinden sich neben Bibliotheken welche ES6 verwenden, auch welche die für NodeJS entwicklert wurden. Libraries welche mit NodeJS Modulen implementiert wurden sind für diese Arbeit nicht geeignet (NodeJSModules 2018). Ein weiteres Kriterium für die Auswahl der zu untersuchenden Projekte ist daher die Verwendung von ES6 Modulen. Anhand dieser Auswahlkriterien wurden folgende drei Projekte ermittelt, welche *Tree Shaking* und ES6 implementieren:

- lodash[9]
- vue[10]
- graphql[11]

Anforderungen für *Tree Shaking* werden mit einer Sourcecode Analyse ermittelt und die daraus gewonnen Erkenntnisse im nachfolgenden Kapitel 4.3.2 dokumentiert.

**Implementierung von *Tree Shaking***

In den ausgewählten Repositories wurde *Tree Shaking* mit `sideEffects: false` in der pakage.json Datei aktiviert. Somit werden bei der Verwendung *Named Imports* wie zum Beispiel `import {add} from 'lodash'`, diese aufgelöst zu `import add from 'lodash/add'`. In weiterer Folge wird sämtlicher nicht genutzter Programmcode der Library entfernt und nur die verwendeten Module in die *Bundle* Datei integriert.

Webpack startet mit der Erzeugung des *Bundles* bei dem definierten *Entry Point*. In den untersuchten Repositories dient diese Datei, um die gewünschten Funktionalitäten nach außen

8. https://www.npmjs.com/browse/depended - besucht am 06.06.2018
9. https://github.com/lodash/lodash/tree/es - besucht am 15.06.2018
10. https://github.com/vuejs/vue - besucht am 15.06.2018
11. https://github.com/graphql/graphql-js - besucht am 15.06.2018

zu exportieren. Wie bereits im Kapitel 4.3 festgestellt wurde, können mögliche *Side Effects* in dieser Datei entfernt werden. In allen drei Fällen werden daher lediglich `import` und `export` verwendet.

Da Programmcode außerhalb des benötigten Moduls nicht garantiert in der *Bundle* Datei enthalten ist, wurde der gesamte Sourcecode in für sich eigenständige Module aufgeteilt. Dies bedeudet, dass Module wie zum Beispiel `add.js` alle Resourcen, welche für die ordnungsgemäße Funktionalität benötigt werden, importieren müssen.

### 4.3.2 Best Practices zur Vermeidung von Side Effects

Anhand der untersuchten Eigenschaften und Libraries wurden die folgenden Anforderungen in Bezug auf Vermeidung von *Side Effects* ermittelt, diese werden bei der Implementierung berücksichtigt und am Ende diese Arbeit auf ihre Wirksamkeit überprüft.

- Besitzt ein Modul *Named Exports* ist darauf zu achten, dass auf diese keine *Side Effects* wirken.
- Bei der Verwendung eines *Default Exports* wird die gesamte Datei eingebunden. Jegliche *Side Effects* innerhalb der Datei werden somit inkludiert.
- Importe welche den Body eines Moduls ausführen werden von Rauschmayer (2018) als *Empty Imports* bezeichnet, können beim bündlen entfernt werden und müssen deshalb als *Side Effect* in der `package.json` Datei vermerkt werden.
- Der *Entry Point* einer Library sollte für sich eigenständige Module importieren und diese als *Named Exports* zur Verfügung stellen. Dieser sollte keine *Side Effects* beinhalten.

# 5 Linting

Der Begriff *Lint* wurde erstmals in den 1970er Jahren in Verbindung mit der Softwareentwicklung erwähnt. Johnson (1978) entwicklete es an den Bell Laboratories um die Schwächen der damaligen C Compiler auszugleichen und unentdeckte Fehler in bereits kompilierten Programmen zu finden.

Es existieren viele Wege um die Anzahl an Bugs innerhalb einer Anwendung zu minimieren. Neben dem Schreiben von Unit Tests bieten Code Reviews laut Louridas (2006) wohl die beste Möglichkeit Bugs und Code Smells aufzufinden und zu eliminieren. Es fordert jedoch einen großen Zeitaufwand mehrere Entwickler zusammenzubringen und die gesamte Codebasis einer Anwendung gemeinsam zu reviewen.

Die meisten Fehler fallen in die Kategorie Known Errors. Dies sind häufig auftretende sich wiederholende Fälle in denen die Entwickler und Entwicklerinnen immer wieder hineinstoplern. Mit *Linting* möchte man die sich ständig wiederholenden Fehler oder auch Code Smells so früh wie möglich finden und ausbessern. (Louridas 2006)

Im Gegensatz zu einem *Compiler* muss beim *Linting* der Code nicht ausgeführt werden. So genannte Static Checker durchlaufen den Programmcode auf der Suche nach bestimmten Mustern, dieser Prozess wird als Statische Code Analyse bezeichnet. (Louridas 2006)

Es gibt verschiedene Wege, um eine statische Code Analyse durchzuführen. Wie zum Beispiel auf Anfrage eines Entwicklers oder einer Entwicklerin, kontinuierlich während der Erstellung einer Anwendung oder aber direkt bevor ein Entiwckler oder eine Entwicklerin Codeänderungen auf ein Repository comitten will. Je nach Projekt können somit Fehler entdeckt und behoben werden bevor diese in der Codebasis am Repository landen.(Johnson u. a. 2013)

Es exisitiert bereits Literatur zur statischen Code Analyse. Dabei wurden nicht nur Aspekte wie Performance und Genauigkeit untersucht (Bessey u. a. 2010), sondern auch verschiedene Einsatzgebiete gezeigt (Bush, Pincus und Sielaff 2000).

Johnson u. a. (2013) erforschten in ihrer Arbeit den Gebrauch und die Verbreitung von statische Analyse Tools. Ein Teil davon war die Ermittlung, was Entwickler und Entwicklerinnen bei dem Gebrauch dieser Werkzeuge als störend empfinden. Ergebnis davon war der nicht zu unterschätzende Anteil an false postives, das heißt die Anzahl an Warnungen, die keine wirklichen Fehler sind. Im Weiteren wurde auch noch die hohe Arbeitslast von Entwicklern und Entwicklerinnen genannt. Laut Johnson u. a. (2013) sollten zukünftige Tools, sowohl eine bessere Integration in den Arbeitsabläufen von Entwicklern und Entwicklerinnen bieten, als auch die Arbeit in einem Team besser unterstützen.

Bush, Pincus und Sielaff (2000) implementierte ein statisches Analyse Tool zum Auffinden von dynamischen Fehlern in C und C++. Dazu zählen unter anderem die Verwendung von nicht intitialisierten Speicher oder falschen Operation auf Dateien (zum Beispiel das Schließen einer Datei die bereits geschlossen wurde). Für das aus der Arbeit entstandene Tool wurde die Bezeichnung *PREFix* ausgewählt. Zur Analyse von Code wird der *Abstract Syntax Tree* generiert und anschließend die Funktionsaufrufe mittels eines topolischen Algorithmus sortiert. Die Funktionen werden daraufhin simuliert und darin befindliche Defekte gemeldet. Als Teil dieser Arbeit wird ein Plugin erstellt, welches wie bei Bush, Pincus und Sielaff (2000) den *Abstract Syntax Tree* auf ähnliche Art und Weise untersucht. Jedoch soll dies für die Programmiersprache JavaScript erfolgen und sich lediglich auf fehlerhafte Muster bei der Verwendung von ES6 Imports und Exports fokussieren.

