IN DIESEM KAPITEL

- erfahren Sie, wie JavaScript entstanden ist und was es mit dem ECMAScript-Standard auf sich hat.
- lernen Sie die wichtigsten Elemente kennen, die Sie zum Entwickeln benötigen.
- sehen Sie, wie clientseitiges JavaScript im Browser funktioniert.
- machen Sie die ersten Schritte, wenn es um serverseitiges JavaScript geht.

Kapitel 1
Auf den Spuren von JavaScript: Zwischen Browser-Zauberei und Server-Magie

JavaScript ist nahezu überall verfügbar, von seiner ursprünglichen Umgebung, dem Browser, hat sich die Programmiersprache mittlerweile in alle Bereiche des täglichen Lebens verbreitet. So können Sie als JavaScript-EntwicklerIn nicht nur dynamische Web-Frontends programmieren, sondern auch die zugehörigen Backends serverseitig mit JavaScript umsetzen. Die Sprache ist jedoch auch in Umgebungen präsent, von denen Sie wahrscheinlich im ersten Moment nicht erwarten, dass dort eine unscheinbare Web-Skriptsprache ausgeführt wird. So finden Sie JavaScript in Autos, Küchengeräten, Industrieanlagen und sogar im Weltall. Das Integrated Science Instrument Module des James-Webb-Weltraumteleskops wird beispielsweise mit JavaScript kontrolliert (`https://www.jwst.nasa.gov/resources/ISIMmanuscript.pdf`). Auch das Weltraumunternehmen SpaceX nutzt JavaScript für seine Raumflüge. Interessant ist hier nicht nur, dass die Crew Dragon per Touchdisplay ins Weltall gesteuert wird, sondern dass deren UI-Elemente auf einer JavaScript-Bibliothek basieren (`https://www.theverge.com/2020/5/30/21275753/nasa-spacex-astronauts-fly-crew-dragon-touchscreen-controls`).

Sie sehen also, es gibt schlechtere Alternativen, wenn es um Programmiersprachen geht, als JavaScript. Die Sprache ist verhältnismäßig einfach zu lernen, weitverbreitet und hat eine

sehr starke Community, die immer wieder neue bahnbrechende Entwicklungen hervorbringt. Außerdem ist JavaScript ein Industriestandard. JavaScript ist der weltweit verbreitete Name der Programmiersprache ECMAScript, die im Standard ECMA-262 definiert wird. Diesen finden Sie unter `https://www.ecma-international.org/publications-and-standards/standards/ecma-262/`. Für JavaScript gibt es außerdem einen ISO-Standard mit der Bezeichnung ISO/IEC-16262, der dem ECMA-Standard entspricht. Die JavaScript-Version aus dem Jahr 2022 würde, wenn Sie die Definition ausdrucken würden, 809 Seiten in Anspruch nehmen. Dabei handelt es sich nur um den Sprachkern. Erweiterungen wie die DOM-API sind hier noch nicht mit inbegriffen. Das ECMAScript-Dokument ist auch nichts, was Sie an einem verregneten Wochenende zum Zeitvertreib lesen sollten, da es sich beim überwiegenden Teil des Texts um formale Beschreibungen der Sprache handelt.

Doch warum heißt JavaScript nicht auch offiziell JavaScript? Die Antwort auf diese Frage ist einfach: Die Markenrechte an JavaScript liegen bei Oracle. Solange das Unternehmen den Namen nicht freigibt, ist es nicht möglich, die Programmiersprache offiziell als JavaScript zu bezeichnen. Die Standardisierung und Organisation klingt zugegebenermaßen etwas trocken und angestaubt, ist für uns als JavaScript-EntwicklerInnen aber von großer Bedeutung, da sich die Hersteller von JavaScript-Engines überwiegend an diesen Standard halten und es so mittlerweile nur noch in geringem Umfang zu Problemen zwischen den einzelnen Plattformen kommt. Zur Entstehungszeit und in den ersten Jahren von JavaScript sah die Situation noch ganz anders aus.

Die Geschichte – JavaScript in 10 Tagen

Kein Buch über JavaScript kommt ohne eine kleine Geschichtsstunde aus. Also möchte ich auch hier keine Ausnahme machen. Ich beschränke mich jedoch auf die interessanten, relevanten und vielleicht etwas kuriosen Eckpunkte. Das Gerücht, dass die erste Version von JavaScript in nur 10 Tagen entstanden ist, hält sich hartnäckig. Das würde auch erklären, warum JavaScript an manchen Stellen so ist, wie es ist. Sie werden sowohl im Verlauf dieses Buchs als auch bei Ihrer Arbeit mit der Programmiersprache feststellen, dass JavaScript nicht nur seine Sonnen-, sondern auch die ein oder andere Schattenseite hat. Netscape Communications, das Unternehmen, das den Netscape Navigator, einen der ersten Browser, entwickelt hat, beauftragte 1995 Brendan Eich damit, eine Skriptsprache zu entwickeln, mit der mehr Dynamik in das bis dahin recht statische HTML des Browsers gebracht werden sollte.

Netscape übernahm diese erste Version der Skriptsprache allerdings noch nicht in den Browser. Einen Prototyp einer Programmiersprache in einen der damals am weitesten verbreiteten Browser zu integrieren wäre viel zu riskant gewesen, vor allem vor dem Hintergrund, dass zwischen 1995 und 1998 der sogenannte Browserkrieg zwischen Microsoft und Netscape um die Vorherrschaft über das Internet tobte. Und so entwickelte Brendan Eich bis 1996 weiter an JavaScript, das zunächst unter dem Codenamen Mocha und später unter LiveScript geführt wurde. Version 1.1 wurde mit dem Netscape Navigator 3.0 im August 1996 veröffentlicht. Ebenfalls im August 1996 zog Microsoft mit seinem Konkurrenzprodukt JScript im Internet Explorer 3 nach.

Beide Skriptsprachen entwickelten sich über die Zeit auf eine recht ähnliche Art, wobei mal Netscape, mal Microsoft die Nase vorn hatte. Zum Leidwesen der WebentwicklerInnen der damaligen Zeit unterschieden sich LiveScript und JScript in einigen Aspekten voneinander, dadurch mussten die EntwicklerInnen die jeweilige Umgebung erkennen und speziell angepasste Programmlogik ausführen. Die Geburtsstunde der Polyfills.

Polyfill

Ein Polyfill ist ein Stück Code, das ein Feature in einer Umgebung emuliert, in der es eigentlich nicht existiert. Mit den Grundzügen der Skriptsprachen der Browser konnten EntwicklerInnen schon zu einem frühen Zeitpunkt viele Funktionen bereitstellen und die Schnittstellen so weit vereinheitlichen, dass eine halbwegs komfortable Entwicklung möglich war.

Bei neuen Sprachfeatures ist es auch heute noch üblich, auf Polyfills zurückzugreifen, bis alle für die Applikation relevanten Umgebungen das Feature unterstützen. Eines der populärsten Polyfills, das man in der Regel gar nicht als solches wahrnimmt, ist TypeScript. Eine Programmiersprache, die auf den Regeln von JavaScript aufbaut und die fehlende Typsicherheit zu JavaScript hinzufügt. Der TypeScript-Code wird dann vom TypeScript-Compiler in JavaScript ausgeführt, das dann im Browser oder serverseitig ausgeführt werden kann. Doch der Compiler ist auch in der Lage, moderne Features in älteren Umgebungen zur Verfügung zu stellen.

Falls TypeScript für Sie keine Option ist, können Sie auch auf spezialisierte Polyfills für bestimmte Sprachfeatures zurückgreifen. Sie sollten jedoch daran denken, dass ein Polyfill in der Regel weniger performant als das JavaScript-Original ist. Sobald Sie ein Polyfill also nicht mehr benötigen, sollten Sie es aus dem Quellcode Ihrer Applikation entfernen.

Der erste ECMAScript-Standard erschien im Juni 1997 und brachte den ersten Hoffnungsschimmer. Die Entwicklung des Standards verlief in der ersten Zeit eher schleppend und gipfelte im Abbruch der Arbeiten an der vierten Version im Jahr 2008. Das letzte Update am Sprachstandard gab es zu diesem Zeitpunkt im Jahr 1999. Wir sprechen von einem Zeitraum von 9 Jahren in der auch damals schon schnelllebigen Web-Welt. Die Standardisierung wurde jedoch wieder aufgenommen und viele der Features aus der vierten Version wurden verworfen. 2009 veröffentlichte die ECMA dann endlich die Version 5 von ECMAScript mit der Unterstützung von JSON.

