Hilfe, ich muss JavaScript programmieren!

Nils Hartmann

Wer professionelle Web-Anwendungen bauen will, kommt an JavaScript nicht vorbei. Für viele Java-Entwickler ist die Vorstellung, mit JavaScript programmieren zu müssen, immer noch ein Graus. Zum einen bringt die Sprache selber einige Fallstricke und Kuriositäten mit; darüber hinaus erscheint auch das Ökosystem, das um die Sprache herum existiert, also Frameworks, Libraries und Tools, unüberschaubar groß und wenig nachhaltig. Der richtige Einstieg ist damit meist sehr schwierig. Dieser Artikel gibt eine Orientierung, indem die wichtigsten Werkzeuge vorgestellt und eingeordnet werden.

Das JavaScript-Ökosystem ist sehr groß und schnelllebig. Das macht es für Neulinge nicht einfach; selbst erfahrene JavaScript-Entwickler haben Mühe, darin die Orientierung zu behalten und mit der sehr schnellen Weiterentwicklung Schritt zu halten. Das hat sich in den vergangenen Jahren in dem Ausdruck „JavaScript Fatique", also „JavaScript-Erschöpfung", manifestiert.

Bevor wir uns auf den Weg durch das Ökosystem machen und ansehen, welche Tools wir wofür brauchen, zunächst die Frage, warum es sich überhaupt lohnt, sich mit JavaScript zu beschäftigen. Können wir nicht einfach unsere bekannten Verfahren wie Spring MVC oder JEE einsetzen und Web-Anwendungen bauen, die mithilfe einer Template-Sprache auf dem Server gerendert werden? Hier kennen wir immerhin die Sprache, die Frameworks sowie die Tools samt ihren Stärken und Schwächen.

Die Antwort ist mehrschichtig. Wir möchten dem Anwender den bestmöglichen Nutzungskomfort bieten, im Optimalfall sollte sich eine Web-Anwendung nicht von einer Desktopanwendung unterscheiden. Insbesondere soll die Anwendung sehr schnell auf Benutzer-Interaktionen reagieren, auf diversen Geräten mit unterschiedlichen Betriebssystemen gut aussehen und am besten auch dann noch funktionieren, wenn die Internet-Verbindung gerade nicht besteht. Moderne Web-Anwendungen wie zum Beispiel Spotify *(siehe „https://open.spotify.com")*, Outlook *(siehe „http://www.outlook.com")* oder das Prototype-Tool Figma *(siehe „https://figma.com")* erfüllen diese Ansprüche. Solche Anwendungen sind jedoch nur denkbar, wenn sie direkt auf dem Client, also im Browser, ausgeführt werden *(siehe Abbildung 1)*.

Die Sprache im Browser ist JavaScript. Wenn wir also Anwendungen bauen wollen, die im Browser laufen, müssen wir uns wohl oder übel damit beschäftigen. Im Übrigen wird der Browser zunehmend auch für Anwendungen interessant, die früher klassischerweise auf dem Desktop liefen. Die genannten Anwendungen sind ein Beispiel dafür, aber auch In-House-Anwendungen werden immer häufiger als Web-Anwendung implementiert. Auch hier ist es sinnvoll, sich mit JavaScript als der Sprache im Browser zu beschäftigen, da auch Java-basierte Ansätze wie JSF oder Vaadin nicht ohne JavaScript auskommen. Die direkte Verwendung von JavaScript kann sogar einfacher sein, als eine Abstraktion zu benutzen.

Das JavaScript-Ökosystem

Bei Java gibt es ein komplettes, offizielles Entwickler-Kit, das JDK. Es bringt eine ganze Reihe von Bibliotheken und Tools mit, die alle aufeinander abgestimmt sind. Dazu gehören der Compiler, die Laufzeitumgebung (JRE), sehr umfangreiche Standard-Bibliotheken und Tools wie „javadoc" und „apt", sogar eine SQL-Datenbank. Das alles existiert bei JavaScript zunächst nicht. JavaScript ist vorerst nur eine Sprache; alles andere – inklusive der Laufzeit-Umgebung, also in erster Linie der Browser – müssen wir uns selbst zusammenstellen oder sogar selbst entwickeln.

Noch vor wenigen Jahren reichte für die JavaScript-Entwicklung ein einfacher Editor aus. Man schrieb seinen Code in eine Datei, bettete diese in eine HTML-Seite ein und ließ sie vom Browser ausführen. Dieses Vorgehen eignete sich jedoch nur für eher kleinere Features, etwa das Validieren von Eingaben auf einer Webseite. Größere Anwendungen waren damit nicht oder nur sehr schwer umsetzbar. Aus dieser Situation heraus entwickelten sich unter anderem Libraries wie jQuery, die bestimmte Probleme adressierten und lösten.

