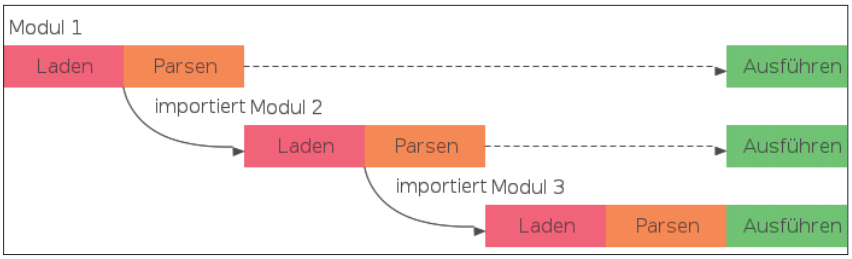
JAVASCRIPT-BUNDLING

Nur das, was man braucht

Mit Rollup modularen JavaScript-Code ohne Overhead für NPM und Web optimieren.

JavaScript ist einen weiten Weg gegangen. Aus der von Brendan Eich legendär „in zehn Tagen" entwickelten dynamischen Skriptsprache für triviale User-Interaktionen ist eine Basis für Unternehmensanwendungen mit teilweise über sechsstelligen Zeilenzahlen geworden, an denen große Entwicklerteams über Jahre arbeiten. Dies ist umso bemerkenswerter, wenn man bedenkt, dass der Sprache lange Zeit ein wichtiges Merkmal fehlte, das sicherlich grundlegend für das Entwickeln großer Applikationen ist: eine Möglichkeit der Modularisierung.



Wasserfall-Effekt beim Laden abhängiger Module (Bild 1)

Not macht erfinderisch

Spuren davon sieht man heute noch selbst in den modernsten Webanwendungen, wenn man einmal einen Blick darauf wirft, wie diese ihren Code verpacken, bevor er vom Browser geladen wird.

```
(function(){
  var localVariable =  ... // nicht ausserhalb der
    // umgebenden Funktion sichtbar
  window.myExport = ... // über das globale Objekt
    // "window" überall verfügbar
}());
```

Dieser Trick einer sogenannten Immediately-Invoked Function Expression, kurz IIFE, stellt sicher, dass Variablen, die in verschiedenen Skript-Tags einer HTML-Seite verwendet werden, sich nicht gegenseitig überschreiben. Gleichzeitig können Informationen zwischen verschiedenen Modulen ausschließlich über globale Variablen ausgetauscht werden. Diese Notlösung erlaubt wenig Kontrolle darüber, ob und woher Variablen importiert werden, und lässt sich relativ schlecht skalieren.

All dies änderte sich, als 2009 mit Node.js, dem Paketmanager NPM und dem CommonJS-Format [1] eine Lösung vorgestellt wurde, welche die JavaScript-Welt revolutionieren sollte und endlich eine solide Basis für professionelle Webentwicklung bereitstellte. Und dies, obwohl Node eigentlich nur für den Betrieb als Server gedacht und konzipiert war.

Tooling für das Web

Nicht nur wird das CommonJS-Modulsystem von Webbrowsern nicht direkt unterstützt, eine feine Aufspaltung von Code stößt bei Browsern auch grundsätzlich an Grenzen. Einerseits limitieren sie streng das parallele Laden von Dateien, andererseits müsste man mit wasserfallartigen Effekten kämpfen, wie sie Bild 1 verdeutlicht.

Eine Lösung brachten sogenannte Bundler. Browserify [2] war einer der ersten und primär dafür gedacht, um NPM-Module ohne Änderung im Browser nutzbar zu machen. Dafür wurde der Code zusammen mit speziellen browserkompatiblen Versionen wichtiger NPM-Bibliotheken in eine gemeinsame Datei, ein sogenanntes Bundle, gepackt.

