HTML5-Tutorium: JavaScript: Hello World 05

Wechseln zu:Navigation, Suche

Vorlesung WebProg

Inhalt | Teil 1 | Teil 2 | Teil 3 | Teil 4 | Teil 5 | Teil 6

Musterlösung: `index.html` (5a), `index.html` (5b), `index.html` (5c), `index.html` (5d), (Git-Repositories: 5a, 5b, 5c, 5d)

Inhaltsverzeichnis

1 Anwendungsfälle (Use Cases)

Gegenüber dem dritten Teil des Tutoriums ändern sich die die Anwendungsfälle nicht. Die Anwendung leistet also genau dasselbe wie zuvor.

In diesem Teil des Tutoriums geht es darum, die Anwendung mittels Webpack[1] automatisch für den Server-Betrieb zu optimieren, sodass möglichst wenige, möglichst kleine Dateien zum Client übertragen werden.

Im Tutorium wird „Webpack 5“ verwendet.

2 Erstellen eines neuen Projektes

Erstellen Sie ein neues Projekt `HelloWorld05` als Fork von `HelloWorld04` (siehe Hello-World-02-Tutorium).

3 JavaScript Module Bundlers

Um Module einer Web-Anwendung für die Auslieferung zum Client zu komprimieren, zu bündeln und anderweitig zu optimieren, gibt es zahlreiche Werkzeuge. Für Node.js gibt es die sogenannten “JavaScript Module Bundlers”, wie z. B. Webpack[1], Browserify[2] oder FuseBox[3].

Üblicherweise optimieren die Bundlers nicht nur JavaScript-Dateien, sondern auch HTML-, CSS-, JSON- und andere Dateien.

Webpack ist ein sehr mächtiges Werkzeug, mit dem eine Vielzahl von Web-Dateien in einer oder einigen wenigen JavaScript-Dateien zusammengefasst und bei Bedarf auch komprimiert werden können. Dieser Bundler kommt in vielen Projekten zum Einsatz. Einer der Entwickler von webpack ist Johannes Ewald, ein ehemaliger Student der HSA (vgl. https://github.com/jhnns). Er bietet auch regelmäßig Vorlesung an der HSA zum Thema „JavaScript“ an.

Allerdings hat Webpack auch einige Nachteile:

> Es gibt diverse zugehörige Node.js-Pakete ganz unterschiedlicher Qualität. Für eine Aufgabe das beste oder zumindest ein gutes Paket zu finden, ist häufig eine Sisyphos-Arbeit.
> Mit jedem Versionssprung der Hauptrevisionsnummer haben sich bislang die Schnittstellen und Prinzipien so gravierend geändert, dass man die Webpack-Konfiguration im Wesentlichen neu erstellen musste.
> Auch kann es einem passieren, dass ein Paket, das bislang gewinnbringend verwendet wurde, plötzlich nicht mehr unterstützt wird und aufgrund von neu entdeckten Sicherheitsproblemen auch nicht mehr benutzt werden kann.

Dennoch werden wir im Tutorium Webpack 5 einsetzen. Die Vorteile überwiegen trotz aller Nachteile.

4 Webpack

4.1 Node.js-Pakete

Um Webpack verwenden zu können, müssen Sie zunächst ein Node.js-Projekt anlegen.

```
npm init -y
```

Tagen Sie wie im Tutorium HTML5-Tutorium: JavaScript: Entwicklungsumgebung: Node.js beschrieben geeignete Werte in die JSON-Datei `package.json` ein (`name`, `version`, `description`, `main`, `repository`, `keywords`, `author` und `license`). Achten Sie darauf, dass Visual Studio Code keine Fehler meldet.

Installieren Sie nun die benötigten Node.js-Pakete:

```
npm i -D webpack webpack-cli
npm i -D copy-webpack-plugin
npm i -D css-loader style-loader file-loader extract-loader
```

Das Paket `webpack` enthält das eigentlich Webpack-System. `webpack-cli` enthält Kommandozeilenbefehle, mit denen Webpack von einer Shell aus oder auch mittels NPM-Skripts gesteuert werden kann.

Das Copy-Webpack-Plugin[4] dient dazu, mittels Webpack Dateien von einem Ordner in einen anderen zu kopieren. Verschiedene Loaders werden verwendet, um CSS- und JavaScript-Dateien für den „Bündelung“ vorzubereiten.[5]

4.2 Git commit

Achtung: Bevor Sie einen Commit durchführen, müssen Sie im selben Ordner, in dem sich die Datei `package.json` befindet, eine Datei namens **`.gitignore`** anlegen. (Der Punkt zu Beginn des Dateinamens darf nicht fehlen!)