Im Laufe der Zeit wurden die Anforderungen an Webseiten immer komplexer, aus ehemals statischen Seiten wurden dynamische Seiten mit immer mehr Benutzer-Interaktionen: Echte Anwendungen im Browser entstanden. Dafür reichte jQuery irgendwann nicht mehr aus; es entstanden neue Lösungen, die wiederum neue Innovationen hervorriefen, diese abermals neue Lösungen, und so weiter. Die entstandenen Lösungen wurden aber, von der Sprache JavaScript abgesehen, nicht – wie bei Java – von einem Konsortium (mit)entwickelt, spezifiziert und standardisiert, sondern entstanden und entstehen weiterhin durch die Community. So sind auch alle im Folgenden vorgestellten Tools und Frameworks Open-Source-Lösungen.

Die Sprache JavaScript

Ähnlich wie in Java gibt es mehrere Versionen der Sprach-Spezifikation, die „ECMAScript" heißt. Von den meisten Browsern *(siehe „https://caniuse.com/#search=es5")* ist die Version 5 implementiert; wenn man eine Anwendung also nach dieser Spezifikation implementiert, ist die Chance sehr hoch, dass sie in allen Browsern auch funktioniert. ECMAScript 5 wurde im Jahre 2009 veröffentlicht, danach gab es erst im Jahr 2015 die nächste Version, die aber immer noch nicht von allen Browsern vollständig implementiert ist.

Allerdings enthält Version „ECMAScript 2015" sehr viele, teilweise auch gravierende Änderungen und Neuerungen, die für die Entwicklung von Anwendungen sehr vorteilhaft sein können; so haben beispielsweise Klassen und Module Einzug in die Sprache gehalten und mit der Einführung von Block-Scoping sowie Arrow-Funktionen wurde auf einige der gravierendsten Unzulänglichkeiten in der Sprache reagiert.

Darüber hinaus wurde mit der Version auch der Release-Zyklus für die Sprache verändert: Seit dem Jahr 2015 gibt es nun jährliche Releases, die jeweils nach dem Erscheinungsjahr benannt und nicht mehr durchnummeriert sind, also „ECMAScript 2015", „ECMAScript 2016", „ECMAScript 2017" etc.

Wie erwähnt, betreffen diese Releases nur die Sprach-Spezifikation, nicht deren Umsetzungen in den Browsern. Zwar ist zu beobachten, dass die Browserhersteller bemüht sind, die neuesten Spezifikationen schnell umzusetzen, in der Praxis dauert das jedoch immer einige Zeit. Außerdem lässt sich eine Anwendung mit neuem JavaScript nur dann sicher betreiben, wenn wirklich alle für die Zielgruppe relevanten Browser die verwendete ECMAScript-Version unterstützen.

Um dieses Problem zu lösen, haben sich in der JavaScript-Entwicklung Compiler etabliert. Damit wird allerdings nicht wie in Java Source-Code zu Byte-Code übersetzt, sondern JavaScript-Code aus einer Version in eine ältere zurückkompiliert. So lassen sich neue Features zeitnah nutzen, indem diese in beispielsweise ECMAScript 5 zurückkompiliert werden, das die allermeisten Browser unterstützen.

Compiler und Polyfills

Der wohl am meisten eingesetzte Compiler ist Babel *(siehe „https://babeljs.io")*. Er kann nicht nur aktuellen JavaScript-Code in ältere Ver-