## 5.1 ESLint

2013 erschuf der Entwickler Nicholas C. Zakas ESLint ein erweiterbares *Linting* Tool für JavaScript und JSX. Neben vielen vorhandenen *Linting* Regeln bietet ESLint eine Möglichkeit für Entwickler und Entwicklerinnen solche auch selbst zu implementieren. Durch Hinzufügen oder Entfernen von Regeln wird der *Linting* Prozess je nach Projekt und Team konfiguriert. Dabei gelten für jede Regel die folgenden drei Grundsätze:

- Jede Regel sollte eigenständig funktionieren
- Jede Regel bietet eine Möglichkeit zum An und Abschalten (nicht darf als zu wichtig zum deaktivieren gelten)

- Es kann zwischen Warnung und Fehler gewechselt werden

Um nicht jede Regel einzeln zu importieren gibt es ebenfalls die Möglichkeit mehrere Regeln als *Bundle* auszuliefern. (ESLintAbout 2018)

Ziel dieser Arbeit ist es, Anforderrungen für die Optimierung von ES6 Exports und Imports zu identifizieren. Des Weiteren werden Regeln für ESLint erstellt, welche schlechte Muster im Programmcode aufdecken und Lösungen dafür anzeigen. (ESLintAbout 2018)

## 5.2 Core Rules

ESLint wird bereits mit mehr als, 200 allgemein gültigen Regeln ausgeliefert. Diese werden als Core Regeln bezeichnet. Gemäß (ESLintNewRules 2018), muss eine eigene Regel die folgenden sechs Eigenschaften erfüllen, um diese der Core Liste hinzuzufügen:

**Allgemeine Anwendbarkeit:** Core Regeln sollten relevant für eine große Anzahl an Entwicklern und Entwicklerinnen sein. Regeln für individuelle Präferenzen oder Edge Cases werden nicht akzeptiert.

**Generisch:** Neue Regeln sollten möglichst generisch sein. Ein Benutzer oder eine Benutzerin darf keine Schwierigkeiten haben zu verstehen, wann eine Regel zu benutzen ist. Als Richtlinie gilt, dass eine Regel zu spezifisch ist, wenn man zur Beschreibung ihrer Funktion mehr als zwei “und” benötigt. (zum Beispiel: Wenn a und b und c und d, dann wird der Benutzer oder die Benutzerin gewarnt)

**Atomar:** Regeln sollten vollkommen selbstständig funktionieren. Abhängikeiten zwischen zwei oder mehreren Regeln sind nicht erlaubt.

**Einzigartig:** Überschneidung zwischen Regeln sind ebenfalls nicht erlaubt. Jede sollte eine eigene Warnung erzeugen. Somit wird der Benutzer nicht unnötig verwirrt.

**Unabhängig:** Regeln dürfen nur auf der JavaScript Laufzeitumgebung basieren und müssen unabhängig von Libaries oder Frameworks sein. Core Regeln sollten stehts anwendbar sein und nicht davon abhängen, ob spezielle Libaries wie JQuery verwendet werden. Hingegen existieren bereits einige Regeln, die nur angewendet werden, wenn die Laufzeitumgebung NodeJS vorhanden ist.

**Ohne Konflikte:** Keine Regel sollte sich mit anderen Regeln überschneiden. Zum Beispiel gibt es eine Regel, welche von einem Entwickler oder einer Entwicklerin verlangt, jedes Statement mit einem Semikolon zu beenden. Es ist nicht erlaubt eine neue Regel zu entwerfen, die Semikolons verbietet. Stattdessen existiert die Regel Semi, welche beides erlaubt und mit Hilfe der Konfiguration gesteuert werden kann.

Das in dieser Arbeit entstehende Plugin und die dazugehörigen Regeln soll Entwickler und Entwicklerinnen beim Umgang mit *Tree Shaking* und in Zusammenhang mit *Static Module Bundler* wie Webpack unterstützen. Die allgemeine Anwendbarkeit der Regeln ist somit nicht gegeben und diese können folglich auch nicht als Core Regeln eingereicht werden.

## 5.3 Erzeugung und Traversierung des Abstract Syntax Tree

Ein *Abstract Syntax Tree (AST)* erfasst die wesentliche Struktur des Programmcodes, mit Verzicht auf unnötige syntaktische Details in der Form eines Baumes. Dabei werden die unterschiedlichen Konstrukte der Programmiersprache als *Node Types* representiert. (Jones 2003)

ESLint verwendet für die Erstellung des AST einen eigenen Parser Namens Espree[12] (ESLintEspree 2018). Dieser wurde auf Basis des JavaScript Parsers Acorn[13] gebaut. Die modulare Architektur von Acorn ermöglicht eine einfache Erweiterung von Kernfunktionalitäten. Der von Espree erzeugte AST und dessen *Nodes* entspricht der von ESTree[14] spezifizierten Syntax.

Jede ESLint Regel implementiert eine `create` Funktion. Diese retourniert ein Objekt mit Methoden. Diese werden von ESlint aufgerufen um *Nodes* zu besuchen während der Traversierung durch den *Abstract Syntax Tree*. Jede dieser *Visitor Functions* wird mit einem *Key* oder auch *Selector* definiert. Sollte dieser einem *Node Type* entsprechen, so wird die *Visitor Function* auf die *Node* aufgerufen. (ESLintRules 2018)

# 6 Erweiterte Analyse durch die Erstellung eines ESLint Plugins

## 6.1 Setup

Um die Projektstruktur aufzusetzen wird die von ESLintNewRules (2018) empfohlene Vorgehensweise verwendet. Wichtige Voraussetzungen um ein ESLint Plugin zu erstellen sind NPM und NodeJS, diese müssen zuerst installiert werden. In dieser Arbeit wird NodeJS v8.0.0 und NPM 5.8.0 verwendet.

Des weiteren wird das Commandline Tool Yeoman benötigt. Die aktuellste Version ist 2.0.2, diese wird mit dem Befehl `npm install -g yo` global installiert und steht somit in der Konsole an jedem Ort zur Verfügung. Zuletzt wird das NPM Modul eslint-generator installiert es dient dazu ein leeres Gerüst für den Plugin Code zu erstellen.

Nach der Installation wird mit dem Befehl `yo eslint:plugin` der Generator gestartet. Folgende Daten wurden dabei angegeben:

- **Name:** silltho
- **Plugin-Id:** threeshaking
- **Beschreibung:** Webpack Tree Shaking Support
- **Werden Benutzerdefinierte Regeln verwendet?:** Ja

12. https://github.com/eslint/espree - besucht am 12.07.2018
13. https://github.com/acornjs/acorn - besucht am 12.07.2018
14. https://github.com/estree/estree - besucht am 12.07.2018

- **Werden Processors verwendet?:** Nein

```
{
  "name": "eslint-plugin-treeshaking",
  "version": "0.0.0",
  "description": "Webpack Tree Shaking Support",
  "keywords": [
    "eslint",
    "eslintplugin",
    "eslint-plugin"
  ],
  "author": "sillth0",
  "main": "lib/index.js",
  "scripts": {
    "test": "mocha tests --recursive"
  },
  "dependencies": {
    "requireindex": "~1.1.0"
  },
  "devDependencies": {
    "eslint": "~3.9.1",
    "mocha": "^3.1.2"
  },
  "engines": {
    "node": ">=0.10.0"
  },
  "license": "ISC"
}
```

Listing 19: generierte package.json Datei

Listing /reflisting:package.json zeigt die daraus resultierende package.json Datei.

In weiterer Folge wurden neben der `README.md` Datei auch ein `lib` und `tests` Ordner im Projektverzeichnis erstellt. Der Ordner `lib/rules` wird später den Programmcode für die neu erstellten Regeln beinhalten.

## 6.2 Entwicklungsumgebung

Für die Programmierung wird die integrierte Entwicklungsumgebung (IDE) WebStorm von JetBrains in der Version 2018.1.2 verwendet. Zum Zeitpunkt dieser Arbeit ist die aktuellste verfügbare ESLint Version 4.19.1. In weitere Folge wird Prettier[15] für die Code Formatierung genutzt. Anschließend wurde noch Husky[16] verwendet, um *Git Hooks* für Prettier und die Test Suite zu registrieren.