Einen Meilenstein markiert die Veröffentlichung von ECMAScript 6, kurz ES6, oder später dann ECMAScript2015. Neben vielen interessanten Erweiterungen wie beispielsweise Klassen oder Block-Scoping gibt es seit dieser Version jährliche Aktualisierungen des Sprachstandards. Diese werden mit der jeweiligen Jahreszahl benannt. Der ECMAScript-Standard aus dem Jahr 2022 heißt also ECMAScript2022.

Wie kommen neue Features in die Sprache?

Das TC39, eine Abkürzung für Technical Committee 39, eine Arbeitsgruppe der ECMA, hat die Aufgabe, den ECMAScript-Standard weiterzuentwickeln und neue Features zu integrieren. Dem TC39 gehören sowohl JavaScript-EntwicklerInnen als auch Abgesandte großer Unternehmen an, die in verschiedenen Rollen an der Entwicklung von ECMAScript-Umgebungen beteiligt sind. Die Sitzungen des TC39 stehen jedem Mitglied der ECMA offen.

Der Prozess, mit dem ein neues Feature in den Standard aufgenommen wird, besteht aus 5 Stufen, die durch die ECMA genau definiert sind.

✔ **Stufe 0 (Strawman)**

Zweck:

- Bildet den Einstieg zur Erweiterung der Spezifikation.

Eingangskriterien:

- Keine

✔ **Stufe 1(Proposal)**

Zweck:

- Beschreibt den Anwendungsfall für das Feature.
- Form der Lösung ist skizziert.
- Potenzielle Probleme sind umrissen.

Eingangskriterien:

- Es gibt eine zuständige Person im TC39, die das Feature weiterbringt.
- Prosatext über die Ausgestaltung des Features existiert.
- Beispiele der Verwendung ist beschrieben.
- Grobe API-Beschreibung ist vorhanden.
- Diskussion von Schlüsselalgorithmen, Abstraktion und Semantik wurde geführt.
- Identifikation potenzieller »cross cutting concerns« und potenzieller Umsetzungsschwierigkeiten ist vorhanden.
- Öffentliches Repository mit den oben genannten Anforderungen existiert.

✔ **Stufe 2 (Draft)**

Zweck:

- Syntax und Semantik werden exakt beschreiben.

Eingangskriterien:

- Initialer Spezifikationstext ist vorhanden.

✔ **Stufe 3 (Candidate)**

Zweck:

- Weitere Verfeinerung benötigt Feedback von BenutzerInnen und HerstellerInnen.

Eingangskriterien:

- Kompletter Spezifikationstext ist vorhanden.
- Ausgewählte ReviewerInnen haben den aktuellen Spezifikationstext genehmigt.
- Alle ECMAScript EditorInnen haben den aktuellen Spezifikationstext genehmigt.

✔ **Stufe 4 (Finished)**

Zweck:

- Feature ist bereit für die Integration in den ECMAScript-Standard.

Eingangskriterien:

- Es wurden Test262 Akzeptanztests für die wichtigsten Nutzungsszenarien geschrieben und integriert.
- Es gibt zwei kompatible Implementierungen die die Tests durchlaufen.
- Es gibt ausreichend Erfahrung mit der Implementierung in der realen Welt.
- Es gibt einen Pull Request in tc39/ecma262 mit dem integrierten Spezifikationstext.
- Alle ECMAScript EditorInnen haben den Pull Request genehmigt.

Die aktuell laufenden Anträge für neue Features finden Sie auf GitHub unter `https://github.com/tc39/proposals/`. Mit diesem Wissen können Sie sich nun in die Welt von JavaScript stürzen und Schritt für Schritt Ihre eigenen Applikationen entwickeln.

Webseite vs. Website vs. Web-Applikation

Bei der Arbeit mit JavaScript werden Sie immer wieder über verschiedene Begriffe stolpern. Unter den ersten werden sich wahrscheinlich Webseite, Website und Web-Applikation befinden. Es gibt zwar kein offizielles Glossar für solche Begriffe, jedoch eine weitverbreitete Interpretation.

Viele EntwicklerInnen benutzen den Begriff Webseite eher abwertend und bezeichnen damit ein einzelnes HTML-Dokument mit etwas CSS und JavaScript, also nichts Weltbewegendes. Eine Website ist eine Sammlung von HTML-Dokumenten, also Webseiten, zu einem bestimmten Thema unter meist unter einer Domain. Der Umfang ist hier schon deutlich größer. Bei einer Website ist auch meist das Backend schon mit inbegriffen, der Anspruch ist also deutlich größer.

Eine Web-Applikation ist schließlich die höchste Ebene. Web-Applikationen werden meist als Single-Page-Applikationen mit großen Frameworks und Schnittstellen im Backend umgesetzt. Mit Web-Applikationen können Sie sowohl große Plattformen im Internet als auch kritische Unternehmensprozesse in Web-Technologien umsetzen.

Je nachdem, mit wem Sie sprechen, werden die Grenzen zwischen den einzelnen Begriffen verschwimmen. Nehmen Sie diese Erklärungen einfach als einen groben Anhaltspunkt. Ich verwende in diesem Buch in den meisten Fällen den Begriff Applikation und meine damit so ziemlich alles, was Sie mit JavaScript umsetzen können, von einfachen animierten Webseiten bis hin zu umfangreichen Applikationen.

Die Entwicklungsumgebung (IDE)

Die Spanne der verfügbaren Editoren und Entwicklungsumgebungen reicht von einfachen Editoren auf der Kommandozeile wie Vi und Emacs über umfangreichere grafische Editoren wie Sublime bis hin zu vollwertigen Entwicklungsumgebungen wie WebStorm und Visual Studio Code. Die Unterscheidung der Begriffe ist hier auch eher verschwommen. Ein Editor stellt Ihnen in der Regel die Möglichkeit zur Verfügung, Quelltext zu schreiben, und bietet nur einige wenige Komfortfeatures wie beispielsweise Syntax Highlighting und Autocompletion. In einer Entwicklungsumgebung sind alle Werkzeuge, die Sie für die Entwicklung Ihrer Applikation benötigen, enthalten. Das bedeutet, dass Sie nicht aus Ihrer Entwicklungsumgebung herauswechseln müssen, um eine bestimmte Aufgabe auszuführen. Ein typisches Beispiel ist das Ausführen von Tests. Die Werkzeuge sind meist so gut in die Entwicklungsumgebung integriert, dass Sie die Ergebnisse auch direkt im Quelltext sehen können. Im Fall der Tests sehen Sie beispielsweise das Testergebnis direkt bei der Funktion, die Sie überprüfen.

Aktuell sind WebStorm von JetBrains und Visual Studio Code von Microsoft die beiden in der JavaScript-Welt vorherrschenden Entwicklungsumgebungen. WebStorm ist zwar kostenpflichtig, punktet jedoch mit einem großen Funktionsumfang und hervorragender Stabilität. Die Entwicklungsumgebung verfügt über eine Plug-in-Architektur, mit der Sie die Entwicklungsumgebung durch externe Module erweitern und so den Funktionsumfang auf Ihre Bedürfnisse zuschneiden können. Die offizielle Webseite von WebStorm finden Sie unter `https://www.jetbrains.com/de-de/webstorm/`.

Der schärfste Konkurrent von WebStorm ist Visual Studio Code oder kurz VSCode, die kostenfreie Entwicklungsumgebung von Microsoft. VSCode wird als Open-Source-Projekt auf GitHub unter `https://github.com/Microsoft/vscode/` entwickelt. Die Webseite des Projekts ist unter `https://code.visualstudio.com/` erreichbar. Ähnlich wie WebStorm können Sie auch VSCode durch Erweiterungen um zusätzliche Features ergänzen. Erwähnenswert zu VSCode ist vielleicht noch, dass die Entwicklungsumgebung selbst in JavaScript beziehungsweise TypeScript implementiert ist.

Egal für welches Werkzeug Sie sich beim Schreiben Ihres Quellcodes entscheiden, Sie sollten darauf achten, dass Ihnen Features wie Syntax Highlighting, also die Hervorhebung bestimmter Elemente wie die Namen von Bezeichnern, oder Autocompletion, das sind Vorschläge wie Funktions- oder Methodennamen, zur Verfügung stehen.