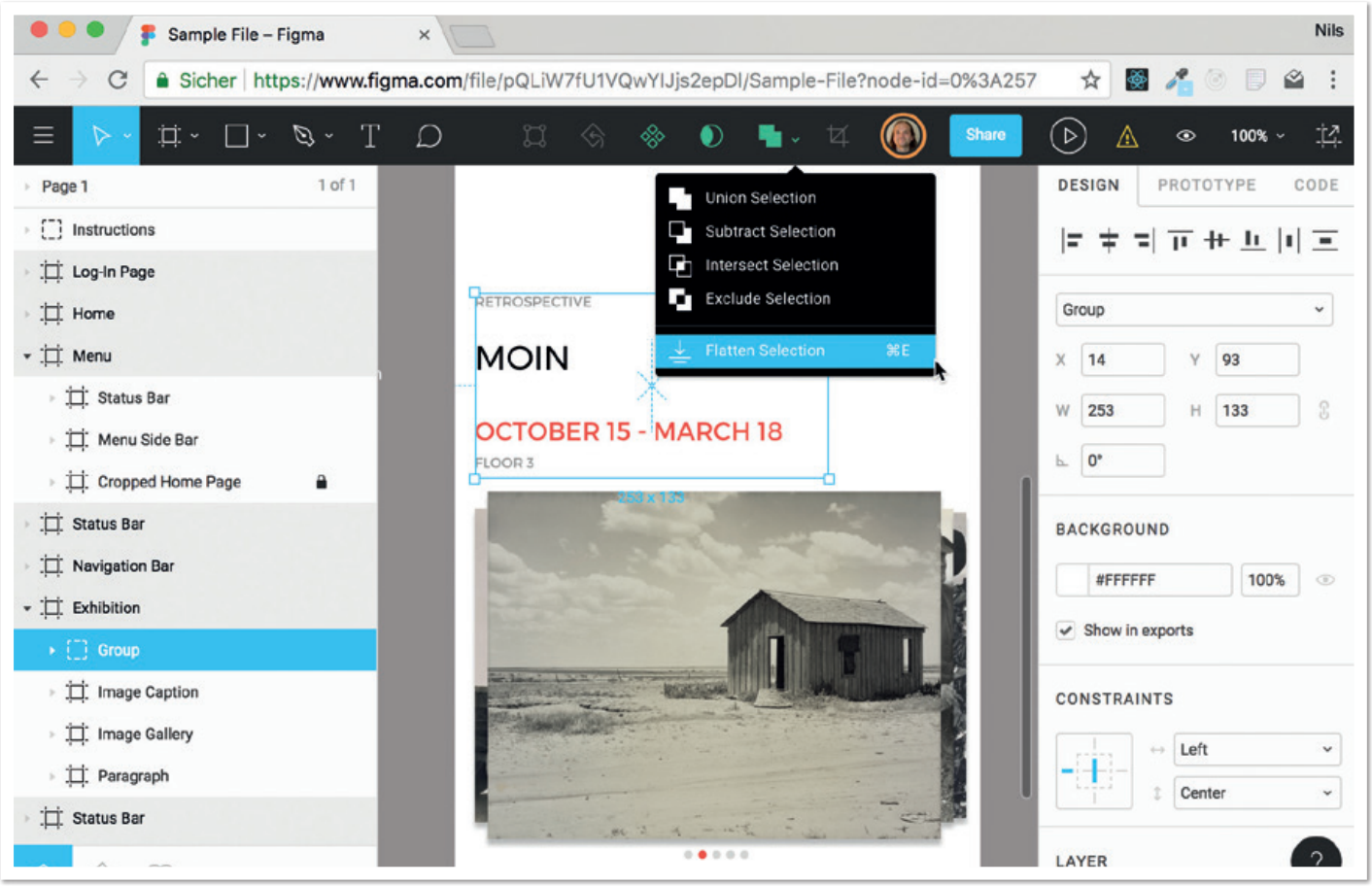

Abbildung 1: Figma, eine echte Anwendung im Browser

sionen zurückübersetzen, sondern dank eines modularen Aufbaus mit Plug-ins um weitere Features ergänzt und für den individuellen Bedarf konfiguriert werden. Oftmals stehen beispielsweise Plug-ins schon für zukünftige JavaScript-Features zur Verfügung, die noch gar nicht endgültig spezifiziert worden sind. Mittels Babel lassen sie sich jedoch bereits vorab in eigenen Projekten testen. Eine Alternative zu Babel ist der TypeScript-Compiler, der von Microsoft entwickelt wird. Er wird später noch näher betrachtet.

Ein Compiler übersetzt nur die Sprache selbst von einer Version zu einer anderen Version zurück. Daneben gibt es allerdings noch einige Standard-APIs, die ebenfalls zur Spezifikation gehören und von der Laufzeit-Umgebung zur Verfügung gestellt werden müssen. Dazu gehören beispielsweise die Funktionen, die auf „Object", „String" und „Number" definiert sind, die (neuen) Maps und Sets oder auch das DOM-API. Hier ergibt sich dasselbe Problem wie bei der Sprache: Fehlen die APIs in einem der gewünschten Ziel-Browser, können sie in der eigenen Anwendung nicht verwendet werden, da es ansonsten zur Laufzeitfehlern kommt.

Abhilfe an dieser Stelle schaffen Polyfills, also Bibliotheken, um eine oder mehrere der Standard-APIs zu implementieren und in die eigene Anwendung einzubinden. Die meisten Polyfills sind so intelligent, dass sie sich selbst nur dann aktivieren, wenn die von ihnen implementierten APIs zur Laufzeit wirklich nicht vorhanden sind. Wenn das API in einem Browser bereits verfügbar ist, wird dann von der Anwendung automatisch die nativ implementierte Variante verwendet. Polyfills für diverse APIs stellt beispielsweise „corejs" *(siehe „https://github.com/zloirock/core-js")* zur Verfügung. Dabei lässt sich auch sehr feingranular auswählen, für welche konkreten APIs ein Polyfill benötigt wird, um den Anwendungscode nicht unnötig aufzublähen. Auch Babel bietet eine Polyfill-Bibliothek *(siehe „https://babeljs.io/docs/usage/polyfill")*.