Browserify hat inzwischen viel von seiner ursprünglichen Bedeutung zugunsten von Webpack verloren, das heute sicher als Standardlösung in diesem Bereich gilt [3]. Die Bindung an das von Node eingeführte dynamische CommonJS-Format brachte jedoch einen Nachteil, den diese Bundler bis heute nicht vollständig überwinden können: Erzeugte Bundles mussten eine Laufzeitumgebung enthalten, das heißt zusätzlichen Code, um Importe und Exporte zur Laufzeit auflösen zu können. Um dies zu verstehen, hilft es, einen Blick darauf zu werfen, wie CommonJS-Dateien ausgeführt werden.

CommonJS: mächtig, dynamisch, schwer zu analysieren

Im folgenden einfachen Beispiel startet die Ausführung bei *index.js*. Im CommonJS-Format importiert die Funktion *require()* eine Moduldatei und liefert als Rückgabe den Wert, den das Modul zuletzt der internen Variable *module.exports* zugewiesen hat:

```
// index.js
const randomNumber = Math.random();
if (randomNumber < 0.5) console.log("index.js 1:",
  require("./dependency.js"));
console.log("index.js 2");
console.log("index.js 3:", require("./dependency.js"));

// dependency.js
console.log("dependency.js");
module.exports = "Gruß von dependency ";
```

Was passiert hier? Wenn *randomNumber* kleiner als 0,5 ist, ist die Ausgabe:

```
dependency.js
index.js 1: Gruß von dependency
index.js 2
index.js 3: Gruß von dependency
```

Andernfalls erhält man:

```
index.js 2
dependency.js
index.js 3: Gruß von dependency
```

Dabei gilt es zu beachten: Sobald ein Modul einmal per *require()* importiert wurde, führen es weitere Importe nicht erneut aus.

Um eine derartige Dynamik in den Griff zu bekommen, packen Bundler wie beispielsweise Webpack die Inhalte der Dateien intern in Modulfunktionen, in denen jeweils eine simulierte *module.exports*-Variable und eine *require()*-Funktion bereitgestellt werden. Außerdem fügen sie eine Laufzeitumgebung hinzu, die entscheidet, welcher Import welche Modulfunktion ausführt und was der richtige Rückgabewert ist (Bild 2).

Als es daran ging, ein „offizielles" Modulformat für JavaScript zu entwerfen, war gerade diese Unvorhersagbarkeit einer der Gründe, warum dann mit ECMAScript 2015 (auch bekannt als ES6) ein gänzlich anderes Format das Rennen machte [4].

ES-Module: Nichts bleibt hier dem Zufall überlassen

Das neue, native Modulformat brachte eine Reihe von Beschränkungen für Importe. So können diese nur als Anweisungen auf oberster Ebene eines Moduls verwendet und somit nicht in Ausdrücke eingebunden werden, wie es für *require()* im vorigen Beispiel möglich war. Außerdem sind sie aus dem normalen Programmfluss herausgehoben: Bevor die erste Codezeile eines Moduls ausgeführt wird, werden zuerst alle Importe in der Reihenfolge ihres Auftretens abgearbeitet. Ein Beispiel:

```
// index.js
console.log("index.js 1");
import {value} from "./dependency.js";
console.log("index.js 2:", value);

// dependency.js
console.log("dependency.js");
export const value = "Gruß von dependency ";
```

Dieser Code liefert die folgende Ausgabe:

```
dependency.js
index.js 1
index.js 2: Gruß von dependency
```