Mit einer funktionierenden Entwicklungsumgebung können Sie sich daranmachen und Ihre Arbeitsumgebung kennenlernen, und das ist für JavaScript-EntwicklerInnen in erster Linie die JavaScript-Engine. Für kleinere Experimente mit JavaScript können Sie alternativ zu einer vollumfänglichen Entwicklungsumgebung auch auf Online-IDEs wie Codepen (`https://codepen.io/`), CodeSandbox (`https://codesandbox.io/`) oder StackBlitz (`https://stackblitz.com/`) ausweichen. Und wenn es mal wirklich schnell gehen soll, können Sie sogar in der Browser-Konsole Ihren JavaScript-Code ausführen.

Die JavaScript-Engine

Egal in welcher Umgebung Sie JavaScript entwickeln wollen, Sie haben es immer mit einer JavaScript-Engine zu tun. Dabei handelt es sich um eine Software, der Sie JavaScript-Quellcode übergeben und die diesen dann ausführt. Zwar ist JavaScript, wie der Name vermuten lässt, eine Skriptsprache. Die Engine arbeitet jedoch keineswegs direkt auf dem Text, den Sie ihr übergeben. Stattdessen wandelt sie den Text in Bytecode, also Maschinencode, um und verwendet diesen. Diese Umwandlung hat zur Konsequenz, dass Sie den Code bei Änderungen erneut der Engine übergeben müssen und diese ihn erneut einlesen muss. Dieser Charakter von JavaScript macht die Sprache deutlich leichtgewichtiger als kompilierte Sprachen wie beispielsweise C, C++ oder Java wo Sie den Quellcode mit einem separaten Compiler zunächst übersetzen müssen, bevor Sie ihn ausführen können.

Aktuell gibt es auf dem Markt eine eher überschaubare Anzahl von JavaScript-Engines. Die wichtigsten sind die V8-Engine, die Sie in Chrome, Edge und Node.js finden, SpiderMonkey, die Engine hinter Firefox, und JavaScriptCore aus Safari. Nachdem die Interpretation von JavaScript durch den ECMAScript-Standard relativ strikt vorgeschrieben ist, unterscheiden sich die Engines hauptsächlich in der Adaption neuer Features, wobei hier V8 und

SpiderMonkey in der Regel die Nase vorn haben, und hinsichtlich der Performance einzelner Features. Denn die interne Umsetzung des Standards bleibt den Herstellern der Engines überlassen, sodass diese hier einen gewissen Spielraum haben. Als EntwicklerIn von client- oder serverseitigem JavaScript haben Sie normalerweise nicht viel mit der Engine selbst zu tun, sodass Sie sich nicht weiter um die internen Strukturen und Abläufe, wie beispielsweise das Speichermanagement, kümmern müssen.

Es ist jedoch hilfreich zu wissen, dass die JavaScript-Engines einen Garbage Collector haben, der den nicht mehr genutzten Speicher in gewissen Abständen wieder freiräumt. Einige Engines weisen Optimierungen auf, die Applikationen bei der Wiederverwendung von Objektstrukturen beschleunigen. Dabei erzeugt die Engine beim Zugriff auf die Eigenschaften des Objekts eine Art Katalog für den Speicher, mit dessen Hilfe sie die Eigenschaften bei wiederholtem Zugriff schneller lokalisieren kann. Diese und weitere Best Practices lernen Sie im Laufe dieses Buchs noch näher kennen und erfahren, wann es sinnvoll ist, Ihren Quellcode auf eine bestimmte Weise zu strukturieren. Dabei müssen Sie auch immer abwägen, wenn Sie Ihren Quellcode auf Performance oder auf Lesbarkeit optimieren.

HTML, CSS und JavaScript im Client

JavaScript ist eine Programmiersprache, die zwar ihren Ursprung im Browser hat, die Sie jedoch unabhängig von der Umgebung verwenden können. Und so besteht ein Browser nicht nur aus einer JavaScript-Engine, sondern noch aus zahlreichen weiteren Bestandteilen wie beispielsweise einer Rendering-Engine, die dafür sorgt, dass Ihre BenutzerInnen die Strukturen, die Sie implementieren, auch zu sehen bekommen. Entschließen Sie sich dazu, clientseitige Applikationen zu implementieren, müssen Sie sich zunächst damit beschäftigen, wie die Umgebung aufgebaut ist und wie sie funktioniert. Beim Browser bedeutet das, Sie sollten wissen, wie die Ressourcen vom Server zum Client kommen, wie dieser sie verarbeitet und schließlich das Ergebnis darstellt.

Damit Sie die Beispiele in den folgenden Kapiteln nachvollziehen können, stelle ich Ihnen in diesem Kapitel eine einfache Beispiel-Applikation vor, die Sie als Umgebung für Ihre Experimente verwenden können. Im Client arbeiten Sie generell mit drei verschiedenen Sprachen: HTML, mit dem Sie die Struktur einer Seite definieren, CSS, das für das Styling der Elemente verantwortlich ist, und JavaScript, mit dem Sie die Logik für Ihr Frontend umsetzen. Die Kombination aus diesen drei Aspekten setzen Sie im einfachsten Fall als statische Webseite um und legen diese entweder im Dateisystem Ihres Systems oder auf einem Webserver ab. Über den Browser können Sie dann entweder über »Datei« > »öffnen« die HTML-Datei öffnen oder Sie geben, im Fall eines Webservers, dessen Adresse in die URL-Zeile ein.

Beispiel-Set-up

Bei clientseitigem JavaScript führen Sie den Code im Browser aus. Die Dateien können Sie entweder direkt von Ihrem System laden oder über einen Webserver ausliefern. Für das clientseitige Beispiel nutzen wir den zweiten Ansatz, also die serverbasierte Auslieferung. Diese Aufgabe erfüllt für uns Node.js als Server. Diese Plattform wird Ihnen auch beim serverseitigen JavaScript wieder begegnen. Die Software erhalten Sie über die

offizielle Webseite des Projekts, die Sie unter `https://nodejs.org\ignorespaceserreichen`. Dort finden Sie zwei Varianten, die LTS-Version, die sehr stabil ist und für die meisten BenutzerInnen empfohlen wird, und die aktuelle Version, in der Sie auf die neuesten, teilweise noch in der Entwicklung befindlichen Features zugreifen können. Für Windows und macOS können Sie Installationspakete herunterladen und auf Ihrem System installieren. Bei Linux-Systemen führt der Weg üblicherweise über den Paketmanager des Systems, wofür Sie ebenfalls eine Schritt-für-Schritt-Anleitung auf der Webseite finden. Neben der Node.js-Plattform installieren Sie dabei auch das Werkzeug NPM, den Node Package Manager, und NPX, ein Werkzeug, mit dem Sie JavaScript-Pakete ausführen können. Diese Kombination verwenden Sie, um einen einfachen Webserver mit Node.js auszuführen und die Dateien Ihres Frontends an Ihren Browser auszuliefern.

Dazu legen Sie in einem Verzeichnis Ihrer Wahl eine Datei mit dem Namen index.html an. Hier erzeugen Sie eine grundlegende HTML-Struktur für Ihre Seite, wie Sie sie im folgenden Codebeispiel sehen können.