In diese Datei schreiben Sie ein einziges Wort: `node_modules`

`.gitignore`

```
node_modules
```

Diese Datei ist **extrem wichtig**. Wie Sie bereits im Node.js-Tutorium gesehen haben, enthält der Ordner `node_modules` sehr schnell **sehr viele** Dateien (mehrere Zehntausend). Wenn Sie diese im Repository speichern, bläht sich dieses extrem stark auf. Die Git-Push- und -Pull-Befehle werden dadurch sehr langsam.

Die Datei `.gitignore` dient dazu, dies zu verhindern. In jeder Zeile steht ein Dateiname oder ein Dateipattern (wie zum Beispiel `*.log`, das alle Dateien mit der Dateiextension `log` beschreibt), um zu verhindern, dass die zugehörigen Dateien ins Git-Repository eingefügt werden.

Wenn Sie dafür sorgen, die Dateien `package.json` und `package-lock.json` ins Repository eingefügt werden, kann ein anderer Benutzer, der Ihr Git-Repository auf seinen Rechner lädt, die fehlenden Node.js-Module jederzeit ganz einfach restaurieren:

```
npm i
```

Fügen Sie nun alle Dateien (mit Ausnahme des Ordners `node_modules` ins Repository ein (zum Beispiels mittels `git add -u`) und committen Sie die Änderungen.

4.3 Webpack

Musterlösung: `index.html` (5a) (Git-Repository: 5a)

Als nächstes Webpack konfiguriert werden, damit die Dateien `web/index.html`, `web/css/head.css` und `web/js/mains.js` automatisch generiert werden können.

Benennen sie zunächst den Ordner `web` ins `src` um. Im Ordner `src` erstellen Sie künftig neue Dateien, nehmen Änderungen vor etc. Dabei sollten Sie unbedingt darauf, dass sinnvolle Module angelegt werden, viele sinnvolle Kommentare vorhanden sind, die Blöcke mittels Leerzeichen sauber eingerückt sind etc. **Verwenden Sie niemals Tab-Zeichen zum Einrücken, da jeder Text-Editor Tab-Zeichen anders einrückt**.

Der Inhalt des Ordners `web`, der vom Browser dargestellt wird, soll künftig mittels Webpack generiert werden. Erstellen Sie dazu im Wurzelverzeichnis des Projektes die Datei `webpack.config.js` (die Sie unbedingt auch in Git speichern sollten). Fügen Sie folgenden Code in diese Datei ein.

`webpack.config.js`

```
const
  path       = require('path'),
  CopyPlugin = require('copy-webpack-plugin');

function absolutePath(l_path)
{ return path.resolve(__dirname, l_path); }

module.exports =
{ entry:
  { head: absolutePath('src/css/head.css'),
    main: absolutePath('src/js/main.js'),
  },

  output:
  { filename: 'js/[name].js',
    path: absolutePath('web'),
  },

  plugins:
  [ new CopyPlugin
    ({ patterns:
       [{ context: 'src',
          from:    '*.html',
          to:      '.',
        },
      ],
    }),
  ],

  module:
  { rules:
    [ { test:    /head\.css$/,
        include: [ /src/ ],
        use:     [ { loader: 'file-loader',
                     options: { name: '[name].[ext]',
                                outputPath: 'css/',
                              }
                   },
                   'extract-loader',
                   'css-loader',
                 ],
      },

      { test:    /\.css$/,
        include: [ /src/ ],
        exclude: [ /head\.css$/ ],
        use:     [ 'style-loader',
                   'css-loader',
                 ],
```

```
      },
    ],
  },
};
```

Es handelt sich um eine JavaScript-Datei, die direkt von Node.js interpretiert wird. Aus diesem Grund wird auch nicht das ECMAScript-6-Modul-Konzept eingesetzt, sondern CommonJS, das Standard-Modulkonzept vom Node. Hier werden Module mittels `require` und nicht mittels `import` importiert. Für Sie ist das allerdings nebensächlich, da Sie im Rahmen der Tutorien keine eigenen komplexen Konfigurationsdateien für webpack implementieren werden.

