URL:  
https://hosting.1und1.de/digitalguide/websites/web-entwicklung/einfuehrung-in-nodejs/  
Category:   
Erweiterung: node.js Vorstellung und Tutorial  
KWs:   
vorhanden: Node.js (34), node.js tutorial (3) / neu: node.js tutorial deutsch, was ist node.js, node.js installieren, node.js server tutorial, node.js einführung  
Internal links:  
Yes  
Title:  
Node.js-Tutorial auf Deutsch | 1&1  
Description:  
Mit Node.js schreiben Sie Server-Anwendungen in der beliebten Skriptsprache JavaScript und profitieren von einer asynchronen, eventgesteuerten Architektur.

Number of words: Target / Actual  
Min200- Max1000 / 9684  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Node.js-Einführung: Die JavaScript-Laufzeitumgebung im Überblick

Kommen serverseitige Programmiersprachen wie C oder PHP zum Einsatz, wird Programmcode in der Regel sequenziell abgearbeitet. Das bedeutet, dass ein Server mit der Umsetzung einer neuen Anweisung im Code erst dann beginnt, wenn die vorhergehende Anweisung ausgeführt wurde und ein Ergebnis geliefert hat. Man spricht in diesem Fall auch von einer **synchronen Bearbeitung**. Die Ausführung weiteren Codes wird dabei so lange gestoppt, bis der aktuelle Vorgang beendet wurde. Dies kann bei aufwendigen Operationen wie Zugriffen auf das Dateisystem, Datenbanken oder Webservices zu deutlichen Verzögerungen führen.

Viele Programmiersprachen, Laufzeitumgebungen und auf diesen basierende Implementierungen unterstützen daher die Möglichkeit, Vorgänge in sogenannten **Threads** parallel durchzuführen. Dabei handelt es sich um einzelne Ausführungsstränge im Rahmen eines Prozesses, mit denen sich Aktionen abarbeiten lassen, während der übrige Code weiter ausgeführt wird. Der Nachteil dieser Methode: Je mehr Threads gestartet werden, desto mehr CPU-Zeit und Speicherplatz im RAM wird benötigt. Mit anderen Worten: **Multithreading** ist ressourcenintensiv. Zudem gehen zusätzliche Threads mit einem deutlich höheren Programmieraufwand einher. Umgehen lassen sich diese Probleme mit einer serverseitigen Implementierung von JavaScript, die eine asynchrone Ausführung von Programmcode ermöglicht. Die Grundlagen dafür werden von der Laufzeitumgebung **Node.js** bereitgestellt.

## Was ist Node.js?

Bei Node.js handelt es sich um eine **Software-Plattform mit eventbasierter Architektur**, die es ermöglicht, die ursprünglich für den clientseitigen Einsatz entwickelte Skriptsprache **JavaScript serverseitig** zu verwenden. Diese kommt somit in der gleichen Weise zum Einsatz wie PHP, Java, .NET, Ruby oder Python, um Code für den Server zu schreiben. Verwendung findet Node.js in der Entwicklung serverseitiger JavaScript-Anwendungen, die große Datenmengen in Echtzeit bewältigen müssen. Beliebt ist die Laufzeitumgebung zur Realisation leichtgewichtiger Webserver.

Das plattformübergreifende Software-Projekt wurde im Jahr 2009 von Ryan Dahl ins Leben gerufen und basiert im Kern auf **Googles JavaScript-Engine V8**, die auch im Webbrowser Chrome zum Einsatz kommt. Von der Firma Joyent gestartet, untersteht das Projekt seit 2015 in Form der Node.js-Foundation einem gemeinnützigen Konsortium: der Linux-Foundation. Aktuelle Versionen stehen für Microsoft Windows, Mac OS und Linux zur Verfügung.

Node.js beinhaltet eine Bibliothek diverser **JavaScript-Module**, die durch eine einfache Funktion geladen werden können und als vorgefertigte Bausteine für die Entwicklung von Webanwendungen bereitstehen. Ein Beispiel ist das HTTP-Modul, das es ermöglicht, mit einer einzigen Funktion einen rudimentären Webserver zu erzeugen. Darüber hinaus lassen sich mit dem integrierten Paket-Manager **npm (Node Package Manager)** zusätzliche Module nachinstallieren.

## Node.js installieren

Node.js steht auf der offiziellen Website des Software-Projekts [Zur offiziellen Projektseite von Node.js](https://nodejs.org/) je nach Betriebssystem als Installer und/oder Binärpaket zum Download bereit. Außer für Windows, Linux und macOS für x86/x64-PCs stellt die Node.js-Foundation dort den Quellcode der Software und Binärpakte für ARM-, Power- und z-Systems-Plattformen zur Verfügung. Auch Versionen für AIX und SunOS sowie ein Docker-Image lassen sich herunterladen.

Die folgende Tabelle zeigt eine Übersicht der unterstützten Plattformen und Architekturen:

|  |  |
| --- | --- |
| **Plattform** | **Unterstützte Architekturen** |
| Windows Installer (.msi) | 32-bit  64-bit |
| Windows Binary (.zip) | 32-bit  64-bit |
| macOS Installer (.pkg) | 64-bit |
| macOS Binaries (.tar.gz) | 64-bit |
| Linux Binaries (x86/x64) | 32-bit  64-bit |
| Linux Binaries (ARM) | ARMv6  ARMv7  ARMv8 |
| SunOS Binaries | 32-bit  64-bit |
| Docker | Docker Image |
| Linux on Power Systems | 64-bit le  64-bit be |
| Linux on System z | 64-bit |
| AIX on Power Systems | 64-bit |

Im Folgenden veranschaulichen wir den Einsatz von Node.js am Beispiel einer lokalen Windows-Installation. Wie Sie Node.js auf Ihrem Windows-Rechner installieren, zeigen wir Ihnen in **drei einfachen Schritten**.