Dieses und alle anderen Listings in diesem Buch finden Sie zum Download unter `https://www.wiley-vch.de/ISBN9783527720644`

```
<!DOCTYPE html>
<html lang="de">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1.0" />
    <title>JavaScript für Dummies</title>
    <link rel="stylesheet" href="style.css" />
    <script src="index.js"></script>
  </head>
  <body>
    <div>Hallo Welt</div>
  </body>
</html>
```

Listing 1.1: Grundstruktur einer HTML-Datei

HTML orientiert sich an XML. Die einzelnen Elemente schreiben Sie als sogenannte Tags, das sind festgelegte Bezeichnungen, die Sie in spitze Klammern fassen. Jedes Element hat üblicherweise ein öffnendes und ein schließendes Tag. Die Elemente bilden eine Baumstruktur, dessen Wurzel das `html`-Element bildet. In diesem Wurzelelement liegen das `head`- und das `body`-Element. Im `head` legen Sie hauptsächlich Metainformationen, wie beispielsweise die Zeichencodierung oder den Titel der Seite, ab. Außerdem binden Sie hier das Stylesheet, also die CSS-Ressource, und das JavaScript mithilfe eines `script`-Tags ein. Im `body` definieren Sie die sichtbare Struktur. Im Beispiel fügen Sie ein `div`-Element mit dem Textinhalt `"Hallo Welt"` ein. Ein `div`-Element ist ein einfaches Container-Element, mit dessen Hilfe Sie beispielsweise Text anzeigen können. Im folgenden Codeblock sehen Sie den Quellcode der `style.css`-Datei. Hierbei handelt es sich um einfaches CSS, das dafür sorgt, dass um das `div`-Element ein durchgehender schwarzer Rahmen gezogen wird.

```
div {
  border: 1px solid black;
}
```

Listing 1.2: Style-Definition

Zu guter Letzt setzen Sie noch das JavaScript um. Hier können Sie die `alert`-Funktion ausführen, um zu testen, ob Ihr Set-up funktioniert. Diese Funktion zeigt ein kleines Dialogfenster im Browser mit Textinhalt und einem OK-Button an. Betätigen Sie den Button, verschwindet der Dialog wieder. Den zugehörigen Quellcode sehen Sie hier:

```
alert('Hallo Welt');
```

Listing 1.3: JavaScript-Code, der einen Dialog öffnet

Semikolon

In JavaScript terminieren Sie Anweisungen wie beispielsweise `const result = 1 + 2` oder das `alert('Hallo Welt')` im Beispiel mit einem Semikolon (;). Dieses Semikolon ist optional, die JavaScript-Engine fügt es automatisch ein. Es gibt jedoch Fälle, in denen dieser Automatismus zu ungewollten Effekten führt. So zum Beispiel, wenn Sie nach dem `return` in einer Funktion den Wert, den Sie zurückgeben möchten, in einer neuen Zeile schreiben. In diesem Fall fügt JavaScript das Semikolon direkt nach dem `return` ein und gibt so den Wert `undefined` zurück. Um auf Nummer sicher zu gehen, sollten Sie immer Semikolons verwenden, um Ihre Anweisungen zu beenden.

Öffnen Sie nun ein Terminalfenster auf Ihrem System und wechseln Sie in das Verzeichnis, in dem Sie die drei Dateien erzeugt haben. Geben Sie dort den folgenden Befehl ein: `npx serve`. Als Ergebnis sollten Sie eine Ausgabe wie in Abbildung 1.1 erhalten.

Abbildung 1.1: Screenshot der Ausführung von npx serve

Dieses Kommando lädt das serve-Paket von npmjs.com herunter und führt es mit Node.js aus. Die Plattform erzeugt einen Webserver und liefert den Inhalt des aktuellen Verzeichnisses aus. Öffnen Sie den Browser Ihrer Wahl und geben in die Adressleiste `http://localhost:3000` ein, erhalten Sie eine Ansicht wie in Abbildung 1.2.

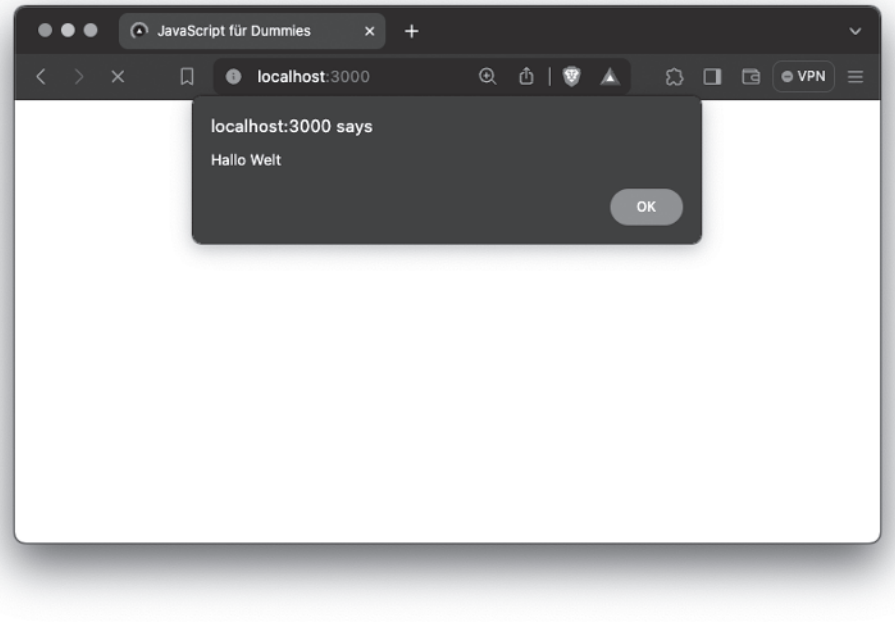

Abbildung 1.2: Anzeige der Applikation im Browser

Klicken Sie auf den Button, sehen Sie den Text »Hallo Welt« in einem schwarzen Kasten. Damit haben Sie den ersten Schritt in die Entwicklung von clientseitigem JavaScript gemacht.

Einbindung von JavaScript

Der Aufbau des vorangegangenen Beispiels sieht vor, dass Sie die Struktur vom Styling und der Logik strikt trennen und alle drei Aspekte in eigenen Dateien liegen. Dieser Lösungsansatz ist zwar der sauberste, jedoch nicht der einzige. Sie können JavaScript und HTML auch näher zusammenbringen, was ich Ihnen jedoch nicht empfehle, da der Quellcode in diesem Fall sehr schnell unübersichtlich wird und Sie Ihr JavaScript auch nicht an anderen Stellen in Ihrer Applikation wiederverwenden können. Die beiden anderen Varianten der Einbindung von JavaScript sind JavaScript innerhalb eines Script-Tags und JavaScript als Eventhandler. Beide Möglichkeiten stelle ich Ihnen der Vollständigkeit halber hier kurz vor. Der folgende Code gleicht vom Funktionsumfang dem aus dem vorangegangenen Beispiel, mit dem Unterschied, dass der JavaScript-Code jetzt direkt im HTML liegt.

```
<!DOCTYPE html>
<html lang="de">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1.0" />
    <title>JavaScript für Dummies</title>
    <link rel="stylesheet" href="style.css" />
    <script>
      alert('Hallo Welt');
    </script>
  </head>
  <body>
    <div>Hallo Welt</div>
  </body>
</html>
```

Listing 1.4: HTML mit Inline-JavaScript

Für die letzte Art der Einbindung müssen Sie etwas tiefer in die Trickkiste greifen. In Listing 1.4 sehen Sie, wie Sie JavaScript innerhalb eines HTML-Tags schreiben und damit auf die Interaktion von BenutzerInnen reagieren können.

```
<!DOCTYPE html>
<html lang="de">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1.0" />
    <title>JavaScript für Dummies</title>
    <link rel="stylesheet" href="style.css" />
  </head>
  <body>
    <div>Hallo Welt</div>
    <button onclick="alert('Hallo Welt');">Klick mich</button>
  </body>
</html>
```

Listing 1.5: Im HTML eingebettetes JavaScript

Benutzen Sie JavaScript bitte nicht auf diese Weise. Die Verflechtung von Struktur und Logik ist in diesem Fall so eng, dass der Code weder lesbar noch erweiterbar ist. Fehler sind in diesem Fall vorprogrammiert. Es gibt deutlich elegantere Arten, mit denen Sie auf Klicks und viele andere Ereignisse im Browser reagieren können. Wie das genau funktioniert, erfahren Sie in Kapitel 11.

Die Browser-Developer-Tools

Ihr Browser kann Ihnen nicht nur Webseiten anzeigen und JavaScript ausführen, sondern kann Sie auch bei der Entwicklung unterstützen. Unter der Haube ist ein ganzer Satz mächtiger Entwicklungswerkzeuge versteckt. Alles, was Sie tun müssen, um darauf zuzugreifen, ist, die F12-Taste zu betätigen. Je nachdem, welchen Browser Sie verwenden, sehen diese Werkzeuge unterschiedlich aus. Die Grundfunktionen finden Sie jedoch in jedem Browser wieder. In Abbildung 1.3 sehen Sie einen Screenshot der Chrome Developer Tools für die Beispielapplikation.