Im Prinzip macht diese Datei nichts anderes als ein Objekt zu exportieren, das Webpack-Konfigurationsinformationen enthält:

`entry`: JavaScript-Dateien, die gebündelt und komprimiert werden sollen. Man muss jeweils nur die Wurzeldatei angeben. webpack sucht in dieser Datei rekursiv nach Import-Befehlen und packt die benötigten Dateien ebenfalls in die Ausgabedatei.
`output`: Hier wird festgelegt, wie die gepackten Dateien heißen sollen. Für jeden Eintrag im Attribute `entry` wird eine Datei mit dem Namen `web/js/[name].js` erzeugt, wobei `[name]` durch den Schlüsselname des Entry-Eintrags ersetzt wird. In der obigen Datei gibt es zwei Entry-Elemente.
`plugins`: Hier werden Webpack-Hilfsmodule geladen, die bestimmte Zusatzaufgaben erledigen. Sie verwenden derzeit nur das Modul `CopyPlugin`. Dieses kopiert alle HTML-Dateien (das ist in Ihrem Fall nur die Datei `index.html`) vom Ordner `src` in den Ausgabe-Ordner `web`. (In einer späteren Webpack-Konfiguration wird auch noch die HTML-Datei komprimiert werden.)
`module`: In diesem Attribut werden Regeln (`rules`) angegeben,die festlegen, auf welche Weise eine Datei transformiert werden sollen. Bei jeder Regel gibt es einen Test, der festlegt, für welche Dateien die jeweilige Regel zutrifft. Mit Hilfe eines regulären Ausdrucks wird im Tutoriumsbeispiel geprüft, ob die aktuelle Datei auf `head.css` oder `.css` endet, sich aber von `head.css` unterscheidet. Je nach Dateiname werden andere Loader zum Laden und Verarbeiten der zugehörigen Datei verwendet. (Für webpack gibt es diverse Loader, einige davon haben Sie zuvor mit Hilfe von `npm` installiert.) Leider gibt es soviel Loader, dass die Auswahl geeigneter Loader nicht gerade einfach ist. Erschwerend kommt hinzu, dass sich mit jeder neuen Hauptversion von webpack nicht nur die Struktur der Konfigurationsdatei, sonder auch die verfügbaren Loader und Plugins verändern. :-( )

CSS-Dateien werden zunächst mit dem CSS-Loader geladen und verarbeitet. Anschließend wird die Datei `head.css` vom File-Loader Anweisungen in die Datei `web/css/head.css` geschrieben. Dies hätten wir auch mit dem Copy-Plugin erreicht. Allerdings kann man für den File-Loader einen Optimierer hinzufügen, der die CSS-Datei komprimiert (minimiert), bevor sie in den Web-Ordner eingefügt wird (siehe Version 5b).

Alle übrigen CSS-Dateien werden nicht auf die Platte geschrieben, sondern in die JavaScript-Datei eingefügt, die die CSS-Datei importiert. Dies ist die Aufgabe des Style-Loaders. JavaScript-Dateien werden von Webpack automatisch zu einer Datei zusammengefasst und (im Produktionsmodus) minimiert. Dies ist das Defaultverhalten des **JavaScript**-Bundlers und braucht daher nicht in eigenen Regeln beschrieben zu werden.

Mit dem Style-Loader wird erreicht, dass die Datei `src/css/main.css` in der Ausgabedatei `web/js/main.js` enthalten ist. Damit wird die Zahl der Dateien reduziert: Aus `src/js/main.js`, `src/js/greet.js` und `src/css/main.css` erzeugt Webpack **eine** (minimierte) Datei namens `web/js/main.js`

Damit steht einer Verwendung von Webpack nichts mehr im Wege. Man könnte Webpack nun direkt

von der Kommandozeile aufrufen. Besser ist es allerdings NPM-Skript-Befehle zu verwenden.

4.4 NPM-Skripte

Fügen Sie in das `scripts`-Objekt der Datei `package.json` folgende Skript-Anweisungen ein:

```
"scripts": {
  "predev": "rm -rf web/*",
  "dev": "webpack --mode development",
  "preprod": "rm -rf web/*",
  "prod": "webpack --mode production",
  "postprod": "rm web/js/head.js",
  "prewatch": "rm -rf web/*",
  "watch": "webpack --mode development --watch"
}
```

Dieser Konfiguration definiert drei NPM-Skripte: `dev`, `prod` und `watch`. Diese Skript können Sie von der Kommandozeile aus starten:

```
npm run dev
npm run prod
npm run watch
```