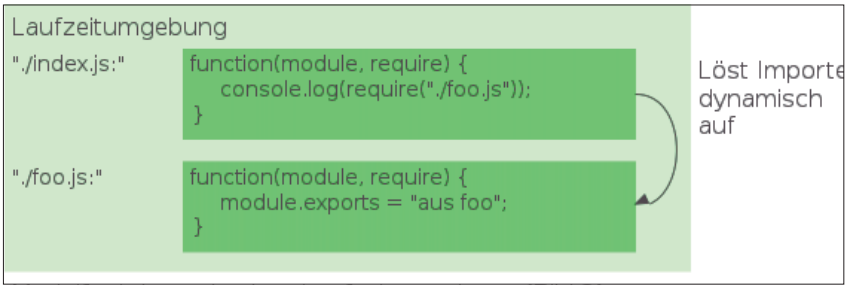
Wie im CommonJS-Format führen auch hier wiederholte Importe derselben Datei (auch aus verschiedenen Modulen) nur zum einmaligen Ausführen der Datei. Im Vergleich zu CommonJS ist allerdings eine wichtige Verbesserung geschaffen: Die Ausführungsreihenfolge der Dateien ist bereits von Anfang an bekannt und es ist nicht mehr nötig, Importe über eine Laufzeitumgebung dynamisch aufzulösen. ES-Module haben jedoch noch ein weiteres Merkmal, das Laufzeitumgebungen überflüssig macht.

Variablen statt Referenzen

Auch wenn sich in einem CommonJS-Modul der Wert des Exports dynamisch ändern kann, müssen Sie in importierenden Modulen den Import per *require()* wie im folgenden Beispiel immer wieder erneut ausführen, um an den aktuellen Wert zu kommen:

```
// index.js
const value = require("./dependency.js");
console.log(value); // Wert 1
setTimeout(() => {
  console.log(value);
    // Wert 1 - selbst nach zwei Sekunden noch
  console.log(require("./dependency.js"));
    // Wert 2 - nur durch den erneuten Import ist der
    // aktuelle Wert verfügbar
}, 2000);

// dependency.js
module.exports = "Wert 1";
setTimeout(module.exports = "Wert 2", 1000);
  // Nach einer Sekunde wird der Export neu zugewiesen
```



Modulfunktionen in einer Laufzeitumgebung (Bild 2)

Daher muss ein Bundler bei jedem Import eine Kopie von *module.exports* erstellen. In ES-Modulen wird hingegen statt eines Wertes oder einer Objektreferenz die Variable selbst exportiert. In diesem Format sind daher weder ein erneuter Import noch ein erneuter Export nötig:

```
// index.js:
import {value} from "./dependency.js";
console.log(value); // Wert 1
```

```
setTimeout(() => console.log(value),
  2000); // Wert 2 - der Import ändert
  // seinen Wert dynamisch

// dependency.js:
export let value = "Wert 1";
setTimeout(value = "Wert 2", 1000);
  // Nach einer Sekunde wird die Variable verändert
```

Also kann ein Bundler die Module nicht nur in einer festen Reihenfolge ausführen, sondern kann auch im Bundle die gleichen Variablen für Importe und Exporte verwenden. Aus dieser Idee wurde Rollup geboren: ein Bundler ohne Laufzeitumgebung für natives JavaScript [5].

Modularisierung ohne Einbußen

Um dies an einem Beispiel auszuprobieren, sollten eine aktuelle Version von Node und damit die Befehle *npm* und *npx* auf Ihrem System verfügbar sein. Setzen Sie ein neues NPM-Projekt in einem Ordner auf:

```
$ npm init --yes # erzeugt eine package.json-Datei,
  # um Module von NPM installieren zu können
$ npm install rollup # macht den "rollup"-Befehl im
  # Projekt verfügbar
```

Legen Sie außerdem folgende Dateien an:

```
// src/index.js
import {value as importedValue} from "./dep1.js";
console.log("index.js:", importedValue);
export const value = "Index";

// src/dep1.js:
import {value as importedValue} from "./dep2.js";
console.log("dep1.js:", importedValue);
export const value = "Dependency 1";

// src/dep2.js:
console.log("dep2.js");
export const value = "Dependency 2";
```

Diese Module sollen nun in eine Datei namens *dist/bundle.js* gepackt werden. Dafür ist eine Konfigurationsdatei nötig:

```
// rollup.config.js:
export default {
  input: "src/index.js",
  output: {
    file: "dist/bundle.js",
    format: "esm"
  }
};
```