### Schritt 1: Software-Paket herunterladen

Um Node.js auf einem Windows-PC zu installieren, wird lediglich der Windows-Installer heruntergeladen, wahlweise in der 32- oder 64-Bit-Version. Neben der aktuellen Version v8.4.0 steht der LTS-Release 6.11.3 (Long-Term-Support) zur Verfügung, der auch Grundlage dieses Node.js-Tutorials ist. Der Long-Term-Support garantiert Ihnen Bugfixes für 1,5 Jahre sowie Updates gegen kritische Sicherheitslücken für das darauffolgende Jahr.

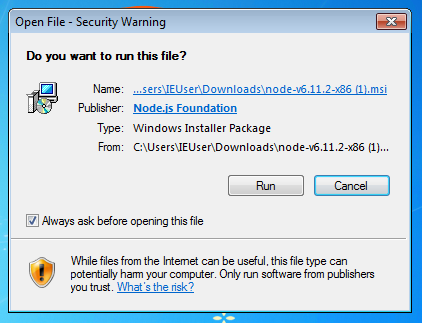


[Alt-Text: Projekt-Website von Node.js]

[BU: Während Ihnen die aktuelle Version von Node.js die neuesten Features der Laufzeitumgebung bietet, steht mit dem LTS-Release eine stabile Softwareversion für den Produktiv-Einsatz bereit / Quelle: https://nodejs.org/en/]

### Schritt 2: Node.js installieren

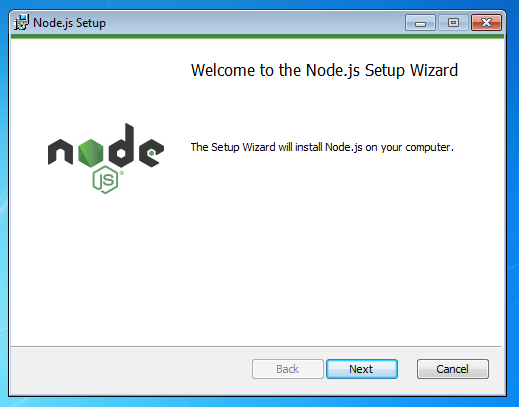
Nach einem Doppelklick auf die **.msi-Datei** öffnet sich eine Windows-Sicherheitswarnung. Bestätigen Sie mit einem Klick auf „Run“, dass Sie die ausgewählte Datei ausführen möchten.



[Alt-Text: Windows-Sicherheitswarnung]

[BU: Windows-Sicherheitswarnung: Nutzen Sie lediglich Software aus vertrauenswürdigen Quellen]

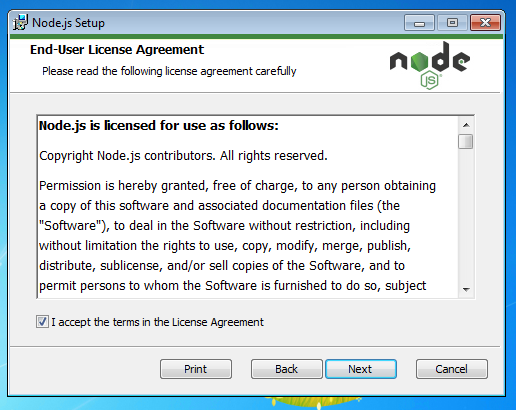
Es öffnet sich das **Installations-Set-up** von Node.js.



[Alt-Text: Installations-Set-up von Node.js]

[BU: Das Installations-Set-up von Node.js führt Sie durch die Installation der Laufzeitumgebung]

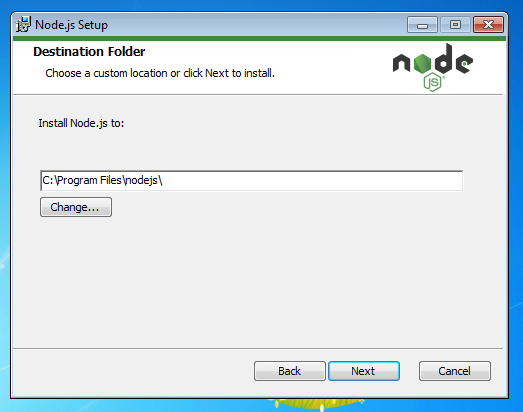
Bevor Sie die Installation starten können, müssen Sie zunächst die **Lizenzbestimmungen** der Software akzeptieren.



[Alt-Text: Node.js-Installations-Set-up: die Lizenzbestimmungen]

[BU: Die Laufzeitumgebung Node.js steht unter MIT-Lizenz]

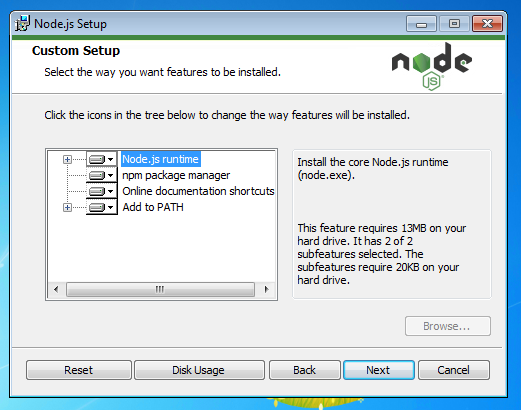
Bestimmen Sie, in welchem Ordner Ihres Dateisystems Node.js installiert werden soll.