Die Skripte `predev`, `preprod` und `prewatch` werden jeweils ausgeführt, bevor die eigentlichen Skripte ausgeführt werden. In allen drei Fällen wird der Web-Ordner vollständig gelöscht, da er ja neu erstellt werden soll. Für das NPM-Skript `prod` gibt es auch noch das Skript `postprod`, das nach dem Skript `prod` ausgeführt wird. Es löscht die Datei `web/js/head.js`, die von Webpack angelegt wurde. Sie wird allerdings nicht mehr gebraucht, da mit daraus mit dem Extract-Loader CSS-Inhalt aus der Datei `web/js/head.js` extrahiert und mit dem File-Loader in die Datei `web/css/head.css` extrahiert wird. Diese CSS-Datei wird von der Datei `web/index.html` direkt eingelesen.

Geben Sie nun im Terminal die Anweisung `npm run dev` (dev = development) ein.

Wenn Sie alles richtig gemacht haben, müsste Webpack fehlerfrei durchlaufen und die Dateien `web/index.html`, `web/js/main.js` sowie `web/css/head.css` (sowie die überflüssige Datei `web/js/head.js`) erstellen. Im Zweifelsfall müssen Sie auf das Reload-Icon hinter `HELLOWRLD05A` klicken.

Wenn Sie jetzt die Datei `index.html` im Browser öffnen, sollten Sie nach kurzer Zeit wieder das Begrüßungsmenü sehen. Nach Eingabe des Namens und Klick auf den Say-hello-Button oder Drücken der Enter-Taste sollten Sie freundlich begrüßt werden.

Allerdings ist das CSS-Layout immer noch mangelhaft. Die Datei `main.css` wird bislang noch nicht eingebunden. Das können Sie ganz einfach ändern. Fügen Sie in die Datei `src/js/main.js` folgenden Import-Befehl am Anfang der Datei ein:

```
import '../css/main.css';
```

Dieser Befehl ist in JavaScript eigentlich nicht erlaubt. Wenn Sie jetzt versuchen, die Datei `src/index.html` im Browser zu öffnen, werden Sie in der Browser-Konsole eine Fehlermeldung erhalten.

Webpack unterstützt allerdings das Importieren von CSS-Dateien in JavaScript-Dateien. Das Tool wurde so konfiguriert, dass die CSS-Datei dynamisch in das HTML-Dokument „injiziert“ wird (siehe oben).

Geben Sie im Terminal noch einmal die Anweisung `npm run dev` ein und öffnen Sie die Datei `web/index.html` im Browser (nicht die Datei `src/index.html`!). Jetzt sollte sie wieder sauber formatiert sein (evtl. erst nach einem Reload im Browser)!

Sehen Sie sich mal den Inhalt der Datei `main.js` an. Darin finden Sie – neben vielen Kommentaren, die zur Strukturierung der Datei verwendet werden – die Inhalte der Dateien `main.js`, `greet.js` und `main.css` sowie spezielle Anweisungen, die von Webpack eingefügt wurden.

In dieser Form ist die Datei für Entwicklungszwecke einigermaßen brauchbar. Bei einem Fehler zeigt einem der Browser, in welcher Zeile dieser Datei der Fehler aufgetreten ist. Normalerweise handelt es sich um eine Zeile, die Sie erfasst haben. Webpack-Befehle sollten keine Fehler werfen. Nachteilig ist, dass diese Zeile etwas anders aussehen kann, wie in Ihrem Sourcecode, da sie eventuell transformiert wurde. Aber man findet die Originalzeile normalerweise recht schnell.

Gehen Sie nochmal ins Terminal und geben Sie diesmal die Anweisung `npm run prod` (prod = production) ein. Es werden wieder die drei Dateien erstellt. Und die Web-Anwendung sollte immer noch funktionieren. Wenn Sie sich die gebündelte JavaScript-Datei `web/js/main.js` noch einmal ansehen, werden Sie feststellen, dass sie nun viel kleiner ist. Sie enthält keine überflüssige Leerzeichen, keine Zeilenumbrüche und auch keine Kommentare mehr (evtl. mit Ausnahme des Kommentars, der Auskunft über den Autor der CSS-Datei `main.css`). Diese Datei ist für den Produktivbetrieb viel besser geeignet, da sie weniger Bandbreite verbraucht. Insbesondere Smartphone-Besitzer freuen sich über kleine Dateien, da ihr Datenvolumen dadurch weniger belastet wird. Für die Fehlersuche bei der Entwicklung der Web-Aanwendung ist diese Variante allerdings vollkommen ungeeignet.