Das Bundle lässt sich nun über die Kommandozeile erzeugen:

```
$ npx rollup -c
```

Bei richtiger Benennung liest das Kommando die Konfigurationsdatei aus dem aktuellen Verzeichnis ein und erzeugt das Bundle. Hier ist das Ergebnis:

```
// dist/bundle.js:
console.log("dep2.js");
const value = "Dependency 2";

console.log("dep1.js:", value);
const value$1 = "Dependency 1";

console.log("index.js:", value$1);
const value$2 = "Index";

export { value$2 as value };
```

Tatsächlich gibt es in diesem Code keine sichtbaren Modulgrenzen mehr. Das bedeutet, dass beliebig feine Modularisierungen keine Nachteile für das Laufzeitverhalten des erzeugten Pakets bedeuten. Diesen Prozess nennt man Scope Hoisting, das heißt so viel wie „Heben der Module in einen gemeinsamen Scope". Bemerkenswert ist, dass Rollup automatisch den Namenskonflikt zwischen den *value*-Variablen durch Umbenennung aufgelöst hat.

Dies unterscheidet sich durchaus von dem, was viele andere Bundler erzeugen. Obwohl einige inzwischen ebenfalls Scope Hoisting unterstützen, packen sie normalerweise immer noch eine Laufzeitumgebung mit verschiedenen Modulfunktionen in das Bundle. Das Ergebnis hat dann beispielsweise in aktuellen Versionen von Webpack für dieses einfache Beispiel ein Vielfaches der Größe – selbst wenn das Bundle mit einem Minifier wie Uglify nachträglich optimiert wird.

Da in dem obigen Beispiel als Format *esm* angegeben war, ist das Ergebnis hier wieder ein ES-Modul. Das ist ideal, wenn zum Beispiel eine Bibliothek später nochmals von einem Bundler verarbeitet werden soll. In modernen Browsern können Sie den Code aber auch direkt über ein *<script>*-Tag mit *type="module"* konsumieren. Für ältere Browser ist allerdings etwas Arbeit nötig, wie sich weiter unten noch herausstellen wird.

Rollup bietet jedoch noch ein zweites Merkmal, das in vielen Fällen den Umfang des Codes noch erheblich reduzieren kann.

Tree-Shaking im Syntaxbaum

Entgegen einer weit verbreiteten Annahme hat Tree Shaking nichts mit ES-Modulen zu tun und ist auch nicht auf diese beschränkt. Im Kern ist es ein Verfahren zur Identifikation ungenutzten Codes, welches das Prinzip des Mark-and-Sweep-Algorithmus zur Speicherbereinigung verfolgt [6][7].

Anhand der genutzten Exporte und der aufgerufenen Funktionen werden alle möglichen Ausführungspfade durchlaufen und dabei alle Teile des Syntaxbaums markiert, die aufgrund von Seiteneffekten für das Bundle benötigt werden. Da in ES-Modulen Variablen per Export aber Modulgrenzen überspannen können, ist hier das Verfahren merklich effektiver.

Dies macht sich sofort bemerkbar, wenn Sie im Code eine Variable definieren und diese nicht verwenden: Sie wird im späteren Bundle nicht auftauchen.

In Rollup ist das Verfahren jedoch noch merklich mächtiger und mit einigen Codetransformationen verknüpft, wie das folgende Beispiel zeigt:

```
// In einem beliebigen Modul
const config = { dev: true};
const log = (...args) =>
  config.dev && console.log(...args);
log("Development-Build");
```

Im Bundle bleibt davon Folgendes übrig:

```
const log = (...args) => console.log(...args);
log("Development-Build");
```

In der Tat erkennt Rollup, dass *config.dev* nur *true* sein kann, und vereinfacht die *log()*-Funktion. Da die Variable *config* ansonsten nicht verwendet wird, verschwindet sie aus dem Bundle. Ändern Sie den Wert von *config.dev* hingegen auf *false*, bleibt vom ganzen Beispiel nichts mehr übrig, da *log()* nun keine Seiteneffekte mehr hat.