Laufzeit-Umgebungen

Bislang wurden als Laufzeit-Umgebung nur Browser betrachtet. Mittlerweile werden mit JavaScript auch Applikationen gebaut, die außerhalb eines Browsers laufen. Hier kommt dann in der Regel NodeJS zum Einsatz. Es verwendet die JavaScript-Implementierung V8 von Google, die auch die JavaScript-Engine von Chrome ist. Seit Anfang 2016 gibt es daneben eine NodeJS-Variante, die mit Chakra Core, der JavaScript-Engine von Microsoft Edge, arbeitet. Auf Basis von NodeJS werden einerseits Serverprozesse (zum Beispiel Webserver oder für ein REST-API) implementiert. Andererseits dient NodeJS auch als Ausführungsumgebung für eine ganze Reihe von Tools wie Compiler (die genannten Compiler Babel und TypeScript sind in JavaScript implementiert), Package-Manager oder Test-Tools. Das ist in gewisser Weise mit Java vergleichbar, wo die JRE auch nicht nur zum Ausführen von Server-Prozessen verwendet wird.

Modul-Systeme

Unabhängig von der späteren Laufzeit-Umgebung müssen gerade größere Anwendungen gut strukturiert sein, damit der Code verständlich und wartbar bleibt. Eine Möglichkeit dafür bieten Module. Ein Modul in JavaScript ist technisch betrachtet eine Datei und damit eine sehr viel kleinere Einheit als die meisten Java-Module, die in der Regel aus mehreren Klassen und Packages bestehen. In

JavaScript ist es nicht ungewöhnlich, dass Module nur aus einer oder zwei Funktionen bestehen.

Im Laufe der Zeit haben sich insgesamt drei Modul-Systeme etabliert: CommonJS, das auch Grundlage für das in NodeJS verwendete Modul-System und für den Einsatz außerhalb eines Browser konzipiert ist; außerdem das AMD-Modul-System, das für den Einsatz im Browser ausgelegt und unter anderen in der Lage ist, Module asynchron nachzuladen, um beispielsweise den Start einer Anwendung zu beschleunigen. Mit ECMAScript 2015 wurde nun auch ein natives Modul-System spezifiziert und in den Sprachstandard aufgenommen. Es wird zunehmend von den Browsern unterstützt und auch von NodeJS implementiert werden. Bis das Modul-System überall unterstützt ist, können Babel oder TypeScript verwendet werden, um Code, der das native System verwendet, nach CommonJS oder AMD zu übersetzen.

Mit dem nativen Modul-System lassen sich aus Modulen (also Dateien) einzelne Bestandteile exportieren und diese damit für andere Module sichtbar machen. Alle Bestandteile eines Moduls (Funktionen, Klassen, Konstanten etc.), die nicht explizit exportiert werden, bleiben für andere Module unsichtbar. Um etwas aus einem anderen Modul zu importieren, ist es explizit zu importieren. *Listing 1* zeigt ein Beispiel: Das „UserService"-Modul exportiert eine Klasse „UserService", die vom App-Modul importiert wird. Alle anderen Teile aus dem Modul (die Konstante „database" und die Funktion „initDatabase") sind für andere Module nicht sichtbar und können auch nicht importiert werden.

```
// UserService.js
export default class UserService {
  constructor() { ... }
  loadUser(id) { ... }
}
// Nur Modul-intern sichtbar
const database = ...;
function initDatabase() { ... }

// App.js
import UserService from "./UserService";
const userService = new UserService();
userService.loadUser(1);
```

Listing 1

Package-Manager

Module lassen sich (ähnlich wie mit Maven in Java) in einem zentralen Repository veröffentlichen und von dort auch installieren. Das dafür zuständige Tool ist der „Node Package Manager" (npm, siehe *„https://www.npmjs.com"),* der Bestandteil der NodeJS-Distribution ist, aber auch einzeln installiert werden kann. Das Repository ist die „npm"-Registry. Die dort veröffentlichten Artefakte heißen „Packages" und können aus mehreren Modulen und weiteren Source-Artefakten, etwa CSS-Dateien, bestehen. Packages, von denen das eigene Projekt abhängig ist, werden im Projekt in einer Beschreibungsdatei („package.json") hinterlegt, vergleichbar mit den Abhängigkeiten in der „pom.xml" von Maven.