15. https://prettier.io/ - besucht am 04.07.2018
16. https://github.com/typicode/husky - besucht am 04.07.2018

## 6.3 Implementierung der Plugin Regeln

Yeoman in Verbindung mit dem ESLint-Generator unterstützt Entwickler und Entwicklerinnen nicht nur beim Erzeugen der Projektdateien sondern ebenfalls bei dem Anlegen neuer Regeln innerhalb des Plugins. Die dafür nötigen Dateien werden mit dem Befehl `yo eslint:rule` erzeugt. Die neu erstellten Regeln innerhalb des Plugins sollen die Aufmerksamkeit von Entwickler und Entwicklerinnen auf die Probleme aus Kapitel 4.3.2 richten und Wege aufzeigen um diese zu vermeiden. In diesem Kapitel wird dokumentiert, wie die Regeln erstellt wurden.

Beim Aufruf des Generators mit `yo eslint:rule` müssen mehrere Fragen beantwortet werden:

- What is your name?
- Where will this rule be published?
- What is the rule ID?
- Type a short description of this rule
- Type a short example of the code that will fail

In dieser Arbeit wird als Name “silltho” verwendet. Als den Publishing Ort kann entweder ESLint Core oder ESLint Plugin gewählt werden. Wie bereits im Kapitel 5.2 beschrieben wurde, eignen sich die in dieser Arbeit erstellten Regeln nicht als Core Regeln, es wird somit bei allen Regeln die Option ESLint Plugin gewählt.

Alle weiteren Fragen können je nach Regel unterschiedlich beantwortet werden. In weiterer Folge werden diese in den Tabellen 2, 3 und 4 bei der Implementierung jeder Regel dargestellt.

Vom Generator werden abschließend die folgenden Dateien im Projektverzeichnis erzeugt:

- **docs/rules/(rule-id).md** - Dokumentation der Regel und der Konfiguration Möglichkeiten.
- **lib/rules/(rule-id).js** - Programmcode.
- **tests/lib/rules/(rule-id).js** - Tests um sicherzustellen das die Regel wie gewünscht funktioniert.

Die nachfolgenden Kapitel beschreiben die Implementierung der Regeln die benötigt werden um Code Smells beim *Tree Shaking* für Entwickler und Entwicklerinnen anzuzeigen.

### 6.3.1 Empty Imports

Tabelle 2: Yeoman Generator: Empty Imports

| | |
|---|---|
| Rule-ID: | no-empty-imports |
| Beschreibung: | disallow Empty Imports |

Gemäß den Erkenntnissen aus Kapitel 4.3 müssen *Empty Imports* wie zum Beispiel `import 'module1'` als *Side Effect* gekennzeichnet werden. Diese werden sonst beim Bündeln des Programmcodes als *Unused Imports* erkannt und entfernt. Ziel dieser Regel ist es die Aufmerksamkeit von Entwicklern und Entwicklerinnen auf diese Art von Imports zu richten und sie daran zu erinnern, diese in der `package.json` Datei als *Side Effect* zu vermerken.

Um diese Funktionsweise zu gewährleisten, wurde folgende Test Datei `test/lib/rules/empty-imports.js` erstellt:

```
1  var ruleTester = new RuleTester();
2  ruleTester.run("empty-imports", rule, {
3
4      valid: [
5        {
6          options: [['empty-import']],
7          code: `
8            import 'empty-import'
9            import lib1 from 'lib1'
10           export const export1 = 'export1'
11           export default 'default'
12         `,
13         errors: []
14       },
15       {
16         options: [],
17         code: `
18           import name from "module-name";
19           import * as name2 from "module-name";
20           import { member } from "module-name";
21           import { member as alias } from "module-name";
22           import { member1 , member2 } from "module-name";
23           import { member3 , member2 as alias2 } from "module-name";
24           import defaultMember, { member4 } from "module-name";
25           import defaultMember2, * as alias3 from "module-name";
26           import defaultMember3 from "module-name";
27         `,
28         errors: []
29       }
30     ],
31
32     invalid: [
33         {
34             code: `
35                 import 'empty-import'
36                 import lib1 from 'lib1'
37                 export const export1 = 'export1'
38                 export default 'default'
39             `,
40             errors: [{
41                 message: 'empty imports are removed by tree shaking. Make
                       sure you add empty-import to package.json sideEffects
                       option.',
42                 type: 'ImportDeclaration'
```

```
            }]
        }
    ]
});
```

Listing 20: Empty Imports Unit Tests

Im ersten Schritt werden mit dem AstExplorer [17] die verschiedenen Import *Nodes* analysiert. Der *Node Type* für Importe in ES6 ist `ImportDeclaration`. In weiterer Folge konnte festgestellt werden, dass *Empty Import Nodes* keine *Specifier* besitzen, somit ist `importNode.specifiers.length === 0` true, falls es sich um einen *Empty Import* handelt. Mit Hilfe dieser Erkenntnisse ist es bereits möglich *Empty Imports* aufzuspüren und zu kennzeichnen.

```
        function isEmptyImport(importNode) {
            if(importNode.specifiers.length === 0) {
              reportEmptyImport(importNode)
            }
        }

        return {
          'ImportDeclaration': isEmptyImport
        };
```

Listing 21: Code zum Aufspüren von Empty Imports

Nach der Implementierung von Listing 21 werden zwei der drei Tests bereits erfolgreich durchgeführt. Beim ersten validen Test wird jedoch der Fehler `AssertionError [ERR_ASSERTION]: Should have no errors but had 1` angezeigt. *Empty Imports* können in der `package.json` Datei als *Side Effects* vermerkt werden. Diese werden anschließend im Programmcode inkludiert und können verwendet werden. Die *Empty Imports* Regel sollte dahingehend konfigurierbar sein.

Hierfür sollte man als Option ein Array von Pfaden und *Glob Pattern* übergeben gleich zu der *Side Effects* Einstellung von Webpack. WebpackTreeShaking (2018) zeigt, dass für die Auflösung der `sideEffects` *Property* die Bibliothek Micromatch [18] verwendet wird. Für die Auflösung der Array Daten innerhalb der Regel wird dieselbe Library verwendet, um die Unterschiede zwischen Webpack und der Regel Konfiguration möglichst gering zu halten.

Damit die *Empty Imports* Regel ein Array von Strings als *Property* akzeptiert, muss die Schema Konfiguration in `lib/rules/empty-imports.js` wie folgt angepasst werden:

```
    schema: [
      {
        type: 'array'
      }]
  }
```

Listing 22: Empty Import Option Schema

17. https://astexplorer.net/ - besucht am 26.06.2018
18. https://github.com/micromatch/micromatch - besucht am 26.06.2018

Anschließend kann auf die Konfiguration mittels `context.options[0]` zugegriffen werden.

```
1         function isKnownSideEffect(emptyImportNode) {
2           return mm.any(emptyImportNode.source.value, knownSideEffects,
                null)
3         }
```

Listing 23: isKnownSideEffect Funktion

Listing 24 beinhaltet die Implementierung der `isKnownSideEffect` Funktion. Diese vergleicht den Value eines Imports mit dem übergebenen Array. Micromatch bietet dafür die Methode `any (str, patterns, options)` an. Diese vergleicht den übergebenen String mit dem Patternsarray und liefert true falls ein beliebiges Pattern in dem Array mit dem String übereinstimmt.

Zuletzt muss noch die `isEmptyImport` Methode angepasst werden, um mit der neuen Funktion `isKnownSideEffect` alle *Empty Imports* zu überprüfen.

```
1         function isEmptyImport(importNode) {
2             if(importNode.specifiers.length === 0 && !isKnownSideEffect(
                  importNode)) {
3               reportEmptyImport(importNode)
4             }
5         }
```

Listing 24: isEmptyImport Funktion

Listing 24 veranschaulicht die Anpassungen an der `isEmptyImport` Funktion. Beim erneuten Ausführen der Tests werden alle erfolgreich abgeschloßen. Die Regel ist somit erfolgreich Implementiert und wird im Kapitel 6.4 auf ihre Wirksamkeit geprüft.