[Alt-Text: Node.js-Installations-Set-up: Wahl des Installationspfads]

[BU: Wir empfehlen die Standardinstallation im Ordner Programme]

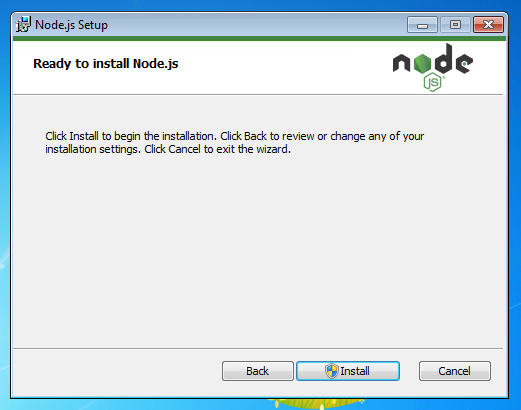
Installieren Sie **Node.js inklusive npm-Paket-Manager**.



[Alt-Text: Node.js-Installations-Set-up: Auswahl der zu installierenden Komponenten]

[BU: Wir empfehlen, Node.js gemäß der der voreingestellten Standardinstallation zu installieren]

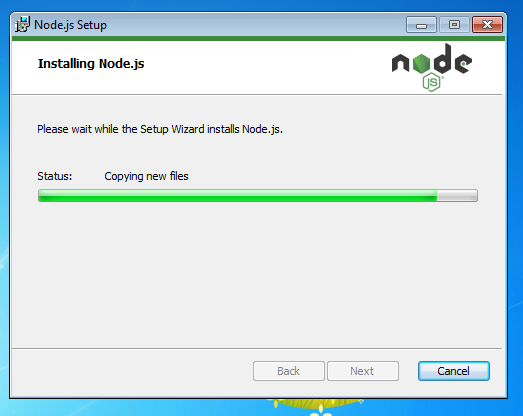
Starten Sie die Installation mit einem Klick auf „Install“.



[Alt-Text: Node.js-Installations-Set-up: Installation starten]

[BU: Beachten Sie: Für die Installation des Programms werden Administratorrechte benötigt]

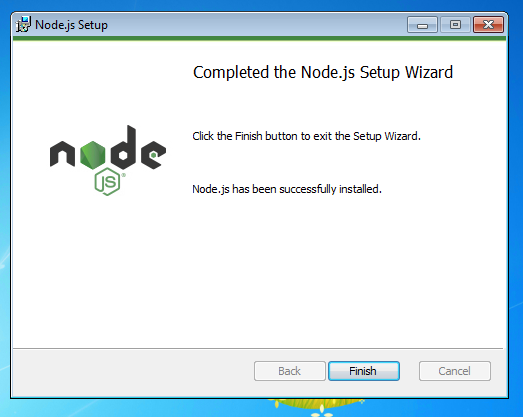
Die Installation der Laufzeitumgebung sollte in wenigen Sekunden abgeschlossen sein.



[Alt-Text: Node.js-Installations-Set-up: Installationsstatus]

[BU: Der Fortschritt des Installationsprozesses wird Ihnen im Set-up-Fenster angezeigt]

Schließen Sie den Installationsprozess mit einem Klick auf den Finish-Button ab.



[Alt-Text: Node.js-Installations-Set-up: Zusammenfassung]

[BU: die Installation von Node.js war erfolgreich]

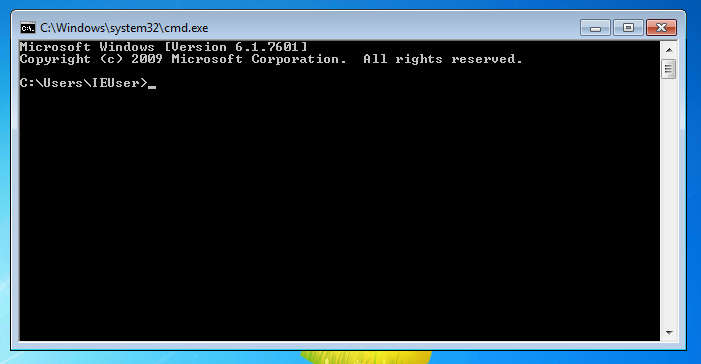
Ihre lokale Node.js-Installation ist nun einsatzbereit.

### Schritt 3: Installation testen

Ob die Installation erfolgreich war, überprüfen Sie, indem Sie Node.js starten und den ersten einfachen JavaScript-Code ausführen. Die Laufzeitumgebung stellt Ihnen dazu **zwei Betriebsmodi** zur Verfügung: Entweder Sie nutzen Node.js im interaktiven Modus oder Sie führen Anwendungscode aus einer vorher angelegten JavaScript-Datei (.js) aus.