In JavaScript-Projekten ist es üblich, über die Abhängigkeitsverwaltung die für die Entwicklung benötigten Tools zu installieren. So stehen beispielsweise Babel und TypeScript, aber auch andere Build-Tools, als „npm"-Pakete zur Verfügung. Auf diese Weise kann jedes Projekt individuell festlegen, welche Tools in genau welcher Version erforderlich sind. Alle Mitglieder des Entwicklungsteams erhalten dann automatisch die benötigten Tools in den korrekten Versionen. Außerdem entfällt weitgehend die Notwendigkeit, Tools überhaupt global zu installieren und dadurch in Versionskonflikte zu geraten. *Listing 2* zeigt einen Ausschnitt aus einer „package.json"-Datei, in der sowohl Abhängigkeiten definiert sind, die von der Anwendung selbst benötigt werden („dependencies"), als auch Abhängigkeiten, die nur für die Entwicklung notwendig sind („devDependencies").

Für (In-House-)Module, die nicht in einem öffentlichen Repository veröffentlicht werden sollen, kann man neben der offiziellen „npm"-Registry auch eigene Repositories betreiben. Sowohl Nexus als auch

```
{
  "name": "my-javascript-app",
  "version": "1.0.0",
  "devDependencies": {
    "node-sass": "^4.7.2",
    "prettier": "^1.11.1",
    "tslint": "5.8.0",
    "typescript": "^2.7.2"
  },
  "dependencies": {
    "react": "^16.3.1",
    "react-dom": "^16.3.1",
    "react-router": "^4.2.0",
    "react-router-dom": "^4.2.2"
  }
}
```

Listing 2

```
// Type Inference: age ist eine Nummer
const age = 32;

// Explizite Typ Angabe
const year:number = 2018;

// Funktionsparameter
function greet(phrase: string) {
  ...
}

// Objekte
interface Person {
  name: string,
  age: number
}

const klaus:Person = {
  name: "Klaus",
  age: 32
}
```

Listing 3: Typ-Angaben in TypeScript

Artifactory beispielsweise unterstützen neben diversen anderen Formaten auch das „npm"-Format.

Neben „npm" zum Installieren und Deployen von Packages gibt es mit dem von Facebook entwickelten „yarn" *(siehe „https://yarnpkg.com")* eine Alternative. Es nutzt dieselbe Abhängigkeitsbeschreibung in der „package.json" wie „npm" und kann außerdem auch das gleiche Registry verwenden. Vom Funktionsumfang her unterscheiden sich die beiden Tools mittlerweile nicht mehr wesentlich, allerdings sind sie hinsichtlich der Bedienung verschieden.

Der Vollständigkeit halber sei erwähnt, dass aus früheren Frontend-Projekten noch „bower" *(siehe „https://bower.io")* bekannt ist, mit dem sich ebenfalls Abhängigkeiten verwalten lassen. Dieses Tool wird zwar noch gepflegt, die Entwickler raten jedoch selber dazu, stattdessen auf „npm"/„yarn" und Webpack (siehe unten) zu setzen.

Module-Bundler

Wir haben nun einen Eindruck davon bekommen, wie sich mit JavaScript Module erstellen, in Form von Packages veröffentlichen und in ein eigenes Projekt einbinden lassen. Zum Ausführen einer Anwendung im Browser reicht das allerdings noch nicht aus, denn die Browser unterstützen beispielsweise das NodeJS-Modulsystem nicht und auch das native Modulsystem wird noch nicht von allen Browsern unterstützt.

Wenn eine Anwendung also selber das NodeJS-Modulsystem oder mittels „npm" ein externes Package verwendet, das mit dem NodeJS-Modulsystem gebaut ist, brauchen wir dafür eine Lösung. An dieser Stelle kommt ein Module-Bundler ins Spiel, ein Tool, das die Abhängigkeiten einer Anwendung zur Laufzeit analysiert und aus allen referenzierten Modulen eine vom Browser verwendbare JavaScript-Datei erstellt. Der Bundler ist dabei so schlau, dass er die Sichtbarkeiten und Scopings der Original-Module berücksichtigt, sodass es in der erzeugten Ausgabe-Datei nicht zu Namenskollisionen kommen kann.

Der wohl prominenteste Module-Bunder ist Webpack *(siehe „https://webpack.js.org")*. Er kann mit allen drei Modul-Systemen umgehen und bietet diverse Optimierungsmöglichkeiten für den erzeugten Code. Webpack selber ist modular aufgebaut und lässt sich für das eigene Projekt sehr gut anpassen. So kann Webpack beispielsweise vor dem Erzeugen des „Bundle", also der fertigen Ausgabedatei, die eingelesenen Module zuvor noch mit Babel oder TypeScript kompilieren. Dabei ist Webpack nicht auf JavaScript beschränkt, sondern kann zum Beispiel auch mit CSS-Dateien umgehen. Wenn diese im Code referenziert werden, findet Webpack die Referenzen und kann die Dateien je nach Konfiguration behandeln, zum Beispiel den CSS-Code komprimieren.