### 6.3.2 Named Exports Side Effects

Tabelle 3: Yeoman Generator: Named Exports Side Effects

| Rule-ID: | no-named-exports-sideeffects |
|---|---|
| Beschreibung: | disallow Side Effects onto Named Exports |

Ein weiterer Code Smell aus Kapitel 4.3 ist der Gebrauch von *Side Effects* auf *Named Exports*.

```
1  let counter = 1
2
3  const incCounter = (newValue) => {
4    counter++
5  }
6
7  export {
8    incCounter,
9    counter
10 }
```

Listing 25: counter.js

```
import { counter, incCounter } from "counter";
incCounter();
export { counter }
```

Listing 26: lib.js

```
import { counter } from "lib";
console.log(counter) // 1
```

Listing 27: index.js

Listing 25, 26 und 27 zeigen ein Beispiel für einen *Side Effect*. Obwohl die Methode `incCounter` in `lib.js` aufgerufen wird, wird durch *Tree Shaking* `import { counter } from "lib"` zu `import { counter } from "counter"`. Somit wird die Datei `lib.js` nicht inkludiert und auch der `incCounter` Aufruf geht in weiterer Folge verloren.

Für die Implementierung werden *Imports* (`import`), *Named Exports* (`export`) und *Expressions* (`incCounter` oder `counter.test = '123'`) benötigt. Mit dem ASTexplorer wurden erneut die dazugehörigen *Node Types* `ImportDeclaration`, `ExportNamedDeclaration` und `ExpressionStatement` ermittelt.

Die folgende Testdatei soll die Funktionsfähigkeit der Regel sicherstellen. Insgesamt wurden 4 valide und 4 invalide Szenarien ausgearbeitet. Diese können im anschließenden Listing 28 betrachtet werden.

```
var ruleTester = new RuleTester()
ruleTester.run('named-exports-sideeffects', rule, {
  valid: [
    {
      options: [],
      code: `
          import name from "module-name";
          import name2 from "module-name2";
          name2.sideeffect = 'sideeffect'
          export {
            name
          }
          export default name2
        `,
      errors: []
    },
    {
      options: [],
      code: `
          testFunction = () => {
            console.log('test')
          }
          export {
            testFunction
          }
        `,
      errors: []
    },
```

```
    {
      options: [],
      code: `
          import name from "module-name";
          import name2 from "module-name2";
          console.log('test123')
          export default name2
        `,
      errors: []
    },
    {
      options: [],
      code: `
          import name from "module-name";
          import name2 from "module-name2";
          const tmp = 'tmp'
          export {
            tmp as name,
            name as name2
          }
        `,
      errors: []
    }
  ],

  invalid: [
    {
      options: [],
      code: `
            import test2 from "module-name2";
            test2.sideeffect = 'sideeffect'
            export { test2 }
    `,
      errors: [
        {
          message:
            'Effects on reexported modules (test2) could be prune by
               TreeShaking.',
          type: 'ExpressionStatement'
        }
      ]
    },
    {
      options: [],
      code: `
          import test2 from "module-name2";
          test2.init('initSomeThing')
          export { test2 }
        `,
      errors: [
        {
          message:
```

```
80              'Effects on reexported modules (test2) could be prune by
                    TreeShaking.',
81            type: 'ExpressionStatement'
82          }
83        ]
84      },
85      {
86        options: [],
87        code: `
88            import { counter as temp1, incCounter as temp2 } from "module-
                  name2";
89            temp2();
90            export { temp1 as tmp }
91          `,
92        errors: [
93          {
94            message:
95              'Effects on reexported modules (temp1) could be prune by
                    TreeShaking.',
96            type: 'ExpressionStatement'
97          }
98        ]
99      },
100     {
101       options: [],
102       code: `
103           import { counter, incCounter } from "module-name2";
104           incCounter();
105           export { counter }
106         `,
107       errors: [
108         {
109           message:
110             'Effects on reexported modules (counter) could be prune by
                    TreeShaking.',
111           type: 'ExpressionStatement'
112         }
113       ]
114     }
115   ]
116 })
```

Listing 28: Named Exports Side Effects Unit Tests

Die invaliden Testfälle aus Listing 28 können in zwei Kategorien eingeteilt werden. Die erste Kategorie bilden direkte *Side Effects* wie zum Beispiel `test2.init('initSomeThing')`. Dieser wirkt direkt auf das zuvor importierte `test2` Modul, welches anschließend als *Named Export* exportiert wird. Die zweite Kategorie in dieser Arbeit bilden indirekte *Side Effects*. Ein Beipiel dafür ist der Aufruf der Methode `incCounter()` in Listing 26. Dieser hat Auswirkungen auf den Export von `counter`. Da die Variable `counter` nie verwendet oder verändert wird, sind indirekte *Side Effects* wesentlich schwerer zu erkennen als direkte *Side Effects*.

Für die Erkennung der direkten *Side Effects* werden alle Imports und *Named Exports* in einem Array gespeichert. Dies geschieht mit der Hilfe des `ImportDeclaration` und `ExportNamedDeclaration` *Node Selectors*. Diese rufen die Funktionen `saveImport` und `saveNamedExport` auf. Darin werden die einzelnen *Specifier* in das Array gespeichert. Zum Beispiel bei dem Export `export {test1, test2}` werden jeweils die *Specifier Node* `test1` und `test2` gespeichert. Im nächsten Schritt werden alle *Side Effects* mit dem `ExpressionStatement` *Selector* und der Funktion `saveExpression` ebenfalls in einem Array abgelegt. Schließlich kann mit dem *Selector* `Program:exit` am Ende eines jeden Moduls der Identifier jeder gespeicherten *Expression* mit den *Named Exports* und *Imports* verglichen werden und bei Überschneidungen diese *Expression Node* als *Side Effect* melden. Somit werden *Side Effects* die direkt einen *Named Export* betreffen erkannt und wird dem Benutzer oder der Benutzerin eine entsprechende Warnung angezeigt.

Das Erkennen von indirekten *Side Effects* gestaltet sich komplizierter. Zuerst werden alle *Named Imports* wie zum Beispiel `import {counter, incCounter} from 'counter'` ebenfalls in einem Array gespeichert. Der dafür verwendete *Node Selector* ist `ImportDeclaration`. Indirekte *Side Effects* können entstehen wenn Importe mehrere *Specifier* besitzen. Beispiele dafür sind die Import *Specifier* aus Listing 26 `counter` und `incCounter`.Der Import deklariert zwei Variablen aus dem selben Modul. Wenn eine der *Expressions* mit einem der Einträge in `import` übereinstimmt und das Modul einen im selben Import befindlichen *Specifier* als *Named Export* exportiert, wird davon ausgegangen, dass es sich dabei um einen *Side Effect* handelt. Die entsprechende *Expression Node* wird somit gemeldet und der Benutzer erhält einen Hinweis, dass dies zu ungewollten Effekten in seinem Programm führen kann.

Indirekte *Side Effects* wie `incCounter()` aus Listing 26 werden zusätzlich zu den direkten *Side Effects* erkannt und alle Tests werden erfolgreich abgeschlossen.