* **Interaktiver Modus:** Um Node.js im interaktiven Modus zu nutzen, öffnen sie die Windows-Eingabeaufforderung (*cmd.exe*) und geben den Befehl *node* ein. Der sogenannte REPL (Read-Evaluation-Print-Loop), ein interaktiver Kommandozeileninterpreter (Command Prompt) wird gestartet. Dieser liest JavaScript-Code ein (Read), wertet diesen aus (Evaluation) und gibt das Ergebnis zurück (Print). Anschließend fängt das Ganze wieder von vorne an (Loop).
* **JavaScript-Code aus Datei ausführen:** Um der Laufzeitumgebung JavaScript-Code aus einer Datei zu übergeben, nutzen Sie den Befehl *node* in Kombination mit dem Pfad zur jeweiligen Datei.

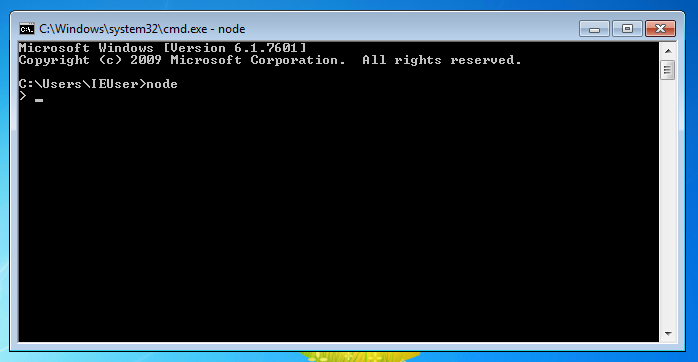
Testen Sie zunächst den interaktiven Modus. Öffnen Sie dazu die Windows-Eingabeaufforderung.



[Alt-Text: Die Windows-Eingabeaufforderung cmd.exe]

[BU: Unter Windows öffnen Sie die Eingabeaufforderung am einfachsten durch die Tastenkombination [Windows-Taste] + [r] und die Eingabe von „cmd“.

Starten Sie die JavaScript-Laufzeitumgebung über den **Befehl *node***.



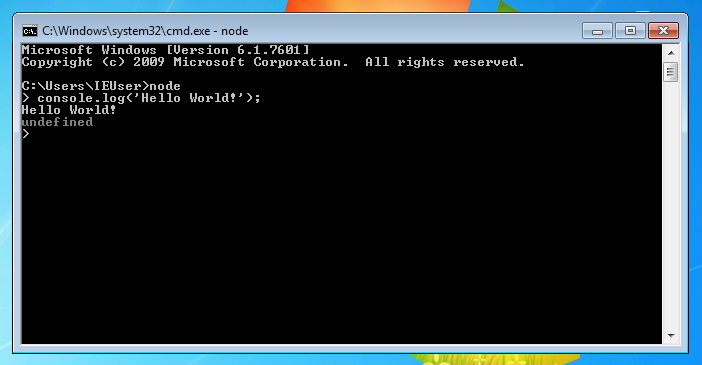
[Alt-Text: Der interaktive Command-Prompt von Node.js]

[BU: Der Wechsel von der Windows-Eingabeaufforderung zum Node.js-Command-Prompt wird Ihnen durch den Zusatz „node“ im Kopf des Terminalfensters angezeigt]

Sie befinden sich nun im interaktiven Command-Prompt von Node.js. Um zu überprüfen, ob Node.js korrekt installiert wurde und einwandfrei funktioniert, empfehlen wir die **Funktion *console.log()***. Diese nimmt Werte entgegen und gibt sie auf dem Terminal aus. Testen Sie dies beispielsweise mit einem einfachen „Hello World!“-Skript.

*console.log('Hello World!');*

In den Node.js-Prompt eingegeben und mit der Eingabetaste bestätigt, weist die oben stehende Funktion Node.js an, die Zeichenfolge (String) „Hello World!“ ins Terminal zu schreiben. Das Ergebnis sehen Sie auf folgendem Screenshot:

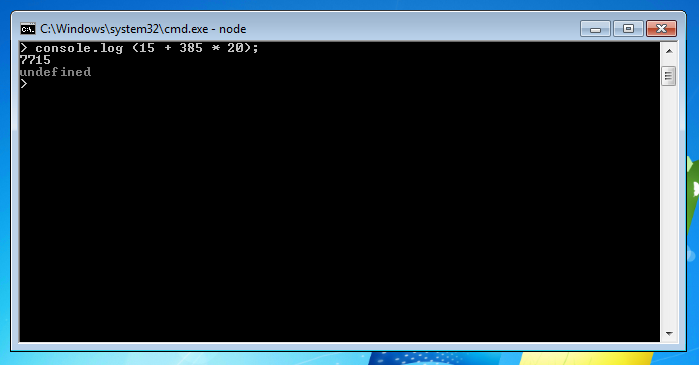


[Alt-Text: Node.js testen: Die Funktion console.log]

[BU: Die Ausgabe „Hello World!“ zeigt an, dass die Installation von Node.js erfolgreich war]

Die Terminalausgabe zeigt, dass Node.js korrekt installiert wurde und in der Lage ist, JavaScript-Code interaktiv auszuführen. Alternativ lässt sich mit *console.log()* z. B. das Ergebnis einer Rechenaufgabe ins Terminal schreiben:

*console.log (15 + 385 \* 20);*

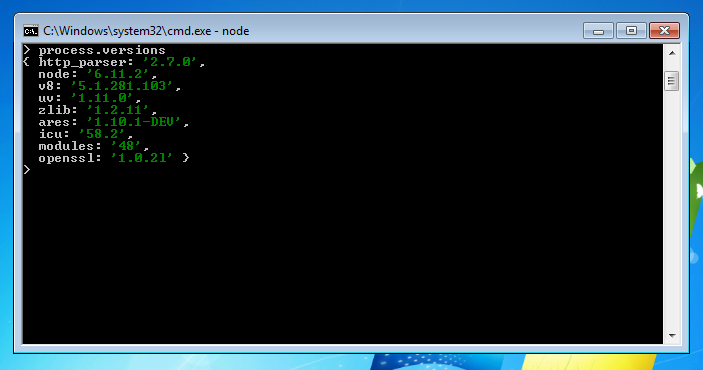
**

[Alt-Text: Installation von Node.js]

[BU: Überprüfen Sie mit einer einfachen Rechenaufgabe, ob Node.js richtig funktioniert]

Möchten Sie sich detaillierte **Informationen über die installierte Node.js-Version** und deren Module ausgeben lassen, kommt folgender Ausdruck (expression) zum Einsatz:

*process.versions*

**

[Alt-Text: Node.js testen: Der Befehl „process.versions“]

[BU: Der Befehl *process.versions* zeigt an, welche Node.js-Version installiert wurde]

Möchten Sie **Node.js beenden** und zurück in die Windows-Eingabeaufforderung wechseln, nutzen Sie die Tastenkombination [STRG] + [C]. Es sind zwei Anschläge notwendig. Dies verhindert, dass Sie den interaktiven Command-Prompt unbeabsichtigt verlassen.

Beachten Sie: Kommt – wie oben beschrieben – das Installations-Set-up zum Einsatz, kann pro System lediglich eine Version der JavaScript-Laufzeitumgebung installiert werden. Bei der Installation einer neuen Node.js-Version wird die ältere Version überschrieben. Möchten Sie mehr als eine Version von Node.js gleichzeitig betreiben, benötigen Sie einen **Versionsmanager** wie nvm [Weitere Infomationen zu nvm auf GitHub] (https://github.com/creationix/nvm) (Node Version Manager).

Wie die parallele Installation mehrerer Node.js-Versionen mit nvm vonstattengeht, zeigt Golo Roden von *the native web* in einem Video-Tutorial auf Vimeo.

<iframe src="https://player.vimeo.com/video/198445904?color=ff0099" width="640" height="400" frameborder="0" webkitallowfullscreen mozallowfullscreen allowfullscreen></iframe>

<p><a href="https://vimeo.com/198445904">tech:lounge, Einführung in Node.js, Folge 1: Node.js installieren</a> from <a href="https://vimeo.com/thenativeweb">the native web</a> on <a href="https://vimeo.com">Vimeo</a>.</p>

[Tipp: Die JavaScript-Laufzeitumgebung Node.js steht Ihnen auf jedem 1&1 Cloud Server [Cloud Server | Günstig Cloud Hosting einrichten | 1&1] (https://hosting.1und1.de/dynamic-cloud-server) über das App-Center zur Verfügung.]

## Node.js-Module

Auf Basis der Laufzeitumgebung V8 schreiben Sie mit Node.js effiziente Webserver und andere Netzwerkanwendungen in der beliebten Skriptsprache JavaScript. Dabei überzeugt Node.js mit einem bewusst kompakt gehaltenen Aufbau, bei dem lediglich Kernfunktionalitäten wie die Interaktion mit dem Betriebssystem, die Netzwerkkommunikation oder Verschlüsselungsmechanismen als **Basismodule** in die Laufzeitumgebung integriert wurden. Alle weiteren Funktionalitäten werden als Erweiterungsmodule eingebunden. Anwender können dabei auf ein **umfangreiches Sortiment von Drittanbietermodulen** zurückgreifen oder eigene Module programmieren.

Um ein beliebiges Modul in eine Node.js-Anwendung zu laden, benötigen Sie lediglich die **Funktion *require()***.Diese erwartet einen String, der als Parameter angibt, welches Modul geladen werden soll. In der Regel handelt es sich um den Namen des Moduls. Kommt *require()* in Kombination mit einem Modulnamen zum Einsatz, durchsucht Node.js diverse Verzeichnisse nach Modulen, die als Basismodule Teil von Node.js sind oder die mit npm projektspezifisch (lokal) oder projektübergreifend (global) installiert wurden. Diese Module werden ohne Pfandangabe gefunden. Nicht durchsucht wird das aktuelle Arbeitsverzeichnis. Möchten Sie ein selbstgeschriebenes Modul aus Ihrem aktuellen Projektverzeichnis laden, wird eine relative Pfandangabe mit *./* benötigt.

Folgende Beispiele zeigen beide **Grundschemata**, nach denen Module in Node.js-Anwendungen geladen werden:

*const modulname = require('modulname');*

*const modulname = require('./modulname');*

In diesem Node.js-Tutorial zeigen wir Ihnen anhand von Beispielen, wie Sie Basismodule verwenden, Drittanbietermodule einbinden und eigene Node.js-Module erstellen.