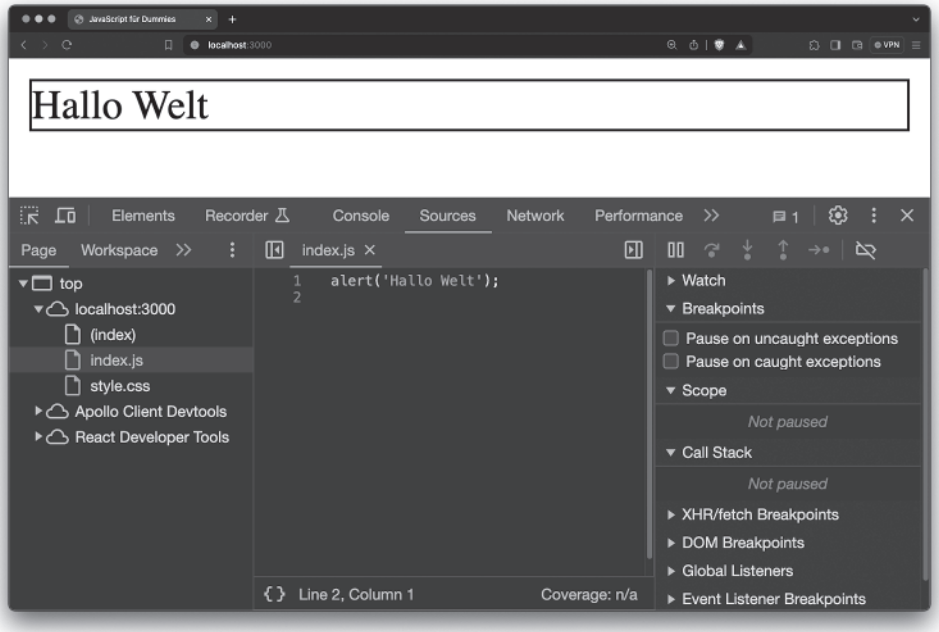

Abbildung 1.3: DevTools im Browser

Die wichtigsten Tabs sind:

- ✔ **Elements:** Über den Elements-Tab können Sie mit der HTML-Struktur und dem Styling der Seite arbeiten. Bewegen Sie die Maus über ein Element im HTML-Baum, markiert der Browser das zugehörige Element im Hauptfenster. Mit einem Rechtsklick im Hauptfenster auf ein Element und dann einem Linksklick auf »Inspect« gelangen Sie direkt zum jeweiligen Element im Elements-Tab.

  Auf der rechten Seite der Developer-Tools sehen Sie die Styles des jeweiligen Elements und woher diese stammen.

  Sowohl das HTML als auch das CSS können Sie über die Developer-Tools manipulieren. Der Browser speichert diese Änderungen jedoch nicht. Laden Sie die Seite neu, sind Ihre Änderungen wieder verschwunden.

- ✔ **Console:** Im Console-Tab können Sie JavaScript direkt ausführen. Außerdem können Sie in Ihrem JavaScript-Quellcode über das `console`-Objekt Ausgaben auf der Konsole erzeugen. Mehr zum Thema JavaScript-Konsole erfahren Sie im nachfolgenden Abschnitt.
- ✔ **Sources:** Der Sources-Tab erlaubt Ihnen den Zugriff auf die Dateien Ihrer Applikation. Hier können Sie den Quellcode einsehen und, noch viel wichtiger, Sie haben die Möglichkeit, mit dem Debugger interaktiv mit Ihrem Quellcode zu arbeiten. Wie das genau funktioniert, sehen Sie im Abschnitt »Debugging«.
- ✔ **Network:** Die Dateien Ihrer Applikation werden vom Server zum Browser gesendet. Wie diese Kommunikation genau aussieht und wie lange die einzelnen Phasen des Downloads gedauert haben, sehen Sie im Network-Tab. Dieser ist vor allem interessant, wenn es darum geht, die Ladeperformance Ihrer Applikation zu überprüfen und nach Optimierungspotenzial zu suchen.

Neben diesen vier Tabs bieten Ihnen die Developer-Tools Ihres Browsers noch viele weitere Hilfsmittel für die Arbeit mit Ihrer Applikation. So können Sie beispielsweise den Verlauf der CPU-Auslastung analysieren oder die Speicherbelegung zu einem bestimmten Zeitpunkt ansehen. In der täglichen Arbeit mit JavaScript greifen Sie auf diese Werkzeuge jedoch eher selten zur. Deutlich häufiger haben Sie mit der Konsole zu tun.

Die JavaScript-Konsole

In der Konsole zeigt Ihnen der Browser Fehlermeldungen und Warnungen an, die bei der Verarbeitung Ihres Quellcodes aufgetreten sind. Außerdem können Sie selbst Ausgaben erzeugen. Wie das funktioniert, sehen Sie in Listing 1.6.

```
console.log('Hallo Welt');
```

Listing 1.6: Erzeugen einer Konsolenausgabe

Damit die Ausgabe funktioniert, binden Sie die `index.js`-Datei, in der Sie den Quellcode gespeichert haben, in eine HTML-Datei ein. Wechseln Sie in den Browser, stellen Sie sicher, dass die Developer Tools mit dem Console-Tab geöffnet sind, und laden Sie die Seite neu. Daraufhin erhalten Sie die Ausgabe `"Hallo Welt"` auf der Konsole.

Dort gibt es nicht nur eine Ausgabe, sondern auch eine Eingabeaufforderung. Hier können Sie beliebiges JavaScript eingeben, das der Browser, nachdem Sie die Enter-Taste betätigt haben, im aktuellen Kontext Ihrer Applikation ausführt. In der folgenden Abbildung 1.4 sehen Sie die Ausgabe, wenn Sie `2 + 2` auf der Konsole eingegeben haben.

Kommentare

Kommentare sind das Salz in der Suppe von JavaScript. Sie können schwer zu lesende Codestellen mit dem nötigen Kontext versehen und größere Strukturen wie beispielsweise Funktionen dokumentieren. Üblicherweise beschreiben Sie hier, wie die Struktur zu verwenden ist und welche Besonderheiten es gibt. Sie werden aber auch Quellcode finden, der wenig bis

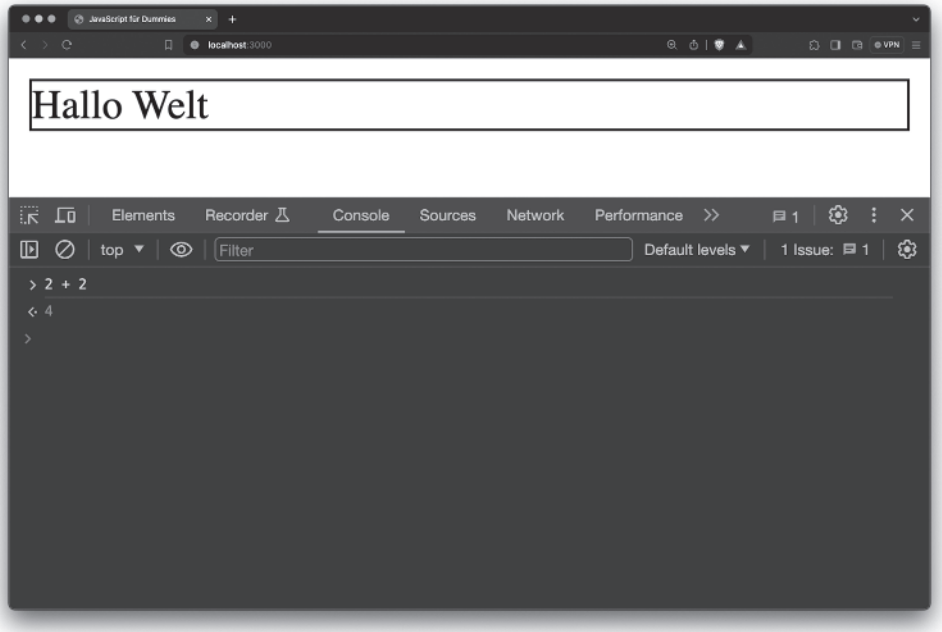

Abbildung 1.4: JavaScript-Ausführung in den DevTools

gar nicht kommentiert ist. Im besten Fall ist der Code selbsterklärend und im schlimmsten Fall müssen Sie sich um die Kommentare kümmern.

Sie können Kommentare auch verwenden, um Code temporär zu deaktivieren, ohne ihn gleich komplett löschen zu müssen.

JavaScript kennt zwei verschiedene Arten von Kommentaren: einzeilige und mehrzeilige.