### 6.3.3 Entry Point Side Effects

Tabelle 4: Yeoman Generator: Entry Point Side Effects

| Rule-ID: | no-entry-point-sideeffects |
|---|---|
| Beschreibung: | disallow Side Effects in the Entry Point |

Die Startdatei oder *Entry Point* einer JavaScript Libary ist ein wichtiger Bestandteil für die Effektivität von *Tree Shaking*. Listing 29 zeigt die zum Testen der Regeln implementierten Tests.

```
1 var ruleTester = new RuleTester()
2 ruleTester.run('entry-point-sideeffects', rule, {
3   valid: [
4     {
5       options: [],
6       filename: 'test/usr/src/entry.js',
7       code: `
8         import module from 'module'
```

```
        console.log('sideeffect')
        export const export1 = 'export1'
        export default 'default'
      `,
      errors: []
    },
    {
      options: [],
      filename: 'test/usr/src/entry.js',
      code: `
        import module from 'module'
        module.test = 'tmp'
        export const export1 = 'export1'
        export default 'default'
      `,
      errors: []
    },
    {
      options: [],
      filename: 'test/usr/src/entry.js',
      code: `
        import module from 'module'
        console.log('test')
        export const export1 = 'export1'
        export default 'default'
      `,
      errors: []
    }
  ],

  invalid: [
    {
      options: ['**/src/entry.js'],
      filename: 'test/usr/src/entry.js',
      code: `
        import module from 'module'
        module.test = 'tmp'
        export const export1 = 'export1'
        export default 'default'
      `,
      errors: [
        {
          message: 'Sideeffects in the entry-point are not allowed.',
          type: 'ExpressionStatement'
        }
      ]
    },
    {
      options: ['**/src/entry.js'],
      filename: 'test/usr/src/entry.js',
      code: `
        import module from 'module'
```

```
61          console.log('test')
62          export const export1 = 'export1'
63          export default 'default'
64        `,
65        errors: [
66          {
67            message: 'Sideeffects in the entry-point are not allowed.',
68            type: 'ExpressionStatement'
69          }
70        ]
71      }
72    ]
73  })
```

Listing 29: Entry Point Side Effects Unit Tests

Die Regel soll für jede *Node* im *Entry Point*, bei der es sich weder um einen Import noch einen Export handelt, dem Entwickler und der Entwicklerin eine Meldung anzeigen. Für die Implementierung werden die *Node Types* `ImportDeclaration`, `ExportNamedDeclaration`, `ExportDefaultDeclaration`, `ExportAllDeclaration` und `Program` benötigt. In weiterer Folge wird zusätzlich der `:not` *Selector* verwendet. Eine Beschreibung dazu lieferte der ESLint Selector Guide[19]. Der *Selector* um alle potenziellen *Side Effects* in einem Modul zu finden ist: `Program > :not(ImportDeclaration, ExportNamedDeclaration, ExportDefaultDeclaration, ExportAllDeclaration)`.

Anschließend muss noch überprüft werden, ob es sich bei dem aktuellen Modul um den *Entry Point* handelt. Der Benutzer oder die Benutzerin kann dies durch eine Option steuern. Die *Entry Point Side Effects Regel* erwartet als Parameter einen *Glob Pattern String*. Dieser dient dazu den *Entry Point* festzulegen. Nur dieser soll absolut frei von *Side Effects* sein und darf lediglich `import` und `export` Deklarationen beinhalten.

## 6.4 Testen des Plugins

Um die Funktionsweise des Plugins `eslint-plugin-treeshaking` zu überprüfen, wird eine Testreihe mit JavaScript Repositories von Github[20] durchgeführt. Als Test Subjekte werden sowohl Bibliotheken welche *Tree Shaking* unterstützen, als auch Bibliotheken, die nicht darauf optimiert wurden, benötigt. In dem Kapitel 4.3.1 werden drei Repositories mit der `sideEffects: false` Konfiguration ermittelt. Diese werden für die Testreihe erneut verwendet. Die drei nicht in Bezug auf *Tree Shaking* optimierte Bibliotheken werden der selben Liste[21] entnommen.

### *Tree Shaking* optimiert:

19. https://eslint.org/docs/developer-guide/selectors - besucht am 29.06.2018
20. https://github.com/ - besucht am 07.07.2018
21. https://www.npmjs.com/browse/depended - besucht am 06.06.2018

- graphql-js[22]
- vue[23]
- lodash[24]

**kein *Tree Shaking***

- d3[25]
- immutable-js[26]
- moment[27]

### 6.4.1 Aufbau

Für die Durchführung der Tests wird von allen Testsubjekten mit dem `git clone` Befehl eine lokale Arbeitskopie angelegt. In weitere Folge wird ESLint benötigt. In allen Repositories muss eine statische Code Analyse mit dem *Tree Shaking Plugin* durchgeführt werden. Am besten ist dies mit dem Tool ESLint-CLI[28] möglich. Dieses ermöglicht es, die Tests direkt mittels Konsoleneingabe zu starten und wird mit dem Befehl `npm install -g eslint-cli` global installiert.

```
{
  "env": {
    "browser": true,
    "es6": true
  },
  "parserOptions": {
    "sourceType": "module"
  },
  "plugins": ["treeshaking"],
  "rules": {
    "treeshaking/no-empty-imports": ["warn",["*.css"]],
    "treeshaking/no-named-exports-sideeffects": "warn",
    "treeshaking/no-entry-point-sideeffects": ["warn", "index.js"]
  }
}
```

Listing 30: Beispiel für eine Test Konfiguration

22. https://github.com/graphql/graphql-js - besucht am 15.06.2018
23. https://github.com/vuejs/vue - besucht am 15.06.2018
24. https://github.com/lodash/lodash - besucht am 15.06.2018
25. https://github.com/d3/d3 - besucht am 08.07.2018
26. https://github.com/facebook/immutable-js - besucht am 08.07.2018
27. https://github.com/moment/moment - besucht am 15.06.2018
28. https://github.com/eslint/eslint-cli - besucht am 09.07.2018

Für die Konfiguration der Analyse wird eine `.eslintrc` Datei verwendet. Listing 30 zeigt ein Beispiel für diese Datei. Zwischen den verschiedenen Repositories ist es nötig die Optionen bei den drei aktiven Regeln anzupassen.

Eine weitere Abhängigkeit ist das *Tree Shaking Plugin*[29]. Dieses wird durch den Befehl `npm install git+ssh://git@github.com/silltho/eslint-plugin-treeshaking.git` installiert. Auch ESLint muss als Projektinteres Modul zur Verfügung stehen. Der Befehl `npm install eslint` wird also ebenfalls in jedem Repository ausgeführt.

Sollte eine Libary bereits ESLint verwenden, werden alle Abhängigkeiten mit `npm install` installiert. Anschließend wird eine Analyse ohne jegliche Anpassungen durchgeführt. Dadurch werden bereits bekannte Warnungen oder Probleme ermittelt und können von den Ergebnissen ausgeschlossen werden. Zuletzt wird die bereits bestehende Konfiguration um das *Tree Shaking Plugin* und die dazugehörigen drei Regeln erweitert und eine erneute Analyse gestartet.

### 6.4.2 Testablauf

Für den Ablauf wird zuerst der *Entry Point* der zu untersuchenden Bibliothek ermittelt. Der Einstiegspunkt für das Programm wird mit der Option `main` in der `package.json` Datei gesetzt. Dieser wird als Option für die Regel `treeshaking/no-entry-point-sideeffects` benötigt.

Anschließend wird die `.eslintrc` Datei, wie in Listing 30 ersichtlich, erstellt und die Optionen für das jeweilige Repository angepasst.

Darüber hinaus muss ermittelt werden, wo sich die Source Dateien der Bibliotheken befinden. Für den ESLint Befehl wird ein *Glob Pattern* benötigt, welches den gesamten Source Code abdeckt. Schließlich kann die Analyse mit dem Befehl `eslint` durchgeführt werden. Für die Ausgabe der Berichte als HTML Datei werden die Optionen `-o lint.html` und `-f html` verwendet.

Sowohl die projektspezifischen ESLint Optionen, als auch der der Befehl zum durchführen der Tests werden als Tabelle bei den Ergebnissen mit präsentiert.