### Basismodule verwenden

Soll z. B. das Basismodul *os* geladen werden, sähe der Code folgendermaßen aus:

*const os = require('os');*

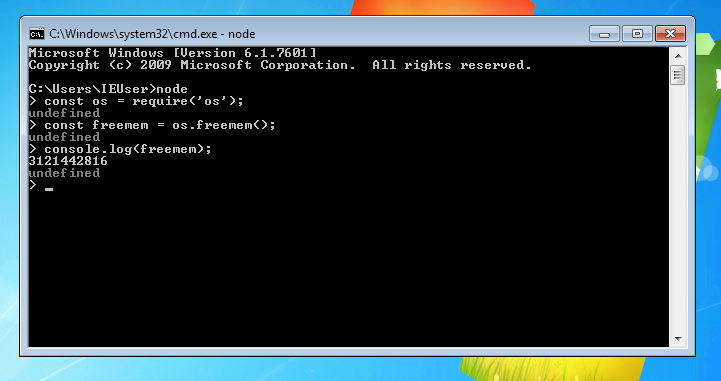
Das **Modul *os***bietet diverse Funktionen, mit denen sich **betriebssystembezogene Informationen** ausgeben lassen. Möchte ein Anwender beispielsweise die Menge an freiem Systemspeicher in Byte erfahren, käme folgender Code zum Einsatz:

*const os = require('os');*

*const freemem = os.freemem();*

*console.log(freemem);*

In der ersten Zeile wird das Modul *os* in die gleichnamige Konstante *os* geladen. Der Codeabschnitt *os.freemem()* in Zeile 2 führt eine der vom Modul *os* unterstützten Funktionen aus und dient dazu, die **Menge an freiem Systemspeicher in Byte** zu ermitteln. Der ermittelte Wert wird in der Konstante *freemem* gespeichert. In der dritten Zeile schließlich wird die bereits bekannte Funktion *console.log()* verwendet, um den in der Konstante *freemem* gespeicherten Wert ins Terminal zu schreiben.



[Alt-Text: Node.js-Module: Basismodule]

[BU: Das Modul *os* gibt betriebssystembezogene Informationen aus]

Wie die Ausgabe im Terminal zeigt, stehen auf unserem Testsystem aktuell 3,1 Gigabyte Systemspeicher zur Verfügung.

Mit anderen Funktionen des *os*-Moduls wie *os.networkInterfaces()* oder *os.hostname()* lässt sich eine Liste mit Netzwerkschnittstellen ausgeben oder der Hostname ermitteln.

Die nachstehende Tabelle enthält eine **Auswahl der wichtigsten Basismodule** der JavaScript-Laufzeitumgebungen inklusive Beschreibung.