Dies lässt sich zum Beispiel sinnvoll nutzen, indem man das *config*-Objekt von einem separaten Modul exportiert und mittels rollup-plugin-alias [8] je nach Situation entweder die Entwicklungs- oder die Produktivversion dieser Datei importiert. Damit steht dann eine Logging-Funktion für die Entwicklung zur Verfügung, die keine Spuren im Produktivcode hinterlässt.

Dieses Beispiel lässt sich natürlich beliebig für Inline-Tests, Analytics und so weiter erweitern. Die Möglichkeiten sind bereits bereits beeindruckend, trotzdem wird gerade Tree-Shaking in Rollup sehr aktiv weiterentwickelt, man darf also gespannt sein. Tree-Shaking lässt sich auch sehr gut im Rollup-REPL ausprobieren [9].

Formate für jeden Zweck

Neben dem nicht vorhandenen Overhead und den Codeoptimierungen ist vor allem die Möglichkeit, leicht in zahlreiche verschiedene Formate zu exportieren, ein Grund dafür, dass sich Rollup vor allem als Tool zum Packen von Bibliotheken etabliert hat, siehe zum Beispiel React und die Datums- und Zeit-Bibliothek Moment.js.

Um im Beispiel vom Anfang das Bundle auch in älteren Browsern ausführen zu können, können Sie das IIFE-Format nutzen, das die Idee der eingangs erwähnten „Notlösung“ aufgreift und den Code in einer Funktion versteckt. Der Export geschieht in diesem Fall über eine globale Variable, für die ein Name anzugeben ist. Dafür ändern Sie die Konfiguration folgendermaßen:

```
// rollup.config.js:
export default {
  input: "src/index.js",
  output: {
    file: "dist/bundle.js",
    format: "iife",
    name: "myGlobal"
  }
};
```

Das Ergebnis ist nun in eine Funktion gepackt, kann direkt in einem beliebigen <*script*>-Tag geladen werden und erzeugt eine globale Variable namens *myGlobal* als Exportobjekt:

```
var myGlobal = (function (exports) {
  // ...
  exports.value = value$2;
  return exports;
}({}));
```

Wählen Sie als Format stattdessen *umd*, das für Universal Module Definition steht, ist diese Datei sogar ohne Modifikation in Node per *require()* sowie in einem AMD-Loader verwendbar [10].

Fremder Code: Plug-ins und externe Module

Ein großer Vorteil der Modularisierung ist natürlich die Möglichkeit, externen Code zu verwenden. Hierfür gibt es zwei mögliche Ansätze. Der erste ist, Module als extern zu deklarieren. Dies führt dazu, dass das fertige Paket statt des Moduls einen Import für das Modul enthält.

Beim zweiten Ansatz lässt sich das Modul direkt aus NPM oder dem *node_modules*-Ordner mit in das Bundle packen. Da Rollup nativ nur JavaScript-eigene Merkmale unterstützt, ist in diesem Fall für NPM-Module mindestens rollup-plugin-node-resolve nötig [11]; das Modul erlaubt es Rollup, Importe, die keinen relativen Pfad enthalten, über das *node_modules*-Verzeichnis aufzulösen. Da diese Module meistens im CommonJS-Format vorliegen, ist außerdem normalerweise noch rollup-plugin-commonjs erforderlich [12], das CommonJS-Module in annähernd äquivalente ES-Module übersetzt.