### 6.4.3 Ergebnisse

#### graphql-js

Tabelle 5: ESLint Optionen für graphql-js

| | |
|---|---|
| no-empty-imports | [”warn”,[]] |
| no-named-exports-sideeffects | ”warn” |
| no-entry-point-sideeffects | [”warn”, ”**/index.js”] |

29. https://github.com/silltho/eslint-plugin-treeshaking

**Default Lint Befehl:**

`eslint src/**/*.js -o lint_default.html -f html --rulesdir ./resources/lint`

**Lint Befehl:**

`eslint src/**/*.js -o lint.html -f html --rulesdir ./resources/lint`

GraphQL verwendet bereits ESLint und es existiert im Repository eine `.eslintrc` Datei. Durch eine Analyse der in `package.json` befindlichen Konfiguration, wird das Script `"lint": "eslint --rulesdir ./resources/lint src || (printf '\\033[33mTry: \\033[7m npm run lint -- --fix \\033[0m\\n' && exit 1)"` zur Durchführung einer statischen Analyse verwendet. Die Option `--rulesdir ./resources/lint` definiert repositoryspezifische Regeln und muss für den Test adaptiert werden, um Fehler zu vermeiden. Die Konfiguration wurde in den Default Lint Befehl übernommen. Die daraus resultierende HTML Datei zeigt, dass sich keine bekannten Fehler im Repository befinden. Anschließend wurde das *Tree Shaking Plugin* mit den in der Tabelle 5 ersichtlichen Regeln hinzugefügt. Hier wurde wie erhofft keine Warnung oder Fehler angezeigt. Aus der Sicht des Plugins unterstützt die GraphQl Bibliothek wie erwartet *Tree Shaking*.

**vue**

Tabelle 6: ESLint Optionen für vue

| | |
|---|---|
| no-empty-imports | [”warn”,[]] |
| no-named-exports-sideeffects | ”warn” |
| no-entry-point-sideeffects | [”warn”, ”**/src/core/index.js”] |

**Default Lint Befehl:** `eslint src/**/*.js -o lint_default.html -f html`

**Lint Befehl:** `eslint src/**/*.js -o lint.html -f html`

Im Vue Repository befindet sich bereits eine `.eslintrc`. Somit muss zuerst ein Lint Bericht ohne dem *Tree Shaking Plugin* erstellt werden. Anders wie bei GraphQl gibt es keine projektspezifische Konfiguration oder Regeln. Für die Erstellung des Prüfberichts musste der `eslint` Befehl nicht angepasst werden und **`default`**`.html` zeigte keine projektinternen Fehler. Darauf folgend wurde das *Tree Shaking Plugin* der Konfiguration hinzugefügt, die Optionen wie in Tabelle 6 ersichtlich gesetzt und ein neuer Bericht erstellt.

`lint.html` zeigt insgesamt 5 Warnungen in der *Entry Point* Datei `src/core/index.js`. Alle zeigen dieselbe Nachricht “Side effects in the entry-point are not allowed.”.

```
1 import Vue from './instance/index'
2 import { initGlobalAPI } from './global-api/index'
3 import { isServerRendering } from 'core/util/env'
4 import { FunctionalRenderContext } from 'core/vdom/create-functional-
    component'
5
6 initGlobalAPI(Vue)
7
8 Object.defineProperty(Vue.prototype, '$isServer', {
```

```
9     get: isServerRendering
10  })
11
12  Object.defineProperty(Vue.prototype, '$ssrContext', {
13    get () {
14      /* istanbul ignore next */
15      return this.$vnode && this.$vnode.ssrContext
16    }
17  })
18
19  // expose FunctionalRenderContext for ssr runtime helper installation
20  Object.defineProperty(Vue, 'FunctionalRenderContext', {
21    value: FunctionalRenderContext
22  })
23
24  Vue.version = '__VERSION__'
25
26  export default Vue
```

Listing 31: Vue Entry Point Datei (src/core/index.js)

Listing 31 zeigt die *Entry Point* Datei der Vue Libary. Diese besitzt, wie das *Tree Shaking Plugin* anzeigt, mehrere *Side Effects*. Wenn man den Code betrachtet wird ersichtlich, dass ein *Default Export* und keine *Named Exports* verwendet wird. *Side Effects* werden somit immer inkludiert und stellen keine Gefahr für die Funktionweise des exportierten Objekts dar. Dadurch is es nicht möglich, nur einzelne Teile der Vue Bibliothek zu verwenden. Folgende Fragen bleiben offen: Sollte eine Warnung angezeigt werden, wenn der *Entry Point* nur einen *Default Export* anbietet? Sind *Side Effects* im *Entry Point* erlaubt, wenn nur ein *Default Export* verwendet wird? Weitere Forschung wäre nötig, um diese Fragen zu klären und das Plugin dahingehend zu erweitern.

**lodash**

Tabelle 7: ESLint Optionen für lodash

| | |
|---|---|
| no-empty-imports | [”warn”,[]] |
| no-named-exports-sideeffects | ”warn” |
| no-entry-point-sideeffects | [”warn”, ”lodash.js”] |

**Default Lint Befehl:** `eslint *.js -o lint_default.html -f html`
**Lint Befehl:** `eslint *.js -o lint.html -f html`

Lodash verwendet ebenso wie GraphQL und Vue ESLint. Beim Erstellen des Prüfberichts werden drei bereits bekannte Fehler angezeigt.
`Unable to resolve path to module './.internal/createFind.js'`. ist ein Problem beim Auflösen von Dateipfade. Dies wird vorerst ignoriert und wieder wird das *Tree Shaking Plugin* hinzugefügt und die Optionen dem Repository angepasst. Bei der erneuten Durchführung der statischen Analyse werden dieselben drei Fehler wie im Default Bericht angezeigt. Es konnten keine Warnungen in Bezug auf *Tree Shaking* gefunden werden.

### d3

Tabelle 8: ESLint Optionen für d3

| | |
|---|---|
| no-empty-imports | [”warn”,[]] |
| no-named-exports-sideeffects | ”warn” |
| no-entry-point-sideeffects | [”warn”, ïndex.js”] |

**Default Lint Befehl:** `eslint *.js -o lint_default.html -f html`
**Lint Befehl:** `eslint *.js -o lint.html -f html`

Die d3 Libary ist das erste Testsubjekt, welches nicht die `sideEffects` Option in der `package.json` Datei verwendet. *Tree Shaking* wurde somit nicht aktiviert und unbenützter Code kann nicht entfernt werden. Das *Tree Shaking Plugin* soll Entwicklern und Entwicklerinnen dabei helfen dies zu ändern und Stellen aufzudecken, bei denen Anpassungen nötig sind. d3 verwendet kein ESLint und besitzt auch keine `.eslintrc` Datei. Deshalb wird für den Default Bericht diese Datei erstellt. Listing 32 zeigt den Inhalt der neuen `.eslintrc` Datei.

```
{
  "env": {
    "browser": true,
    "es6": true
  },
  "parserOptions": {
    "sourceType": "module"
  },
  "plugins": [],
  "rules": {}
}
```

Listing 32: Default Konfiguration für die d3 Bibliothek

Der erstellte Bericht zeigt keine Fehler oder Warnungen im Repository. Nach dem Hinzufügen des *Tree Shaking Plugins* und seiner Konfiguration wird ein erneuter *Lint* Bericht `lint.html` erstellt. Dieser zeigt ebenso keine Fehler oder Warnungen. Damit sollte die d3 Bibliothek bereits *Tree Shaking* unterstützen. Es fehlt lediglich die `sideEffects: false` Einstellung in der `package.json` Datei.