|  |  |  |
| --- | --- | --- |
| **Node.js-Basismodule (Auswahl)** | | |
| **Modul** | **Beschreibung** | **Aufruf** |
| ***assert*** | Das Basismodul *assert* bietet einen Satz einfacher Assertionstests, mit denen Entwicklern überprüfen, ob die Methodenparameter und Invarianten eines Programms korrekt sind.  Assertions (Zusicherungen) dienen dazu, bestimmte Aussagen über den Zustand eines Programms zu verifizieren und sicherzustellen, dass diese eingehalten werden. Handelt es sich um Aussagen, die über die Ausführung bestimmter Programmbefehle hinweg unveränderlich gelten und somit bei jedem Schleifendurchlauf erfüllt sein müssen, spricht man von Invarianten.  *assert* ist in erster Linie für den internen Gebrauch von Node.js gedacht, lässt sich über die Funktion *require()* aber auch in eigenen Anwendungscode einbinden. | *require('assert')* |
| ***buffer*** | Das *buffer*-Modul stellt Node.js einen Mechanismus zur Verfügung, der es ermöglicht, Binärdatenströme zu verarbeiten. | Das Buffer-Modul ist eine globale Klasse innerhalb von Node.js. Es ist somit nicht nötig, dass Modul via *require()* aufzurufen. |
| ***child\_process*** | Das Basismodul *child\_process* erweitert Node.js um eine Funktion, die es ermöglicht, Kind-Prozesse zu erzeugen. *child\_process* funktioniert ähnlich wie die Unix-Funktion *popen(3)* [Mehr zu popen(3) unter http://man7.org] (http://man7.org/linux/man-pages/man3/popen.3.html). | *require('child\_process')* |
| ***cluster*** | Normalerweise läuft Node.js in einem einzelnen Thread auf einem einzelnen Prozessor des zugrundeliegenden Rechners. Entwickler, die die Vorteile eines Multi-Core-Systems nutzen möchten, müssen das Modul *cluster* einbinden. Dieses stattet Node.js mit einem Mechanismus aus, der es ermöglicht, ein Netzwerk separater Prozesse zu erstellen, die sich einen gemeinsamen Server-Port teilen. Werden diese Prozesse auf die verschiedenen Kerne des zugrundeliegenden Rechners verteilt, steht Node.js die volle Leistung des Systems zur Verfügung. | *require('cluster')* |
| ***console*** | Das Basismodul *console* stellt Entwickler eine einfache Konsole zur Verfügung, die zwei Komponenten umfasst:   * eine Klasse *console*, die Funktionen bereitstellt, mit denen sich Daten in einen Node.js-Stream schreiben lassen * eine globale Konsole, die Funktionen bereitstellt, mit denen sich Daten auf die Standardausgabe (*stdout*) oder die Standardfehlerausgabe (*stderr*) schreiben lassen.   Das *console*-Modul kommt in der Regel im Rahmen des Debuggings zum Einsatz. | Die Klasse *console* wird via *require('console')* aufgerufen.  Die globale Konsole kann ohne weitere Vorbereitungen genutzt werden. |
| ***crypto*** | Das *crypto*-Modul umfasst diverse Wrapper für verschiedene Funktionalitäten des quelloffenen TLS-Toolkits OpenSSL [Mehr zu OpenSSL auf der Projekt-Website] (https://www.openssl.org/) inklusive Hashing, HMAC, Verschlüsselung, Entschlüsselung, Signieren und Verifizieren. | *require('crypto')* |
| ***debugger*** | Mit dem Modul *debugger* enthält Node.js ein Werkzeug zum Diagnostizieren und Auffinden von Fehlern in JavaScript-Dokumenten. | Um das Modul *debugger* zu verwenden, starten Sie Node.js mit dem Argument *debug*, gefolgt von dem JavaScript-Dokument, das untersucht werden soll. Beispiel:  *node debug myscript.js* |
| ***dns*** | Das Basismodul *dns* bietet zwei Arten von Funktionen:   * Funktionen, die im Rahmen der Namensauflösung auf das zugrundeliegende Betriebssystem zurückgreifen. * Funktionen, die im Rahmen der Namensauflösung eine Verbindung zu einem DNS-Server aufbauen. | *require('dns')* |
| ***events*** | Entwickler, die im Rahmen einer Node.js-Anwendung Ereignisseauslösen oder verarbeiten möchten, greifen dafür auf das Basismodul *Events* zurück. | *require('events')* |
| ***fs*** | Das Basismodul *fs* (file system) ermöglicht es Node.js, auf das Dateisystem zuzugreifen, und enthält diverse Funktionen zum Lesen und Schreiben von Dateien. | *require('fs')* |
| ***http*** | Mit *http* bietet Node.js ein Basismodul, das alle Funktionen enthält, die Entwickler benötigen, um einfache HTTP-Server und -Clients zu erstellen. | *require('http')* |
| ***net*** | Das Basismodul *net* stellt Entwicklern Funktionen bereit, mit denen sich TCP- oder IPC-Server und -Clients erstellen lassen. | *require('net')* |
| ***os*** | Bei *os* handelt es sich um ein Basismodul, das diverse Funktionen für grundlegende Systemoperationen bieten. Mit *os.freemem()* haben Sie eine dieser Funktionen im Rahmen dieses Node.js-Tutorials bereits kennengelernt. | *require('os')* |
| ***path*** | Entwickler, die das Basismodul *path* aufrufen, erhalten Zugriff auf Funktionen, die verschiedene Operationen im Zusammenhang mit Datei- und Verzeichnispfaden ermöglichen. Kommt Node.js auf einem Windows-System zum Einsatz, sorgt das Path-Modul beispielsweise dafür, dass Datei- und Verzeichnispfade im Windows-Stil korrekt interpretiert werden. | *require('path')* |
| ***process*** | Bei *process* handelt es sich um ein globales Objekt, das Entwicklern Informationen über den aktuellen Node.js-Prozess liefert und es erlaubt, diesen zu kontrollieren. | Als globales Objekt kann *process* jederzeit verwendet werden, ohne dass das Basismodul via *require()* aufgerufen werden muss. |
| ***querystring*** | Das Basismodul *querystring* bietet Funktionen, mit denen sich URL-Query-Strings analysieren und formatieren lassen. | *require('querystring')* |
| ***stream*** | Als Stream wird in der Node.js-Terminologie eine abstrakte Schnittstelle bezeichnet, die es ermöglicht, mit Streaming-Daten zu arbeiten. Das Basismodul *stream* bietet eine API, mit der sich Objekte erstellen lassen, die die Stream-Schnittstelle enthalten. | *require('stream')* |
| ***timers*** | Das Basismodul *timers* stellt eine globale Schnittstelle für das Scheduling von Funktionen bereit, die zu einem späteren Zeitpunkt aufgerufen werden sollen. | Die Funktionen des *timers*-Moduls arbeiten global, dieses muss somit nicht per *require()* aufgerufen werden. |
| ***url*** | Das *url*-Basismodul bietet Funktionen für die URL-Auflösung und das Parsen von URLs. | *require('url')* |
| ***util*** | Bei *util* handelt es sich um ein Basismodul, das in erster Linie die internen APIs von Node.js unterstützt. Bei Bedarf kann das Modul aber auch in andere Anwendungen eingebunden werden. | *require('util')* |
| ***vm*** | Das Basismodul *vm* stellt Schnittstellen bereit, die es erlauben, JavaScript-Code zu kompilieren und auf einer V8 Virtual Machine auszuführen. | *require('vm')* |
| ***zlib*** | Node.js bietet bereits in der Grundinstallation Kompressionsfunktionalitäten auf Basis von *Gzip* und *Deflate/Inflate*. Diese werden vom Basismodul *zlib* zur Verfügung gestellt. | require('zlib') |

Weitere Informationen zu diesen und anderen Basismodulen von Node.js finden Sie in der offiziellen **Dokumentation des Software-Projekts** [Zur Dokumentation von Node.js v6.11.3] (https://nodejs.org/dist/latest-v4.x/docs/api/).