Auf Wunsch erzeugt Webpack auch immer SourceMaps. Dabei handelt es sich um Informationen für den Browser, die den kompilierten Code zurück auf den Source-Code mappen, sodass der Browser im Debugger den originalen Code anzeigen kann, den wir selber programmiert haben. Auch dieses Verhalten ist ähnlich wie in Java, wo im Debugger zum Beispiel in der IDE ja ebenfalls der Source-Code und nicht der erzeugte Byte-Code angezeigt wird. Andere bekannte Module-Bundler neben Webpack sind Browserify *(siehe „http://browserify.org")* und Rollup *(siehe „https://rollupjs.org")*.

Testen und Qualitätssicherung

Mit den bis hierher gezeigten Werkzeugen lassen sich Anwendungen bauen, die entweder im Browser oder mit NodeJS auch außerhalb des Browsers laufen können. Genau wie in Java greift man dabei auf externe Bibliotheken („npm"-Packages) zu und legt auch eigene Packages in einer Registry ab.

Eine erste Möglichkeit, die Anwendung mit Tests und anderen Maßnahmen gegen Fehler abzusichern, besteht darin, einen Type-Checker zu verwenden, der dabei helfen kann, eine ganze Reihe typischer JavaScript-Fehler zu vermeiden, so wie wir das auch aus Java gewohnt sind (eine Zahl keinem String zuweisen, falsche Anzahl und/oder Typen von Parametern einer Funktion übergeben, unsichere Zugriffe auf potentielle Null-Werte etc.). Prominente Type-Checker sind Flow *(siehe „https://flow.org")* von Facebook oder TypeScript *(siehe „http://typescriptlang.org")* von Microsoft. Beiden gemeinsam ist, dass die Angabe von Typen optional ist, man muss also Typ-Angaben nur dort hinschreiben, wo man sie auch wirklich benötigt. An vielen Stellen leiten die Type-Checker die korrekten Typen automatisch ab.

Während Flow ausschließlich ein Type-Checker ist, handelt es sich bei TypeScript um eine eigene Sprache, die allerdings auf JavaScript aufbaut. So ist jeder gültige JavaScript-Code zunächst auch gültiger TypeScript-Code. Mit TypeScript lässt sich der Code um Typ-Angaben ergänzen, zudem fügt TypeScript der Sprache auch noch einige neue Features hinzu, wie Sichtbarkeiten („protected" und „private" in Klassen) oder Aufzählungstypen („enum").

Insgesamt ist TypeScript sehr weit verbreitet – unter anderem wurden Angular und auch die neue Outlook-Webanwendung damit entwickelt – und es gibt mittlerweile Typ-Beschreibungen für fast alle bestehenden JavaScript-Bibliotheken. Auch der DIE-Support ist insbesondere mit der IntelliJ-Produktfamilie und Visual Studio Code sehr gut. Aus diesem Grunde setzt der Autor in Projekten eher TypeScript als Flow ein. *Listing 3* zeigt beispielhaft, wie Typ-Informationen in TypeScript geschrieben werden (die Syntax sieht in Flow übrigens sehr ähnlich aus), und in *Abbildung 2* ist exemplarisch zu sehen, wie in Visual Studio Code von TypeScript gefundene Fehler angezeigt werden.

Alternativ oder zusätzlich zu einem Type-Checker lassen sich sogenannte „Linter" zur Sicherstellung der Code-Qualität einsetzen. Dabei handelt es sich um Tools, die mit statischer Code-Analyse typische Probleme aufdecken. Außerdem lassen sich darüber selbst definierte Code-Konventionen überprüfen. Der am häufigsten eingesetzte Linter ist „ESLint" *(siehe „https://eslint.org")* beziehungsweise „TSLint" *(siehe „https://palantir.github.io/tslint")* für TypeScript-Code. Beide Linter lassen sich auch im Rahmen des Webpack-Builds ausführen, sodass Webpack abbricht, wenn eines der Tools ein Problem meldet *(siehe Abbildung 3)*. Zum Einhalten identischer Code-Konventionen eignet sich übrigens das Formatierungstool „Prettier" *(siehe „https://prettier.io")* hervorragend.

Type-Checker und Linter ersetzen allerdings keine Tests für die eigene Anwendung und genau wie in Java gibt es auch für JavaScript-Anwendungen diverse Test-Tools und -Frameworks für Tests auf allen möglichen Ebenen. Leider existiert aus Sicht des Autors zurzeit kein Standard, was die Auswahl der Test-Frameworks angeht. Im Wesentlichen gibt es drei Optionen: Mocha *(siehe „https://mochajs.org")*, Jasmine *(siehe „https://jasmine.github.io")* und Jest *(siehe „https://facebook.github.io/jest")*.