Nun können Sie noch die Anweisung `npm run watch` testen. Dieser startet einen Webpack-Watcher, der bei jeder Änderung an einer Datei, die von Webpack beim Bündeln berücksichtigt wird, dafür sorgt, dass die Dateien im Web-Ordner neu erstellt werden und Sie die Ergebnisse Ihrer Änderung durch einen einfachen (Auto-)Reload der Datei `index.html` im Browser betrachten können. Ändern Sie doch einmal in der CSS-Datei `src/css/head.css` (**nicht in der Datei `web/css/head.css`**) die Hintergrundfarbe, speichern Sie die Datei und laden Sie danach die HTML-Datei im Browser mittels des Reload-Buttons neu.

5 Verbesserungen der Webpack-Konfiguration

5.1 Hinzufügen verschiedener Minimizer

Musterlösung: `index.html` (5b) (Git-Repository: 5b)

Wenn man die Dateien betrachtet, wurde nur die Datei `main.js` im Produktions-Modus minimiert. Künftig sollen alle Dateien im Produktionsmodus minimiert werden.

Dafür brauchen wir drei neue Node.js-Pakete, zum Minimieren von CSS-, HTML- und JavaScript-Dateien:

```
npm i -D css-minimizer-webpack-plugin
npm i -D html-minimizer-webpack-plugin
npm i -D terser-webpack-plugin
```

Nun muss man diese drei Plugins noch in die Webpack-Konfigurationsdatei `webpack.config.js` einarbeiten.

Zunächst muss die neuen Pakete importieren. Dazu ersetzt fügt man am Anfang der Datei drei weitere Require-Anweisungen ein.

```
const
  path       = require('path'),
  CopyPlugin = require('copy-webpack-plugin');
```

wird zu

```
const
  path                = require('path'),
  CopyPlugin          = require('copy-webpack-plugin'),
  TerserPlugin        = require('terser-webpack-plugin'),
  CssMinimizerPlugin  = require('css-minimizer-webpack-plugin'),
  HtmlMinimizerPlugin = require('html-minimizer-webpack-plugin');
```

Außerdem muss das Konfigurations-Objekt, das exportiert wird, künftig mit einer Funktion berechnet werden, damit man ermitteln kann, ob der Benutzer Webpack im Development- oder im Produktionsmodus ausführt.

```
module.exports =
{ entry:
  ...
};
```

wird zu

```
module.exports =
function(p_env, p_argv)
{ const isProd = p_argv.mode === 'production';

  return {

  // the mode of the current webpack run: 'development' or 'production'
  mode: p_argv.mode || 'development',

  entry:
  ...
}};
```

Innerhalb dieser Funktion steht einem nun die Konstante `isProd` zur Verfügung. Sie hat den Wert `false`, wenn Webpack im Development-Modus ausgeführt wird, und `true`, wenn Webpack im Produktionsmodus läuft.

Zu guter Letzt muss man noch ein paar Optimierungs-Regeln im Anschluss an das Objekt `module` in die Konfigurationsdatei einfügen.

```
module.exports =
{ module:
  { ...
  },
};
```

wird zu

```
{ module:
  { ...
  },

  optimization:
  { minimize: isProd,

    minimizer:
    [ new TerserPlugin                                  // minimize JS
          ({ extractComments: true, }),
      new CssMinimizerPlugin                            // minimize CSS
          (),
      new HtmlMinimizerPlugin                           // minimize HTML
          ({ minimizerOptions:
             { collapseBooleanAttributes:   true,
               collapseWhitespace:          true,
               collapseInlineTagWhitespace: true,
             }
          }),
    ],
  },
};
```

Das Attribut `minimize` wird im Development-Modus auf `false` gesetzt und im Produktions-Modus auf `true`. Das heißt, im Development-Modus werden keine Dateien minimiert. Im Produktionsmodus werden dagegen sowohl JS-, als auch CSS-, als auch HTML-Dateien minimiert. Die einzelnen Minimizer heißen unterschiedlich, stammen von unterschiedlichen Autoren und akzeptieren unterschiedliche Konfigurationsattribute. Dies ist ganz typisch für Webpack. Und in der nächsten Version funktioniert der ein oder andere Minimizer gar nicht mehr. Allerdings sind in der Zwischenzeit zumindest das Terser-Plugin und das CSS-Minimizer-Plugin offizieller Bestandteil von Webpack.