### immutable-js

Tabelle 9: ESLint Optionen für immutable-js

| | |
|---|---|
| no-empty-imports | [”warn”,[]] |
| no-named-exports-sideeffects | ”warn” |
| no-entry-point-sideeffects | [”warn”, ßrc/Immutable.js”] |

**Default Lint Befehl:** `eslint src/**/*.js -o lint_default.html -f html`
**Lint Befehl:** `eslint src/**/*.js -o lint.html -f html`

Wie andere Test Subjekten verwendet die Immutable-JS Bibliothek bereits ESLint und besitzt eine Konfigurations Datei. Nachdem Installieren der dafür benötigten Abhängigkeiten wird ein erster *Lint* Bericht, ohne die Verwendung des *Tree Shaking Plugins*, erstellt. Dabei treten weder Warnungen noch Probleme auf. Beim *Linting* mit dem *Tree Shaking Plugin* werden sechs Warnungen in der Date `src/CollectionImpl` gefunden. Alle sind vom Typ no-named-exports-sideeffects. Der von der Regel generierte Hinweis lautet: “Effects on reexported modules (Collection) could be prune by TreeShaking”.

```
1  import {
2    Collection,
3    KeyedCollection,
4    IndexedCollection,
5    SetCollection,
6  } from './Collection';
7
8  // Note: all of these methods are deprecated.
9  Collection.isIterable = isCollection;
10 Collection.isKeyed = isKeyed;
11 Collection.isIndexed = isIndexed;
12 Collection.isAssociative = isAssociative;
13 Collection.isOrdered = isOrdered;
14
15 Collection.Iterator = Iterator;
16
17 export {
18     Collection,
19     KeyedCollection,
20     IndexedCollection,
21     SetCollection,
22     CollectionPrototype,
23     IndexedCollectionPrototype,
24 };
```

Listing 33: immutable-js Warnungen (src/CollectionImpl)

Der Programmcode in Listing 33 zeigt die von ESLint markierten Zeilen und die dazugehörigen Import und Export Deklarationen. Bei der Verwendung von *Tree Shaking* könnten die Zeilen 9 bis 15 unbeabsichtigt entfernt werden. Weitere Nachforschungen sind nötig, um genauer festzustellen ob die vom Plugin, markierten Zeilen auch wirklich zu Fehlern bei der Verwendung von `Collection` führen.

**moment**

Tabelle 10: ESLint Optionen für moment

| | |
|---|---|
| no-empty-imports | [”warn”,[]] |
| no-named-exports-sideeffects | ”warn” |
| no-entry-point-sideeffects | [”warn”, ßrc/moment.js”] |

**Default Lint Befehl:** `eslint src/**/*.js -o lint_default.html -f html`
**Lint Befehl:** `eslint src/**/*.js -o lint.html -f html`

moment-js ist die letzte Javascript Bliobliothek die für die Überprüfung des *Tree Shaking Plugins* untersucht wird. Ebenso wie d3 und immutable-js ermöglicht sie kein *Tree Shaking*. Für die Entwicklung wurde nicht ESLint verwendet. Somit wird eine `.eslintrc` Datei ohne aktivierte Regeln erstellt und eine Analyse damit durchgeführt. Der resultierende Bericht zeigt keine Fehler in den Source Dateien. Nach dem Hinzufügen der *Tree Shaking Regeln* wie in Tabelle 10 gezeigt, werden im Bericht insgesamt 19 Warnungen angezeigt. Diese betreffen vier Dateien und sind vom Typ no-empty-imports.

Die vier betroffenen Dateien sind:

- src/lib/duration/duration.js
- src/lib/locale/en.js
- src/lib/locale/locale.js
- src/lib/units/units.js

```
1  import {
2    Collection,
3    KeyedCollection,
4    IndexedCollection,
5    SetCollection,
6  } from './Collection';
7
8  // Side effect imports
9  import './prototype';
10
11 import { createDuration } from './create';
12 import { isDuration } from './constructor';
13 import {
14       getSetRelativeTimeRounding,
15       getSetRelativeTimeThreshold
16 } from './humanize';
17
18 export {
19       createDuration,
20       isDuration,
21       getSetRelativeTimeRounding,
22       getSetRelativeTimeThreshold
23 };
```

Listing 34: moment Datei mit Warnungen (src/lib/duration/duration.js)

Listing 34 zeigt in Zeile 9 einen *Empty Import*, der die Warnung in dieser Datei verursacht. Er wurde bereits vorab von einem Entwickler oder einer Entwicklerin als *Side Effect* makiert. Die genauen Fehlermeldungen des *Tree Shaking Plugins* für diese Zeile lauten:“empty imports are removed by tree shaking. Make sure you add ./prototype to package.json sideEffects option”. Durch die Verwendung eines Arrays von Strings statt false für die `sideEffects` Konfiguration, kann Webpack informiert werden, diese Datei auf jeden Fall einzubinden. Die dafür nötige

Einstellung wäre `sideEffects: ['prototype']`. Die no-empty-imports Regel bietet an, dies durch eine Option zu konfigurieren. Es wird daher der Eintrag für no-empty-imports in der `.eslintrc` Datei zu `"treeshaking/no-empty-imports": ["warn", ["prototype"]]` angepasst. Ein erneuter Bericht wird erstellt und dieser zeigt die Warnung in der Datei aus Listing 34 nicht an. Wiederum wären weitere Forschungen nötig, um die richtige Funktionsweise von *Tree Shaking* an dieser Stelle genauer zu evaluieren.

# 7 Ausblick auf zukünftige Arbeiten

Als Teil dieser Arbeit wurden im Kapitel 4.3.2 vier Annahmen zur Vermeidung von *Side Effects* getroffen. Diese konnten durch eine Literatur Analyse und Unteruchung von GitHub Repositories ausgearbeitet werden. Nachfolgende Forschung mit mehr Untersuchungsobjekten und einer größeren Testreihe könnte weitere Erkenntnisse zur Vermeidung von *Side Effects* liefern. Dies kann zur Implementierung weiterer Regeln oder einer Anpassung der bereits existierenden Regeln führen.

Ein weiterer Aspekt ist die Erweiterung des *Tree Shaking Plugins* auf weitere ECMAScript Versionen. Diese Arbeit konzentriert sich auf die Verwendung von ECMAScript 2015. Mit ECMAScript 2016[30] und ECMAScript 2017[31] existieren bereits nachfolgende Spezifikationen. Vielmehr wäre auch die Verwendung verschiedener *Bundler* weitere Untersuchungen wert. RollupJS ist eine Alternative zu Webpack. Auf dessen Verwendung spezialisiert sich diese Arbeit.

Zuletzt könnten zukünftige Arbeiten das Erstellen von *Fixes* für die Regeln des *Tree Shaking Plugins* untersuchen. *Fixes* bieten die Möglichkeit, Fehler oder Warnungen automatisiert zu reparieren, ohne Eingreifen eines Entwicklers oder einer Entwicklerin. In weiterer Folge sind *Fixes* ein wichtiges Feature, um Entwickler und Entwicklerinnen zu entlasten. Im Zusammenhang mit *Tree Shaking* und dem in dieser Arbeit erstellten Plugin, wäre zuerst zu ntersuchen ob, eine automatisierte Lösung möglich ist und wie diese zu implementieren wäre.

# 8 Schlussfolgerung

Ziel dieser Arbeit ist es, für Entwickler und Entwicklerinnen im JavaScript Bereich ein Werkzeug anzubieten, welches dabei hilft ihren Sourcecode dahingehend zu gestalten, um *Tree Shaking* zu ermöglichen. Dies führt zu einer Reduktion der an die Benutzer und Benutzerinnen gesendeten Daten, verringert die Ladezeit und hat somit einen positiv Einfluss auf die *User Experience*.

Um *Tree Shaking* von JavaScript zu ermöglichen ist ein *Static Module Bundler* wie Webpack und die Verwendung der ECMAScript 6 Spezifikationen nötig. Im Gegensatz zu CommonJS

30. https://www.ecma-international.org/ecma-262/7.0/ - besucht am 13.07.2018
31. https://www.ecma-international.org/ecma-262/8.0/ - besucht am 13.07.2018

oder der Asynchronous Model Definition bietet die statische Struktur von ES6 Modulen die Möglichkeit, diese mit einer statischen Analyse zu untersuchen, bevor sie von einem *Bundler* evaluiert und in einer Datei kombiniert werden.