Mocha ist nur ein Test-Runner, der es erlaubt, einzelne Tests und Testsuites zu definieren und auszuführen. Für Dinge wie Assertions oder Mocks sind allerdings zusätzliche, nach eigenen Vorlieben frei auswählbare Module zu installieren. Jasmine und Jest hingegen bringen diese Dinge von Haus aus schon mit („Batteries included"), Jest bietet sogar Code Coverage und eine „headless"-Implementie-

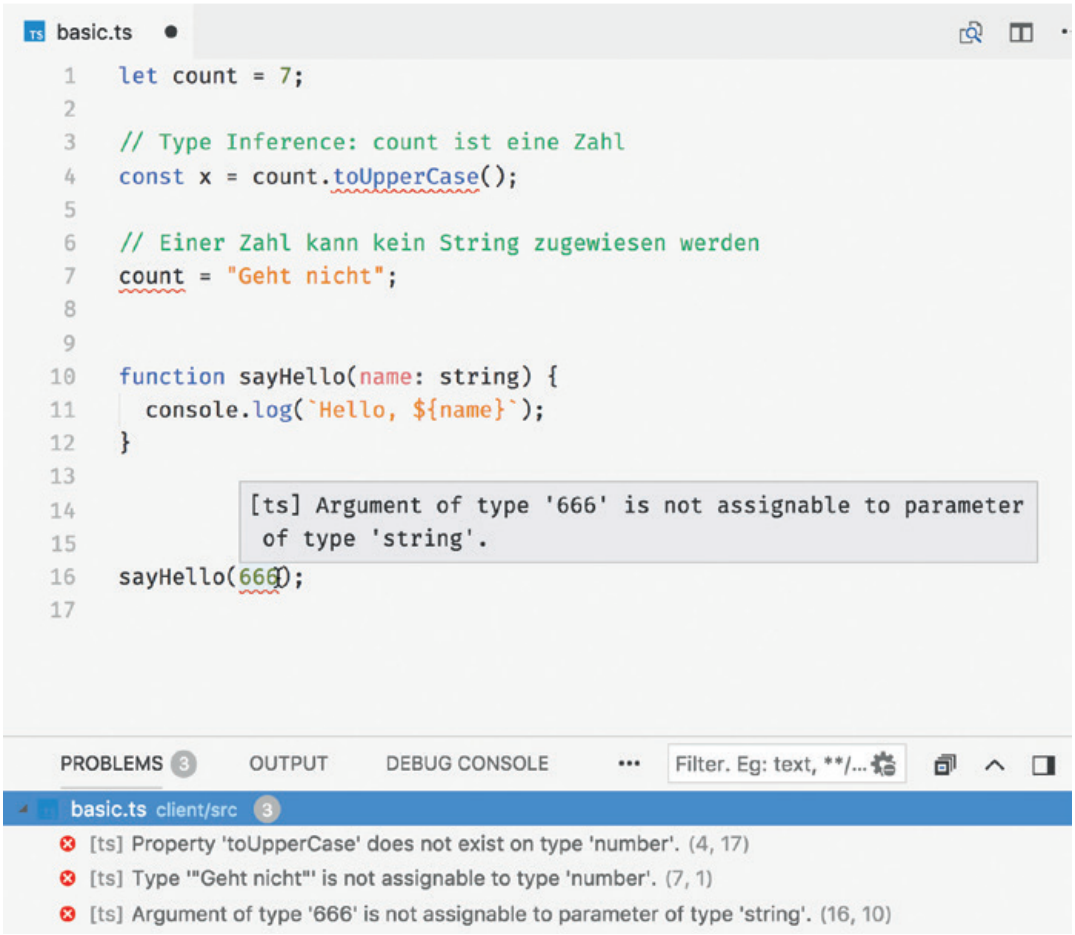

Abbildung 2: TypeScript in der IDE

Abbildung 3: ESLint findet mögliche Probleme im Source-Code

rung eines DOM, sodass man dort auch Code testen kann, der zur Laufzeit das DOM-API benötigt. Mit dem „Snapshot Testing" bringt Jest außerdem eine interessante Möglichkeit mit, JSON-basierte APIs (und auch React-Komponenten) zu testen. *Listing 4* zeigt einen einfachen Unit-Test mit Jest.