Wenn Sie jetzt `npm run dev` ausführen, sollte keine Datei im Web-Ordner komprimiert werden. Führen Sie dagegen `npm run prod` aus, sollten alle Dateien minimiert werden. Außerdem werden die Lizenzinformationen in eine eigene Datei ausgelagert.

5.2 Inlining von `head.css`

Musterlösung: `index.html` (5c) (Git-Repository: 5c)

Google PageSpeed Insights hatte angemahnt, den Inhalt der Datei `head.css` direkt in die Datei `index.html` einzufügen.

> Google PageSpeed Insights Version 5b: Ressourcen beseitigen, die das Rendering blockieren: `.../css/head.css`
> Google PageSpeed Insights Version 5c: Hier gibt es diese Empfehlung nicht mehr.

Um die Datei `head.css` in die Datei `index.html` einzufügen, muss man die HTML-Datei erzeugen. Ein einfaches Kopieren reicht nicht mehr aus. Anstelle des Copy-Plugins verwendet man das HTML-Webpack-Plugin. Dieses kann HTML-Dateien auf Basis von HTML-Templates erstellen und auch komprimieren. Da heißt, man benötigt auch das HTML-Minimizer-Plugin nicht mehr. Die CSS-Datei

kann man mittels, des HTML-CSS-Inline-Plugins einfügen. Dafür braucht man allerdings das Mini-CSS-Extract-Plugin anstelle des Extract-Loaders und des File-Loaders. Insgesamt bedeutet dies einen größeren Umbau der Konfigurationsdatei. Zunächst sollten veraltete Node.js-Pakete entfernt und die dafür benötigten Pakete installiert werden:

```
npm uninstall -D file-loader extract-loader
npm uninstall -D copy-webpack-plugin html-minimizer-webpack-plugin

npm i -D html-webpack-plugin html-replace-webpack-plugin
npm i -D mini-css-extract-plugin html-inline-css-webpack-plugin
```

Das HTML-Replace-Plugin wird benötigt, um an der modifizierten zum Schluss mit Hilfe von regulären Ausdrücken noch zwei kleine Änderungen vorzunehmen. Damit werden Probleme behoben, die sich ergeben, weil es in den benutzten Paketen keine geeigneten Konfigurationsoptionen gibt. Hier sieht man deutlich das Problem von Webpack: Es gibt sehr viele Loader, Plugins und Optimierer. Allerdings hat jedes Paket seine individuellen Schwächen und oft arbeiten zwei Pakete auch nicht wie gewünscht zusammen. Hier hilft meist nur Try and Error. Ob man zum Schluss die beste Variante gewählt hat, ist dabei unklar.

Die Datei `webpack.config.js` wird folgendermaßen geändert:

```
const
  path                = require('path'),
  CopyPlugin          = require('copy-webpack-plugin'),
  TerserPlugin        = require('terser-webpack-plugin'),
  CssMinimizerPlugin  = require('css-minimizer-webpack-plugin'),
  HtmlMinimizerPlugin = require('html-minimizer-webpack-plugin');
```

wird zu

```
const
  path                       = require('path'),
  TerserPlugin               = require('terser-webpack-plugin'),
  CssMinimizerPlugin         = require('css-minimizer-webpack-plugin'),
  MiniCssExtractPlugin       = require('mini-css-extract-plugin'),
  HtmlWebpackPlugin          = require('html-webpack-plugin'),
  HTMLInlineCSSWebpackPlugin =
require('html-inline-css-webpack-plugin').default,
  HtmlReplaceWebpackPlugin   = require('html-replace-webpack-plugin');
```

Das Plugins-Array muss vollständig ersetzt werden werden:

```
plugins:
[ ...
],
```

wird zu

```
plugins:
[ new MiniCssExtractPlugin
  ({ filename: "[name].css" }),
  new HtmlWebpackPlugin
  ({ template: absolutePath('src/index.html'),
     chunks:   ['head', 'main'],
     minify:
     { collapseWhitespace:   isProd,
       removeComments:       true,
       ignoreCustomComments: [/@preserve/],
     },
  }),
  new HTMLInlineCSSWebpackPlugin
  (),
  new HtmlReplaceWebpackPlugin
  ([{ pattern:     /<script .*head\.js"><\/script>/,
      replacement: '',
    },
    { pattern:     /defer/g,
      replacement: 'async',
    },
  ]),
],
```

Das HTML-Webpack-Plugin kopiert in die Template-Datei `src/index.html` zwei Script-Anweisungen, um die von Webpack erzeugten Dateien (“Chunks”, Klumpen) `web/js/head.js` und `web/js/main.js` zu Laden. Diese Anweisungen dürfen also in der Template-Datei nicht enthalten sein. Wie man sieht, kann man im HTML-Webpack-Plugin die Komprimierung mittels der Option `minify` direkt konfigurieren. Auch hier werden die Leerzeichen und -zeilen nur im Produktivmodus, aber nicht im Developmentmodus entfernt.