Die Herausforderungen beim *Tree Shaking* wurden durch eine Literatur Recherche und der Untersuchung von Projekten auf GitHub, welche dieses bereits implementieren ermittelt. Daraus resultierte die Erkenntnis, dass für eine fehlerlose Funktionweise von *Tree Shaking* sogenannte *Side Effects* ein wichtiger Faktor sind. Im Zuge dessen wurden mehrere Best Practices und Code Smells für den Umgang mit *Side Effects* ermittelt:

- Bei der Vewendung von *Named Exports* ist auf *Side Effects* im Modul zu achten.
- *Side Effects* wirken sich nicht auf den *Default Export* aus.
- Alle *Empty Imports* müssen dem *Bundler* mit Hilfe der `sideEffects` *Property* mitgeteilt werden.
- Der *Entry Point* eines Projektes sollte keine *Side Effects* implementieren.

Im Anschluss wurden diese Informationen verwendet um ein Plugin für ESLint zu erstellen. ESLint ermöglicht die statische Analyse von JavaScript Code und bietet durch Plugins eine einfache und gut dokumentierte Lösung um diese zu erweitern. In weiterer Folge wurde das Plugin *eslint-plugin-treeshaking* erstellt und drei Regeln zur Vermeidung von Code Smells implementiert.

Abschließend wurden sechs Projekte auf GitHub ausgewählt, um die Funktionweise des Plugins zu überprüfen. Drei davon wurden bereits auf *Tree Shaking* optimiert und bei der Analyse der Bibliothek Vue konnte eine Schwierigkeit in Bezug auf *Default Exports* erkannt werden. Bei den beiden anderen Repositories konnten keine Fehler oder Warnungen vom *Tree Shaking Plugin* ermittelt werden.

Weitere wichtige Erkenntnisse konnten durch die Überprüfung der drei Repositories erlangt werden, welche kein *Tree Shaking* unterstützen. Die JavaScript Libary d3 besitzt eine sehr modulare Architektur und kann *Tree Shaking* ohne Anpassungen unterstützen. Bei den beiden anderen Test Subjekten konnten Probleme ermittelt werden, welche zu unerwünschten Nebeneffekten durch *Tree Shaking* führen können.

Die durchgeführten Tests bestätigen den positiven Nutzen der statischen Analyse in Verbindung mit dem *Tree Shaking Plugins* für Entwickler und Entwicklerinnen. Es werden Code Segmente die beim *Tree Shaking* zu Problemen führen können angezeigt. Die sofortige Rückmeldung erleichtert das Lösen von Fehlern wärend der Entwicklung. Vielmehr kann das Plugin auch dazu beitragen nachträglich JavaScript Code zu optimieren, um *Tree Shaking* zu ermöglichen.

# Abkürzungsverzeichnis

| | |
|---|---|
| AMD | Asynchronous Model Definition |
| AST | Abstract Syntax Tree |
| ES6 | ECMAScript 6 |

# Abbildungsverzeichnis

# Listings

# Tabellenverzeichnis

# Literaturverzeichnis

Babel. 2018. *Babel · The compiler for writing next generation JavaScript.* Besucht am 22. März. `http://babeljs.io/`.

Bessey, Al, Dawson Engler, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky und Scott McPeak. 2010. »A few billion lines of code later«. *Communications of the ACM* 53, Nr. 2 (Februar): 66–75. ISSN: 00010782. doi:`10.1145/1646353.1646374`.

Bush, William, Jonathan Pincus und David Sielaff. 2000. »A static analyzer for finding dynamic programming errors«. *Software: Practice and Experience* 30 (7).

CanIUse. 2018a. *Browser Usage Table.* Besucht am 22. März. `https://caniuse.com/usage-table`.

———. 2018b. *Support tables for ES6.* Besucht am 22. März. `https://caniuse.com/%7B%5C#%7Dsearch=es6`.

Dominik Wilkowski. 2018. *npm and the front end.* Besucht am 11. März. `https://medium.com/dailyjs/npm-and-the-front-end-950c79fc22ce`.

ECMAScript. 2015. *ECMAScript 2015 Language Specification.* Besucht am 22. Januar 2018. `http://www.ecma-international.org/ecma-262/6.0/`.

ESLintAbout. 2018. *About - ESLint - Pluggable JavaScript linter.* Besucht am 29. März. `https://eslint.org/docs/about/`.

ESLintEspree. 2018. *Introducing Espree, an Esprima alternative - ESLint - Pluggable JavaScript linter.* Besucht am 12. Juli. `https://eslint.org/blog/2014/12/espree-esprima`.

ESLintNewRules. 2018. *New Rules - ESLint - Pluggable JavaScript linter.* Besucht am 4. April. `https://eslint.org/docs/developer-guide/contributing/new-rules`.

ESLintRules. 2018. *Working with Rules - ESLint - Pluggable JavaScript linter.* Besucht am 12. Juli. `https://eslint.org/docs/developer-guide/working-with-rules`.

Johnson, B, Y Song, E Murphy-Hill und R Bowdidge. 2013. »Why Don't Software Developers Use Static Analysis Tools to Find Bugs?« In *35th International Conference on Software Engineering (ICSE).*

Johnson, SC. 1978. »Lint, a C program checker«. *Computer Science Technical Report* 65.

Jones, Joel. 2003. »Abstract Syntax Tree Implementation Idioms«. In *Proceedings of the 10th conference on pattern languages of programs (plop2003).*

Louridas, P. 2006. »Static code analysis«. *IEEE Software* 23:58–61. doi:10.1109/MS.2006.114.

Nielson, Jakob. 1997. *The Need for Speed.* Besucht am 3. Juli 2018. https://www.nngroup.com/articles/the-need-for-speed/.

———. 2010. *Website Response Times.* Besucht am 3. Juli 2018. https://www.nngroup.com/articles/website-response-times/.

NodeJSModules. 2018. *Modules — Node.js v10.4.1 Documentation.* Besucht am 15. Juni. https://nodejs.org/api/modules.html.

Nolan Lawson. 2018. *The cost of small modules — Read the Tea Leaves.* Besucht am 25. März. https://nolanlawson.com/2016/08/15/the-cost-of-small-modules/.

NpmDocs. 2018. *What is npm?* Besucht am 11. März. https://docs.npmjs.com/getting-started/what-is-npm.

Rauschmayer, Axel. 2014. *ECMAScript 6 modules: the final syntax.* Besucht am 13. März 2018. http://2ality.com/2014/09/es6-modules-final.html.

———. 2018. *Exploring ES6.* Besucht am 25. Juni 2018. http://exploringjs.com/es6/index.html.

Rich Harris. 2018. *Webpack and Rollup: the same but different – webpack – Medium.* Besucht am 25. März. https://medium.com/webpack/webpack-and-rollup-the-same-but-different-a41ad427058c.

RollupJSDocs. 2018. *RollupJS.* Besucht am 12. März. https://rollupjs.org/guide/en.

Sebastian Peyrott. 2018. *JavaScript Module Systems Showdown: CommonJS vs AMD vs ES2015.* Besucht am 12. März. https://auth0.com/blog/javascript-module-systems-showdown/.

WebpackComparison. 2018. *Comparison.* Besucht am 3. Juli. https://webpack.js.org/comparison/.

WebpackConcepts. 2018. *Concepts.* Besucht am 12. März. https://webpack.js.org/concepts/.

WebpackTreeShaking. 2018. *Webpack - Tree Shaking.* Besucht am 20. März. https://webpack.js.org/guides/tree-shaking/.

# Appendix

## A git-Repository

https://github.com/silltho/bachelor-thesis-2