Um im obigen Beispiel beispielsweise das NPM-Paket bluebird (eine Bibliothek zum Umgang mit Promises [13]) in einer Browser-kompatiblen Version zum Paket hinzuzufügen, installieren Sie dazu das Paket und die Plug-ins:

```
$ npm install bluebird rollup-plugin-node-resolve
  rollup-plugin-commonjs
```

Dann ändern Sie die Konfiguration folgendermaßen:

```
// rollup.config.js
import nodeResolve from "rollup-plugin-node-resolve";
import commonJs from "rollup-plugin-commonjs";

export default {
  input: "src/index.js",
  // sorgt dafür, dass die Browserversion von bluebird
  // gefunden, übersetzt und eingebunden wird
  plugins: [nodeResolve({browser: true}),
```

```
  commonJs()],
  output: { /* ... */ }
};
```

Das Bundle enthält direkt den Code von bluebird. Wollen Sie hingegen zum Beispiel jQuery als externe Bibliothek verwenden, brauchen Sie nichts zu installieren und können die folgende Konfiguration verwenden:

```
// rollup.config.js:
export default {
  input: "src/index.js",
  external: ["jquery"],
  output: {
    file: "dist/bundle.js",
    format: "iife",
    name: "myGlobal",
    // für ein IIFE-Bundle müssen Sie angeben, unter
    // welcher globalen Variable jquery gefunden werden
    // kann
    globals: {jquery: "$"}
  }
};
```

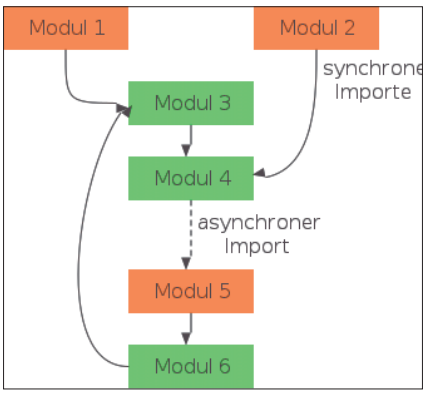
Das Bundle wird dann ein *import $ from "jquery"* durch die Verwendung der globalen Variable *$* ersetzen, für deren Vorhandensein Sie allerdings selbst sorgen müssen.

Eine duale Bibliothek für NPM

Da das ES-Format bessere Optimierungen ermöglicht, gleichzeitig aber (noch) nicht mit Node kompatibel ist, werden viele Bibliotheken inzwischen auf der NPM-Website als „duale Module“ angeboten. Tatsächlich ist Rollup in der Lage, von einem Eingangsdatensatz mit geringem Aufwand mehre Versionen in verschiedenen Formaten zu schreiben. Für maximale Kompatibilität ließe sich dies folgendermaßen aufsetzen:

```
// rollup.config.js:
export default {
  input: "src/index.js",
  output: [
    {
      file: "dist/bundle.mjs",
        // .mjs ist die Kennung für
        // ES-Module in zukünftigen
        // Node-Versionen
      format: "esm"
    },
    {
      file: "dist/bundle.js",
      format: "cjs" // Dies ist das von
        // Node verstandene CommonJS-
        // Format
    }
  ]
};

// package.json:
{
  ...
  main: "dist/bundle",
  module: "dist/bundle.mjs"
}
```



Beispielhafter Modulgraph (Bild 3)

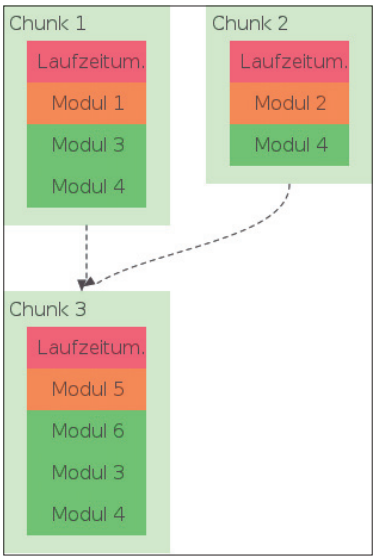
Beachten Sie, dass das *main*-Feld in der Datei *package.json* keine Endung erhält. Dies sorgt dafür, dass zukünftige Node-Versionen je nach Art des Imports immer das richtige Bundle anhand der Namenserweiterung auswählen. Das *module*-Feld hingegen wird speziell von Paketmanagern wie Rollup oder Webpack erkannt, die dann die ES-Version bevorzugt verwenden.