Mit Hilfe des HTML-Replace-Plugins wird der Eintrag `<script "src=css/head.js"></script>` wieder aus `index.html` gelöscht. Diese JavaScript-Datei enthält nur den CSS-Code, der mittels des HTML-Inline-CSS-Plugins direkt eingefügt wurde. Warum dieses Plugin das überflüssige Script-Tag nicht selbst entfernt, habe ich noch nicht herausgefunden.

Außerdem wird `defer` durch async ersetzt, da ich das asynchrone Laden gegenüber dem verzögerten Laden von JavaScript-Dateien bevorzuge. Warum ich das im HTML-Webpack-Plugin nicht konfigurieren kann, weiß ich auch nicht.

Und die Regel für die `head.css` im Array `rules`) muss ersetzt werden.

```
{ test:    /head\.css$/,
  include: [ /src/ ],
  use:     [ MiniCssExtractPlugin.loader,
             'css-loader',
           ],
},
```

Wenn Sie jetzt `npm run dev` ausführen, sollte keine Datei im Web-Ordner komprimiert werden. Führen Sie dagegen `npm run prod` aus, sollten alle Dateien minimiert werden. Außerdem werden die Lizenzinformationen in eine eigene Datei ausgelagert. In beiden Fällen sollte der Inhalt der CSS-Datei `head.css` in der HTML-Datei selbst enthalten sein. Einmal unkomprimiert und einmal komprimiert.

5.2.1 Ein kleine Verbesserung der Import-Anweisungen

Wenn Sie schon gerade dabei sind, fügen Sie noch folgendes Konfigurationsobjekt in die Export-Funktion von `webpack.config.js` ein:

```
resolve:
{ alias: { '/json': absolutePath('src/json/'),
           '/css':  absolutePath('src/css/'),
           '/img':  absolutePath('src/img/'),
         },
},
```

Damit erreichen Sie, dass Dateien, die sich in den Ordnern `src/json/`, `src/css/`, `src/img/` befinden, direkt importiert werden können. Die Notwendigkeit, relative Pfadangaben wie `import '../../../css/irgenwas.css';` zu verwenden, entfällt. Sie können künftig einfach `import '/css/irgenwas.css';` schreiben. Das gilt auch für Bilder oder JSON-Dateien, die Sie (mittels des erweiterten Import-Befehls von Webpack) in eine JavaScript-Datei importieren. Sie können selbstverständlich beliebig viele weitere Alias-Befehle definieren, natürlich auch für JavaScript-Bibliotheken, die in anderen Ordnern enthalten sind.

Ersetzen Sie also in der Datei `src/main.js`, den Befehl `import '../css/body.css';` durch `import '/css/body.css';`

5.3 Verwendung des SCSS-Minimizers

Musterlösung: `index.html` (5d) (Git-Repository: 5d)

Wenn man sich die Datei `web/js/main.js` ansieht, bemerkt man, dass die CS-Befehle, die von Webpack in diese Datei eingefügt wurden (wegen des Import-Befehls `import '/css/body.css';`), im Produktions-Modus nicht komprimiert werden. Da heißt, dass das CSS-Minimizer-Plugin hier nicht greift. Also sollte es ersetzt werden.

Glücklicherweise gibt es einen ausgezeichneten Ersatz: Den SASS-Loader. Anstelle von CSS sollte man

SASS/SCSS verwenden, um auch CSS modularisieren und „DRY machen“ zu können. Glücklicherweise enthält diese Loader einen eigenen Minimizer.

Modifizieren Sie die Pakete entsprechend:

```
npm uninstall -D css-minimizer-webpack-plugin
npm i -D sass sass-loader
```

Löschen Sie das CSS-Minimizer-Plugin aus der Datei `webpack.config.js`, sowohl den Require-Befehl, als auch im Minimizer-Array im Optimization-Objekt.