Einzeilige Kommentare

Einen einzeiligen Kommentar leiten Sie mit zwei `//` ein. Alles, was danach bis zum Zeilenende folgt, ist für die JavaScript-Engine ein Kommentar und wird nicht ausgeführt.

```
const result = 1 + 2; // Das Ergebnis ist 3

// Die folgende Zeile erzeugt eine Konsolenausgabe
console.log('Das Ergebnis ist: ' + result);
```

Listing 1.7: Einzeiliger Kommentar in JavaScript

Mehrzeilige Kommentare

Der Begriff mehrzeiliger Kommentar ist in JavaScript etwas irreführend, denn Sie können mit der Syntax `/* ... */` sowohl einen mehrzeiligen als auch einen einzeiligen Kommentar erzeugen. Nutzen Sie diese Syntaxvariante, ist es JavaScript egal, wie viele Zeilen dieser Kommentar umfasst. Er beginnt mit der Zeichenkette `/*` und endet mit `*/`. Zwischen den beiden Sequenzen können eine Handvoll Zeichen, aber auch Hunderte Zeilen liegen.

```
/*
  Die folgende Funktion addiert zwei Zahlen
*/
function add(a, b) {
  return a + b; /* Diese Zeile gibt den berechneten Wert zurück */
}
```

Listing 1.8: Mehrzeiliger Kommentar in JavaScript

Debugging

Noch haben Sie nicht viel von JavaScript gesehen, aber wir beschäftigen uns schon mit der Suche und dem Beheben von Fehlern. Denn das Debugging und die damit einhergehende Beherrschung der entsprechenden Werkzeuge ist eine elementare Fertigkeit, die Sie in Ihrer Tätigkeit immer wieder benötigen. Es ist also niemals zu früh, sich mit diesen Hilfsmitteln zu beschäftigen.

Ihr Browser verfügt über einen vollwertigen Debugger, mit dem Sie sich schrittweise durch Ihren Quellcode bewegen können, und zwar zur Laufzeit Ihrer Applikation. Mit einem einzelnen `console.log`-Statement ergibt Debugging natürlich wenig Sinn, also greife ich den folgenden Kapiteln etwas vor und zeigen Ihnen in Listing 1.9 einen Codeblock, den Sie etwas sinnvoller debuggen können.

```
function add(a, b) {
  const result = a + b;
  return result;
}

const number1 = 3.14;
const number2 = 42;

const sum = add(number1, number2);
console.log(sum);
```

Listing 1.9: Codebeispiel als Grundlage für Debugging

Im Quellcode definieren Sie zunächst eine Funktion `add`, die zwei Werte akzeptiert. Diese beiden Werte addieren Sie und speichern sie in einer Konstante mit dem Namen `result` und geben diese anschließend mithilfe des `return`-Statements zurück. Im Anschluss daran definieren Sie zwei weitere Konstanten mit den Werten `3,14` und `42`. Diese beiden Werten übergeben Sie anschließend beim Aufruf der `add`-Funktion und speichern das Ergebnis in

der Konstanten `sum` zwischen. Im letzten Schritt geben Sie das Ergebnis auf der Konsole des Browsers aus. Fügen Sie diesen Quellcode in Ihre `index.js`-Datei und speichern Sie diese ab, können Sie mit dem Debuggen beginnen.

Zum Debuggen Ihres Codes müssen Sie einen Breakpoint setzen. Ein Breakpoint ist ein Punkt im Quellcode, an dem der Browser die Ausführung der Programmlogik anhält und Ihnen die Möglichkeit bietet, sich in Ihrer Applikation zum Zeitpunkt der Ausführung umzusehen und sogar direkt mit der Programmlogik zu interagieren. Diese Breakpoints können Sie entweder direkt im Quellcode oder mithilfe der Entwicklerwerkzeuge Ihres Browsers setzen.

Um einen Breakpoint im Code zu setzen, fügen Sie an der gewünschten Stelle ein `debugger`-Statement ein. Wie das funktioniert, sehen Sie im folgenden Beispiel.

```
function add(a, b) {
  const result = a + b;
  return result;
}

const number1 = 3.14;
const number2 = 42;

const sum = add(number1, number2);
debugger;
console.log(sum);
```

Listing 1.10: Einsatz des `debugger`-Statements

Das `debugger`-Statement sorgt dafür, dass der Debugger Ihres Browsers nach dem Aufruf der `add`-Funktion anhält. Damit dies funktioniert, müssen die Entwicklerwerkzeuge des Browsers geöffnet sein und Sie müssen die Applikation einmal neu laden. Sind die Entwicklerwerkzeuge nicht geöffnet, ignoriert der Browser das `debugger`-Statement und führt Ihre Applikation wie gewohnt ohne Halt von Anfang bis Ende aus.

Der Vorteil dieser Variante ist, dass Sie die Breakpoints direkt dort definieren, wo Sie arbeiten. Sie wissen also genau, wo die Ausführung angehalten werden soll, und müssen nicht erst mühevoll nach der passenden Datei suchen. Setzen Sie einen Breakpoint manuell im Quellcode, hat das den Nachteil, dass Sie den Quellcode modifizieren müssen. Das bedeutet, dass Sie dabei Fehler einfügen könnten, falls Sie sich vertippen, und dass Sie Ihre Applikation im ungünstigsten Fall neu bauen müssen, falls Sie Werkzeuge wie TypeScript oder Webpack verwenden.

Deutlich eleganter, weil ohne Modifikation am Quellcode, funktioniert das Setzen von Breakpoints direkt aus den Entwicklerwerkzeugen des Browsers. Hierfür öffnen Sie zunächst wie gewohnt Ihre Applikation im Browser, öffnen anschließend die Entwicklerwerkzeuge und wechseln dann in den Sources-Tab. Auf der linken Seite sehen Sie einen Dateibaum. Dort können Sie die Quelltext-Datei auswählen, in der Sie Ihren Breakpoint setzen möchten. Alternativ können Sie auch nach der Datei suchen, indem Sie entweder über das Kontextmenü des Dateibaums und »Open file« oder den Tastatur-Shortcut CTRL-P, beziehungsweise auf einem Mac CMD-P, eine Suchmaske öffnen, in der Sie den

gewünschten Dateinamen eingeben können. Sobald Sie den Quellcode der Datei sehen, können Sie links auf die Zeilennummer klicken, um einen Breakpoint zu setzen. Dieser wird Ihnen durch eine Markierung an der Zeilennummer angezeigt.

Klicken Sie mit der rechten Maustaste auf eine Zeilennummer, können Sie über den Menüpunkt »add conditional breakpoint...« einen sogenannten bedingten Breakpoint setzen. Dieser wird nur aktiv, wenn die Bedingung, die Sie für diesen Breakpoint angeben, wahr ist. Fügen Sie also beispielsweise in Zeile 3 einen solchen bedingten Breakpoint mit der Bedingung »result === 45.14« ein, hält der Browser hier nur an, wenn der Wert der Konstanten `result` 45.14 ist.

Der Debugging-Prozess

Haben Sie Ihre Applikation im Browser geöffnet und die Entwicklerwerkzeuge sind aktiv, hält der Browser am Breakpoint an. Sie sollten dann eine Ansicht wie in Abbildung 1.5 sehen.