Code-Splitting: einfach mit komplexen Konsequenzen

Es gibt jedoch Gründe, die Modularisierung nicht vollständig aufzugeben. Beispielsweise wollen Sie mehrere Varianten einer Bibliothek anbieten, die allerdings einen großen Teil Ihres Codes gemeinsam haben, den Sie wiederum nicht duplizieren wollen; oder Sie wollen im Browser Codeteile nur bei Bedarf nachladen.

Bild 3 zeigt, wie das aussehen kann. Das Diagramm hat drei Einstiegspunkte: *Modul 1* und *Modul 2* sind explizite Einstiegspunkte, die verschiedene Teile der restlichen Module benötigen. *Modul 5* soll nur bei Bedarf asynchron nachgeladen werden und ist damit auch ein Einstiegspunkt.

Im einfachsten Fall ermöglichen Bundler das Aufteilen von Code, indem sie ihn ausgehend von den Einstiegspunkten in „Chunks“ unterteilen, die alle direkten Importe sowie die Laufzeitumgebung enthalten und damit eigenständig laufen können (Bild 4). Dieses Verfahren kann schnell zu viel dupliziertem Code führen. Wird über einen Chunk ein Modul nachgeladen, das sich bereits im Speicher befindet, so wird dieses Modul ignoriert – Sie haben toten Code geladen.



Einfaches Code-Splitting (Bild 4)

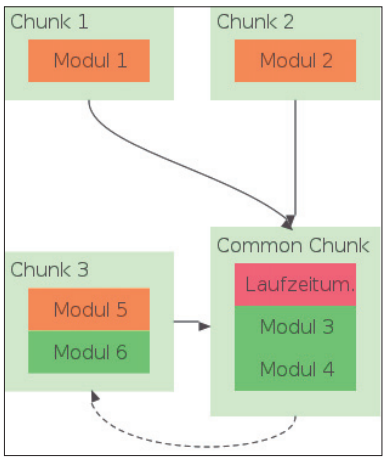
Um dem Problem Herr zu werden, bietet zum Beispiel Webpack die Möglichkeit an, einen „common chunk“ für gemeinsame Abhängigkeiten zu definieren (Bild 5). Wenn dieser auch die Laufzeitumgebung enthalten soll, müssen Sie allerdings immer manuell sicherstellen, dass dieser vor allen anderen Chunks geladen wird.

Im Allgemeinen liefert dies jedoch nicht optimale Ergebnisse: Wenn im Beispiel von Bild 5 nur *Chunk 2* geladen wird, so wird trotzdem *Modul 3* importiert, obwohl es eigentlich nicht benötigt wird. Rollup verfolgt hier einen anderen Ansatz, der diese Probleme nicht hat.

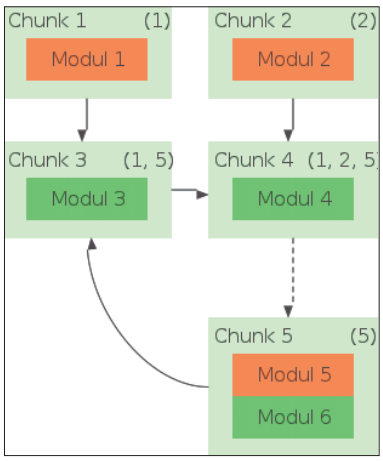
Beispiel: Code-Splitting in Rollup

Da Rollup keine Laufzeitumgebung hat, ist es für Code-Splitting auf das Gastsystem angewiesen, das echte Importe unterstützen muss. Es ist daher nur in den Formaten ES-Modul (*esm*), CommonJS (*cjs*), SystemJS (*system*) und AMD (*amd*) möglich, nicht aber in *umd* oder *iife*. Um sicherzugehen, dass kein Modul doppelt geladen wird, packt Rollup alle Module, die von den gleichen Einstiegspunkten direkt erreicht werden können, in gemeinsame Chunks. In Bild 6 geben die Ziffern in der Ecke die Einstiegspunkte an.