Fügen Sie dafür den SASS-Loader in die CSS-Objekte des Rules-Arrays ein.

```
module:
{ rules:
  [ { test:    /head\.(css|scss|sass)$/,
      include: [ /src/ ],
      use:     [ MiniCssExtractPlugin.loader,
                 'css-loader',
                 { loader: 'sass-loader',
                   options:
                   { sassOptions: { outputStyle: isProd ? 'compressed' :
'expanded' } },
                 },
               ],
    },

    { test:    /\.(css|scss|sass)/,
      include: [ /src/ ],
      exclude: [ /head\.(css|scss|sass)$/ ],
      use:     [ 'style-loader',
                 'css-loader',
                 { loader: 'sass-loader',
                   options:
                   { sassOptions: { outputStyle: isProd ? 'compressed' :
'expanded' } },
                 },
               ],
    },
  ],
},
```

Dieser Loader übersetzt SCCS- und SASS-Anweisungen in CSS-Anweisungen. Er bietet zusätzlich eine Option `outputStyle` an, die festlegt, ob die CSS-Anweisungen komprimiert werden sollen oder nicht. Dies nutzen wir aus, um die CSS-Anweisungen im Produktionsmodus zu komprimieren.

5.4 Weiteres Optimierungspotenzial

Google PageSpeed Insights Version 5d ist immer noch nicht ganz zufrieden.

Zum einen sollten die Anzahl der Dateien (zwei) und die Dateigrößen weiter reduziert werden, was aber schwierig ist. Man könnte eine Datei `budget.json` definieren, die Google mitteilt, welche Dateianzahl und welche Dateigrößen man für akzeptabel hält. Dan erhält man erst ab diesen Werten eine Warnung.

Zum anderen teilt einen Google das größte Element in der HTML-Seite mit. Hier gibt es tatsächlich noch Optimierungspotenzial, gerade bei Single-Page-Anwendungen. Eine Single-Page wird in der Regel auf einmal geladen, aber nur portionsweise angezeigt. Wichtig ist eigentlich nur, dass der Inhalt „above the fold“ („über dem Zeitungsknick“) in der HTML-Seite enthalten ist. Das ist der Bereich, den der Benutzer nach dem Laden der Seite ohne Scrollen sieht. Bei der Hello-World-Anwendung wäre das die Section mit der ID `section_form`. Alle übrigen Bestandteile der HTML-Seite könnten dynamisch nachgeladen werden, sobald sie benötigt werden. Genauso funktioniert Google Maps: Es werden nur die Kacheln der Erdoberfläche in der Auflösung geladen, die momentan benötigt werden. Um Verzögerungen, die durch das Nachladen entstehen, gerin zu halten, werden Randkacheln und Kacheln in benachbarter Auflösung schon mal im Voraus geladen, während der Benutzer die aktuellen Karte betrachtet. Es ist natürlich nicht sichergestellt, dass die Kacheln tatsächlich irgendwann angezeigt werden müssen. Aber es ist wahrscheinlich. Und daher wird der potenziell unnötige Datenverkehr in Kauf genommen. Hier kommt es auf die Abwägung zwischen User Experience (möglichst kurze Wartezeiten) und Energie- und Kosteneffizienz (möglichst geringes Transfervolumen) an.

Das Transfervolumen kann weiter reduziert werden, indem die Dateien vor dem Ausliefern vom Server mittels gzip weiter komprimiert werden. Der Client muss sie dann erst wieder entpacken, bevor er sie verarbeiten kann. Dies ist aber eine Server-Optimierung, die nicht mit Webpack oder einen vergleichbaren Tool realisiert werden kann.

6 Fortsetzung des Tutoriums

Sie sollten nun Teil 6 des Tutoriums bearbeiten. In diesem Teil des Tutoriums wird ausgenutzt, dass in die Konfigurationsdatei von Webpack ein SCSS-Loader integriert wurde. Damit kann man das Prinzip “**D**on't **R**epeat **Y**ourself” (DRY) auch für CSS-Dateien umsetzen.

7 Quellen

Webpack-Homepage
Browserify-Homepage
FuseBox-Homepage
https://webpack.js.org/plugins/copy-webpack-plugin/
https://webpack.js.org/loaders/
Kowarschick (MMProg): Wolfgang Kowarschick; Vorlesung „Multimedia-Programmierung“; Hochschule: Hochschule Augsburg; Adresse: Augsburg; Web-Link; 2018; Quellengüte: 3 (Vorlesung)

Multimedia-Programmierung/Tutorium
Praktikum:MMProg
HTML5-Tutorium: JavaScript: Hello World
HTML5-Beispiel
Kapitel:Multimedia-Programmierung:Beispiele