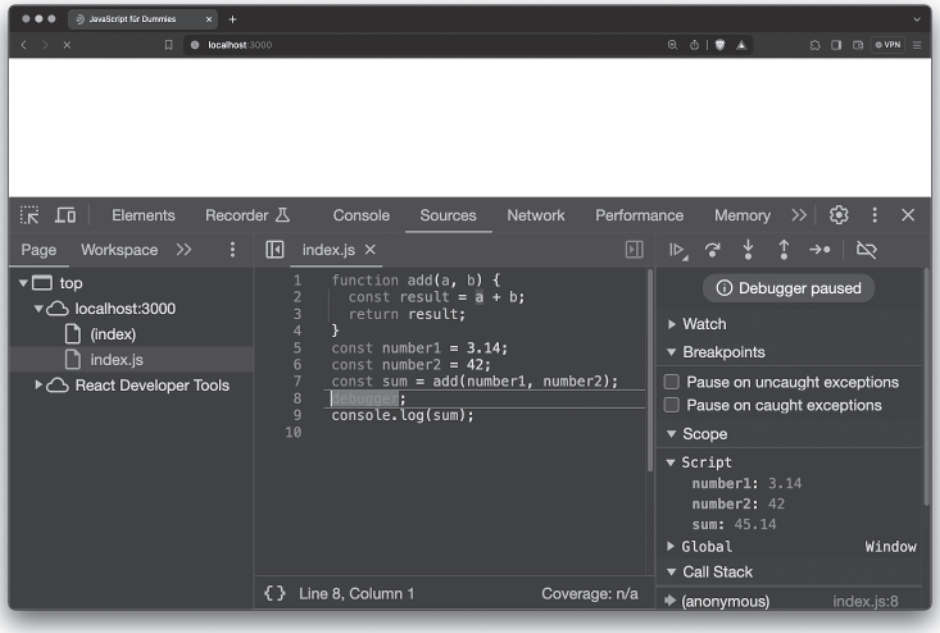

Abbildung 1.5: JavaScript-Debugging im Browser

Dass sich der Browser im Debug-Modus befindet und aktuell angehalten ist, sehen Sie, da die Zeile mit dem aktiven Breakpoint hervorgehoben ist und in der rechten Spalte der Entwicklerwerkzeuge die Information »Debugger paused« steht. Im Debugger können Sie nun über die Schaltelemente in der rechten Spalte der Entwicklerwerkzeuge navigieren.

Neben diesen Steuerelementen können Sie Watch Expressions definieren. Das sind Ausdrücke, die bei jedem Schritt des Debuggers erneut ausgewertet werden und Ihnen den jeweiligen Wert anzeigen. Außerdem sehen Sie eine Liste Ihrer Breakpoints, in der Sie

Steuerelement	Bedeutung
Resume	Die Ausführung wird bis zum nächsten Breakpoint fortgesetzt
Step over	Der Debugger springt über den nächsten Funktionsaufruf, sodass direkt dessen Ergebnis vorliegt
Step into	Der Debugger springt in den nächsten Funktionsaufruf, sodass Sie sich die Funktion genauer ansehen können
Step out	Springt aus dem aktuellen Funktionsaufruf, sodass Sie mit dem Rückgabewert weitermachen können
Step	Springt zur nächsten Anweisung

Tabelle 1.1: Steuerelemente des Debuggers

diese verwalten können. Ein weiteres wichtiges Element ist die Scope-Liste, in der Sie die Belegung aller aktuell verfügbaren Variablen und Konstanten einsehen können.

Wechseln Sie mit aktivem Debugger auf die JavaScript-Konsole, können Sie mit Ihrer Applikation interagieren und beispielsweise Funktionen aufrufen oder sich die Belegung von Variablen ausgeben lassen. Sie haben an dieser Stelle Zugriff auf den gesamten JavaScript-Funktionsumfang.

Die Beispiele für das Debuggen haben sich bisher auf Chrome bezogen. Sie sind jedoch beim Debuggen Ihres JavaScript-Quellcodes nicht auf diesen Browser beschränkt. Auch Firefox, Safari, Edge und alle weiteren Browser bieten Ihnen ähnliche Funktionalität, die eventuell etwas anders benannt ist und sich an unterschiedlichen Stellen wiederfinden. Für die Details der jeweiligen Plattform sollten Sie einen Blick in die Dokumentation Ihres Browsers werfen.

Debugging aus der Entwicklungsumgebung heraus

Das Debugging ist jedoch nicht nur auf Browser beschränkt. Moderne Entwicklungsumgebungen bieten Ihnen die Möglichkeit, sich mit dem Browser zu verbinden und dann direkt aus der Entwicklungsumgebung heraus zu debuggen. Bei der Konfiguration unterscheiden sich die jeweiligen Entwicklungsumgebungen stark voneinander, sodass ich Sie an dieser Stelle auf die Dokumentation Ihrer Entwicklungsumgebung verweisen und Ihnen exemplarisch das Vorgehen in Visual Studio Code zeigen möchte.

Visual Studio Code verfügt über eine integrierte Debugger-Anbindung für Chrome und Edge. Als Voraussetzung für die folgende Debugging-Session muss Ihre Applikation über eine URL erreichbar sein. Wie Sie bereits im ersten Beispiel gesehen haben, bewerkstelligen Sie dies beispielsweise über das Kommando `npx serve`, das einen lokalen Webserver startet. Anschließend öffnen Sie über »View« > »Command Palette« das Suchfeld für Kommandos und geben »Debug: Open Link« ein. Wählen Sie den Vorschlag der Entwicklungsumgebung

aus und geben Sie dann die Adresse `http://localhost:3000` ein. Anschließend startet der Debugger von Visual Studio Code und stellt Ihnen die gleichen Features wie schon der Browser zur Verfügung. Der entscheidende Vorteil ist hier, dass Sie sich direkt im Quellcode Ihrer Applikation befinden und sich in Ihrer gewohnten Arbeitsumgebung bewegen. Beachten Sie allerdings, dass Änderungen am Quellcode nicht direkt in der aktuellen Debugging-Session wirksam werden, da die JavaScript-Engine den Quelltext eingelesen und umgewandelt hat.

In Abbildung 1.6 sehen Sie, wie der Debugger in VSCode aussieht.

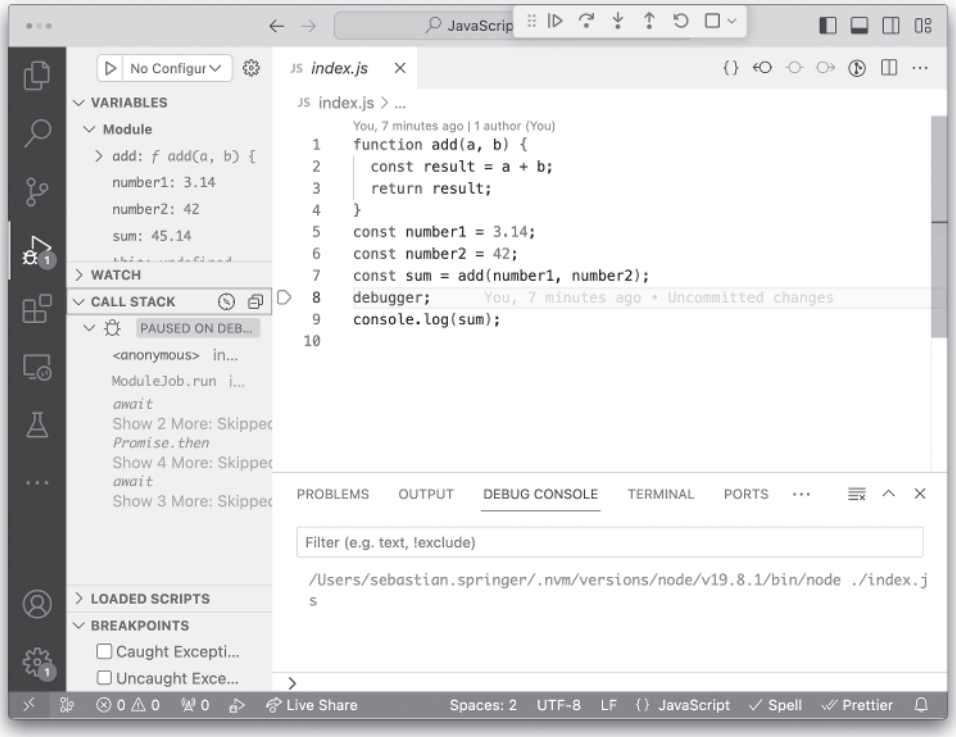

Abbildung 1.6: JavaScript-Debugging in VSCode

Serverseitiges JavaScript

JavaScript ist zwar im Browser groß geworden, ist jedoch mittlerweile auch serverseitig eine feste Größe. Im Jahr 2009 wurde die Plattform Node.js veröffentlicht, die sich im Kern auf die V8-Engine aus Chrome stützt. Neben diesem Kern aus nativen Bibliotheken bildet eine Reihe von Modulen, die in JavaScript implementiert sind, die Basis der Plattform. Über diese Module erhalten Sie beispielsweise Zugriff auf das Dateisystem oder auf Netzwerkressourcen.

Node.js verfolgt die Idee einer leichtgewichtigen Plattform, die Ihnen eine Low-Level-Schnittstelle auf das System zur Verfügung stellt, auf dem Sie den Prozess ausführen. Um alle weiteren Features, die auf dieser Schnittstelle aufsetzen, wie beispielsweise Datenbankzugriffe oder Backend-Frameworks, müssen Sie sich selbst kümmern oder das Ökosystem in Form eines Pakets nutzen.