Um dies auszuprobieren, erweitern Sie das Scope-Hoisting-Beispiel von oben um einen zweiten, asynchronen Einstiegspunkt:



Code-Splitting mit Common Chunk (Bild 5)



Code-Splitting, wie es Rollup betreibt (Bild 6)

```
// src/index2.js:
setTimeout(import("./dep1.js")
  .then(value => console.log("index2.js:", value)),
  1000
); // Importiert dep1.js nach 1 Sekunde asynchron
```

Das dynamische *import()* ist ein neues JavaScript-Merkmal, das manche Browser und Rollup bereits unterstützen und das voraussichtlich 2019 Teil des offiziellen Sprachstandards werden wird [14]. Außerdem müssen Sie die Konfiguration anpassen:

```
// rollup.config.js:
export default {
  input: {
    bundle1: "src/index.js",
    bundle2: "src/index2.js"
  },
  experimentalCodeSplitting: true,
    // wird ab Version 1.0 nicht mehr nötig sein
  output: {
    dir: "dist ",
    format: "esm"
  }
};
```

Um den Chunks verschiedene Namen geben zu können, machen Sie aus *input* nun ein Objekt; dafür geben Sie mit *output.dir* hier nur noch ein Zielverzeichnis an. *rollup -c* erzeugt nun tatsächlich drei Dateien: *dist/bundle1.js*, die *index.js* entspricht; *dist/bundle2.js* enthält *index2.js*; und *dist/dep1.js* enthält die verbleibenden Module *dep1.js* und *dep2.js*. Dieser letzte Chunk wurde automatisch erstellt und nach der von den anderen Chunks importierten Datei *dep1.js* benannt, die damit hier der Einstiegspunkt ist.

Geben Sie als Format stattdessen *cjs* an, so ist das Ergebnis direkt in den meisten Node-Versionen ausführbar.

Fazit

Der Code-Bundler Rollup ist ein sehr mächtiges Tool, das dem Entwickler die vollständige Kontrolle über den Code erlaubt, allerdings auch einen gewissen Konfigurationsaufwand erfordern kann. Seine wahre Stärke zeigt das Tool, wenn es an die Optimierung von NPM-Bibliotheken geht, aber auch bei Webanwendungen, bei denen bestes Laufzeitverhalten das Ziel ist. ■

[1] Node.js v10.11.0 Documentation, www.dotnetpro.de/SL1812Rollup1
[2] Browserify, http://browserify.org
[3] Webpack, https://webpack.js.org
[4] Axel Rauschmayer, Exploring ES6, 4.18 From CommonJS modules to ES6 modules, www.dotnetpro.de/SL1812Rollup2
[5] RollupJS, https://rollupjs.org
[6] Wikipedia, Garbage Collection, Mark-and-Sweep-Algorithmus, www.dotnetpro.de/SL1812Rollup3
[7] Rich Harris, Tree-shaking versus dead code elimination, www.dotnetpro.de/SL1812Rollup4
[8] rollup-plugin-alias, www.dotnetpro.de/SL1812Rollup5
[9] RollupJS, Online-Konsole, https://rollupjs.org/repl
[10] RequireJS, https://requirejs.org
[11] rollup-plugin-node-resolve, www.dotnetpro.de/SL1812Rollup6
[12] rollup-plugin-commonjs, www.dotnetpro.de/SL1812Rollup7
[13] bluebird, http://bluebirdjs.com
[14] Can I use, JavaScript modules: dynamic import(), www.dotnetpro.de/SL1812Rollup8

Lukas Taegert
hat Mathematik studiert und über mathematische Eigenschaften Schwarzer Löcher promoviert. Heute ist er Fullstack-Entwickler bei TNG und auch einer der Maintainer und treibenden Entwickler des Tools Rollup.
lukas.taegert@tngtech.com

dnpCode A1812Rollup