Automatisierung

Mit dem bis hier gezeigten Technologie-Stack lassen sich echte Anwendungen entwickeln, testen und betreiben. Gerade zur Entwicklungszeit sind allerdings einige Dinge immer wieder durchzuführen, insbesondere muss der Compiler und/oder Bundler ausgeführt werden, die Tests müssen laufen und eventuell auch der Linter. Das möchten wir natürlich nicht immer und immer wieder von Hand machen, sondern Tools zur Automatisierung dieser Vorgänge nutzen. Zum einen gibt es mit Gulp *(siehe „https://gulpjs.com")* und Grunt *(siehe „https://gruntjs.com")* zwei Tools, die wiederkehrende Tasks ausführen können. Dazu werden Tasks definiert, die bestimmte Aufgaben

```
// sum.js
export function sum(a,b) {
  return a+b
}

// sum.test.js
import {sum} from '../sum.js';

test('sum of 2 and 2 is 4', function() {
  expect(sum(2, 2)).toBe(4);
});

test('sum of 2 and 2 is not 3', function() {
  expect(sum(2, 2)).not.toBe(3);
});
```

Listing 4: Unit-Tests mit Jest

übernehmen (zum Beispiel das Ausführen des Compilers) sowie deren Auslöser (Änderung im Source-Verzeichnis) und Abhängigkeiten untereinander (nach dem Kompilieren den Linter ausführen). Damit lassen sich ähnlich wie mit Maven oder Gradle auch komplexe Build-Prozesse umsetzen.

Eine andere, leichtgewichtigere, meist aber ausreichende Alternative sind die „npm scripts". In der „package.json"-Datei eines Projekts lassen sich neben den Abhängigkeiten auch Skripte hinterlegen, die dann mit „npm" über die Betriebssystem-spezifische Kommandozeile (wie Bash) ausgeführt werden können. Das Besondere daran ist, dass dabei sämtliche lokal von „npm" installierten Tools (wie TypeScript oder Webpack) automatisch im Path vorhanden sind und somit einfach aufgerufen werden können. In der Praxis spielt das Problem, dass die Skripte Betriebssystem-abhängig sind, selten eine Rolle und für häufige Probleme, wie das Setzen von Umgebungsvariablen, gibt es bereits auch fertige Lösungen. Bevor man in seinem JavaScript-Projekt mit Gulp oder Grunt arbeitet, sollte man auf jeden Fall prüfen, ob „npm scripts" ausreichend sind.

Zusammenfassung

Damit sind wir nun am Ende unseres Wegs angekommen. Wir haben gesehen, dass es einen funktionierenden Technologie-Stack für die JavaScript-Entwicklung gibt, mit dem sich auch ernsthafte Anwendungen professionell entwickeln lassen:

- Die Sprache JavaScript wurde im Jahr 2015 um viele wichtige Features, insbesondere ein Modul-System, erweitert.
- Module beziehungsweise Packages können mit „npm" veröffentlicht und eingebunden werden.
- Um Anwendungen, die mit neueren Sprachversionen geschrieben sind, in älteren Browsern ausführbar zu machen, verwenden wir einen Compiler (Babel oder TypeScript) und Polyfills.
- Für unterschiedliche Modul-Systeme gibt es Bundles (Webpack), die aus einer modularisierten Code-Basis eine für den Browser ausführbare JavaScript-Datei erstellen.
- Zur Erfüllung von Qualitäts-Anforderungen, Wartbarkeit und Tests gibt es Type-Checker (Flow und TypeScript), statische Code-Analyse-Tools (Linter) und natürlich Test-Frameworks (Mocha, Jasmine, Jest).
- Um die während der Entwicklung anfallenden Aufgaben zu automatisieren, lassen sich Task-Runner einsetzen, deren einfachste Form die „npm scripts" sind.

Im Gegensatz zum JDK gibt es allerdings keine zentrale Instanz, die Tools und Frameworks für uns auswählt und als fertiges Development Kit bereitstellt. Stattdessen werden die meisten Lösungen auf Eigeninitiative entwickelt und bereitgestellt – und dann von der Community angenommen, verworfen oder weiterentwickelt. Das bedeutet, dass wir einerseits eine sehr hohe Innovationsgeschwindigkeit haben, die das Arbeiten mit JavaScript sehr spannend macht, weil wir immer mehr Dinge mit JavaScript umsetzen können. Auf der anderen Seite kann das Tempo jedoch auch nervenaufreibend sein, wenn man versuchen will, Schritt zu halten.

Aus Sicht des Autors sollte man nicht jedem Trend hinterherrennen. Es ist völlig legitim, beim Erscheinen neuer Frameworks, Tools und Bibliotheken erst einmal abzuwarten, wie diese sich entwickeln, und vor allem zu verstehen, welche konkreten Probleme sie lösen. Denn nur wenn das Problem verstanden wurde, kann geprüft werden, ob es für das eigene Projekt überhaupt relevant ist.

Nils Hartmann
nils@nilshartmann.net

Nils Hartmann ist Software-Entwickler aus Hamburg. Er programmiert sowohl in Java als auch in JavaScript/TypeScript und beschäftigt sich zurzeit hauptsächlich mit der Entwicklung von React-Anwendungen. Nils Hartmann gibt seine Erfahrungen, Ideen und Fragen gern auf Konferenzen, Trainings und Workshops wieder.