Node.js ist zwar nicht die einzige serverseitige JavaScript-Plattform, jedoch die älteste und aktuell am weitesten verbreitete. Aus diesem Grund werden wir uns hier vor allem mit Node.js beschäftigen. Andere Plattformen wie Deno oder Bun verfolgen ähnliche Paradigmen, sodass Sie vieles von dem, was Sie über Node.js lernen, auch auf die anderen Plattformen anwenden können.

Eine Applikation mit Node.js ausführen

Die einfachste Variante, eine Applikation mit Node.js auszuführen, ist, wenn Sie eine JavaScript-Datei erzeugen. Der Name der Einstiegsdatei in eine solche Applikation lautet üblicherweise index.js, seltener app.js oder main.js. Diese Datei kann beliebigen JavaScript-Code enthalten, den Node.js dann ausführt. Als konkretes Beispiel für den Einstieg in Node.js nutzen Sie den Quellcode der Clientseite. In Listing 1.11 finden Sie den Code, der ohne weitere Modifikationen auch in Node.js ausführbar ist.

```
function add(a, b) {
  const result = a + b;
  return result;
}

const number1 = 3.14;
const number2 = 42;

const sum = add(number1, number2);
console.log(sum);
```

Listing 1.11: Beispielcode für Node.js

Haben Sie diesen Quellcode in einer Datei mit dem Namen index.js gespeichert, können Sie ihn auf der Kommandozeile ausführen. Wechseln Sie dafür in das Verzeichnis, in dem die Datei liegt, und setzen Sie das folgende Kommando ab: `node index.js`. Als Ausgabe erhalten Sie auf der Kommandozeile die Zahl `45.14`.

Intern liest Node.js die Datei, die Sie beim Kommando angegeben haben, ein, verarbeitet sie und führt sie schließlich aus. Das Ergebnis kann entweder eine direkte Ausgabe, wie hier im Beispiel, oder ein lange laufender Prozess sein. Die zweite Variante ist für Web-Applikationen deutlich öfter anzutreffen, da Node.js meist als Serverprozess verwendet wird und dieser in der Regel so lange läuft, bis er von außen beendet oder durch einen Fehler zum Absturz gebracht wird. Anders sieht die Situation aus, wenn Sie Node.js als Grundlage für Entwicklungswerkzeuge verwenden, die Quellcode analysieren oder modifizieren. In diesem Fall starten Sie den Prozess, er verrichtet seine Arbeit und wird dann wieder beendet.

Wie Sie im Beispiel gesehen haben, können Sie auch in Node.js mit `console.log` Ausgaben auf der Konsole erzeugen. Die Plattform verfügt jedoch auch über einen integrierten Debugger und dieser unterscheidet sich in seinen Features kaum von dem, den Sie clientseitig kennengelernt haben.

Debugging von Node.js-Applikationen

Nachdem sich Node.js und Chrome die gleiche JavaScript-Engine teilen, ist es auch nicht verwunderlich, dass sie über die gleichen Möglichkeiten zum Debuggen verfügen. Den integrierten Debugger können Sie mit dem Kommando »node inspect index.js« starten. Dabei interagieren Sie über die Kommandozeile mit dem Debugger und Ihrer Applikation. Diese Art der Fehlersuche funktioniert zwar, ist jedoch wenig komfortabel. Eine bessere Lösung besteht darin, dass Sie mit `node --inspect index.js` den V8 Inspector aktivieren. Dieser erlaubt es Ihnen, sich mit den Chrome-Entwicklerwerkzeugen über das Chrome-DevTools-Protokoll mit Ihrem Node.js-Prozess zu verbinden, sodass Sie Ihre aus dem Frontend gewohnte Debugging-Umgebung auch hier verwenden können. Im aktuellen Beispiel startet Node.js zwar im Debug-Modus, durchläuft jedoch das Skript so schnell, dass Sie nicht die Möglichkeit haben, sich mit den Entwicklerwerkzeugen zu verbinden. Die Lösung besteht in diesem Fall darin, dass Sie den Prozess mit dem Kommando `node --inspect-brk` starten. Diese Option sorgt dafür, dass der Debugger zu Beginn der Ausführung anhält und Ihnen so die Möglichkeit gibt, die Entwicklerwerkzeuge zu verbinden und Breakpoints zu setzen.

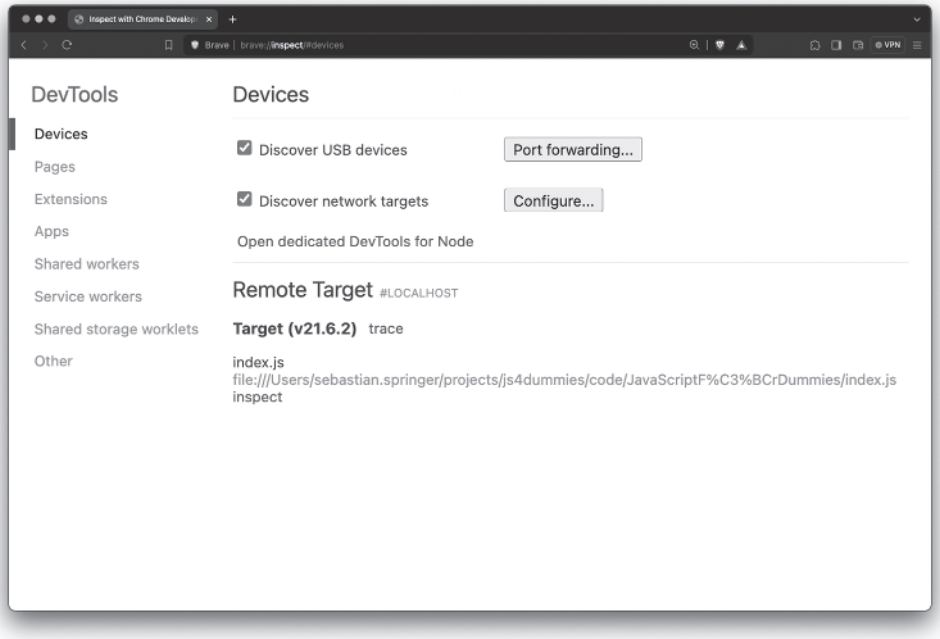

Abbildung 1.7: JavaScript-Debugging in VSCode

Um Chrome mit Node.js zu verbinden, geben Sie in der Adresszeile von Chrome chrome://inspect ein. Daraufhin zeigt Ihnen der Browser eine Liste von Node.js-Instanzen, mit denen Sie sich verbinden können. In Abbildung 1.7 sehen Sie ein Beispiel der Instanzliste.

Wählen Sie bei einer der Instanzen »Inspect« aus, öffnet Chrome die Entwicklerwerkzeuge für Sie und Sie können sie analog zum Debugging im Client verwenden.

Debugging aus der Entwicklungsumgebung heraus

Wie beim clientseitigen JavaScript müssen Sie auch in Node.js nicht auf die Annehmlichkeiten Ihrer Entwicklungsumgebung beim Debuggen verzichten und können den Debugging-Prozess auch direkt von dort aus starten. Dies funktioniert in allen gängigen Entwicklungsumgebungen und ist in der Regel in der Dokumentation der jeweiligen Software gut nachvollziehbar beschrieben.

Ich zeige Ihnen an dieser Stelle wieder exemplarisch in Visual Studio Code, wie Sie vorgehen können. Im ersten Schritt sollten Sie einen Breakpoint setzen. Dies erreichen Sie entweder über ein `debugger`-Statement oder indem Sie links neben die Nummer der gewünschten Zeile klicken. Öffnen Sie danach die Einstiegsdatei Ihrer Applikation und wählen Sie anschließend im Menü »Run« den Punkt »Start Debugging« aus. Alternativ können Sie die Taste F5 verwenden. Visual Studio Code gibt Ihnen dann eine Auswahl von Debuggern. Klicken Sie auf den Eintrag »Node.js«. Daraufhin aktiviert die Entwicklungsumgebung die Debugging-Ansicht und stellt Ihnen Kontrollelemente zur Navigation und weitere Informationen wie die Scope-Ansicht oder eine Liste von Breakpoints zur Verfügung.