### Drittanbietermodule einbinden

Die **Nachinstallation von Programmmodulen** erfolgt bei Node.js über den integrierten **Paket-Manager npm**. Dieser erlaubt Node.js-Nutzern den Zugriff auf die **npm-Registry**, ein communitygestütztes Online-Archiv für Node.js-Module. Erweiterungsmodule müssen somit nicht manuell von externen Seiten heruntergeladen und in ein entsprechendes Verzeichnis kopiert werden. Der Installationsaufwand beschränkt sich auf eine einzige Codezeile, die in die Konsole des Betriebssystems eingegeben wird (bei Windows-Betriebssystemen der Kommandozeileninterpreter *cmd.exe*). Benötigt wird lediglich der Befehl *npm install* und der Name des zu installierenden Moduls.

*npm install modulname*

[Hinweis: Per Default ist die Installation lokal, findet also innerhalb des aktuellen Verzeichnisses statt. Um npm-Module global zu installieren, braucht man die Option *-g* (oder *--global*).]

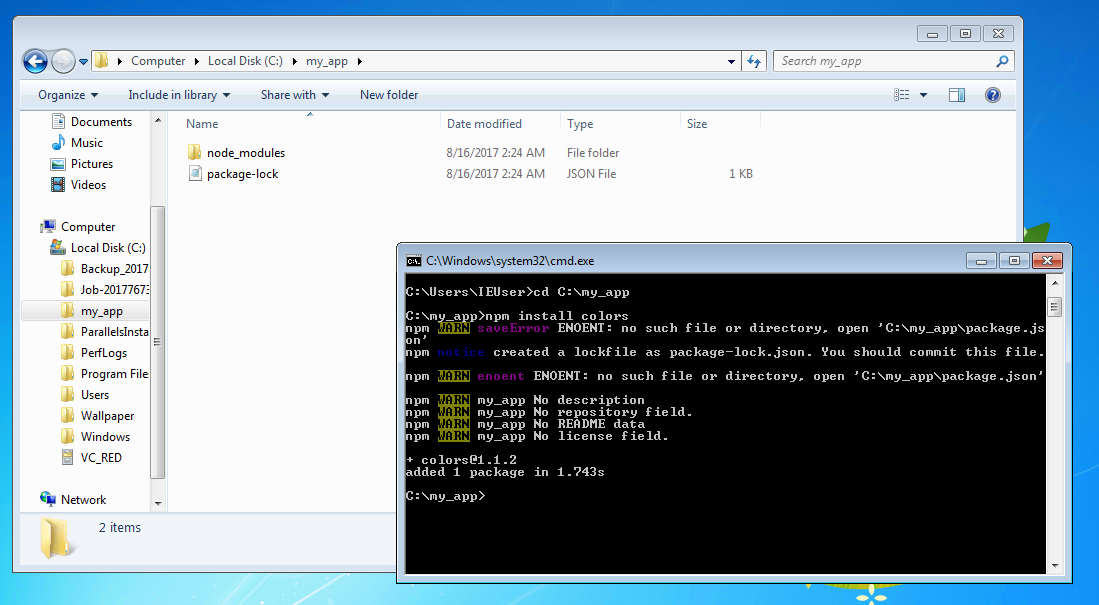
Alle verfügbaren Zusatzmodule für Node.js lassen sich über eine Suchfunktion auf der offiziellen Website von npm [Zur Website des Node Package Managers](https://www.npmjs.com/) einsehen.

Soll beispielsweise das **Drittanbietermodul *colors*** installiert werden, navigieren Sie in der Konsole des Betriebssystems in ein gewünschtes Verzeichnis und bestätigen folgenden Befehl mit der Eingabetaste:

*npm install colors*

[Hinweis: Es empfiehlt sich, für jede mit Node.js entwickelte Anwendung ein eigenes Anwendungsverzeichnis zu erstellen. In diesem Verzeichnis speichern Sie sämtliche lokalen Dateien, die im Rahmen der Programmausführung benötigt werden.]

Der folgende Screenshot zeigt die Installation des *color*-Moduls in das zuvor angelegte Verzeichnis *C:\my\_app*:



[Alt-Text: Terminalausgabe nach der Installation des Drittanbietermoduls *colors*]

[Bildunterschrift: Die Warnhinweise im Rahmen der Installation deuten auf fehlende Metadaten hin, können jedoch zunächst ignoriert werden]

Im Rahmen der Installation erstellt npm in Ihrem Anwendungsverzeichnis das Unterverzeichnis *node\_modules.* Dieses dient als **Speicherort für alle weiteren Module**, die Sie installieren möchten. Wird ein Modul im Verzeichnis *node\_modules* angelegt, ändert sich dadurch dessen Baumstruktur. Änderungen dieser Art werden in einer automatisch generierten JSON-Datei (JavaScript Object Notation), der sogenannten*package-lock*, erfasst.

[Hinweis: Im aktuellen Beispiel bekommen wir bei der Installation des *colors*-Modul diverse Warnhinweise angezeigt. Dies liegt daran, dass wir für unser Anwendungsverzeichnis *my-app* keine Metadaten-Datei – die sogenannte *package.json* – angelegt haben. Diese wird jedoch erst relevant, sobald Sie eigene Anwendungen als Module veröffentlichen möchten. Wie Sie eine *package.json* erstellen, erfahren Sie im Kapitel „Eigene Module veröffentlichen“.]

Um das nachinstallierte Modul in eine Anwendung zu laden, verwenden Sie die bereits eingeführte Funktion *require()*.Folgendes **Codebeispiel** zeigt, wie das *color*-Modul eingesetzt wird, um Ausgaben in der Node.js-Konsole farbig hervorzuheben:

*const colors = require(*'*colors*'*);*

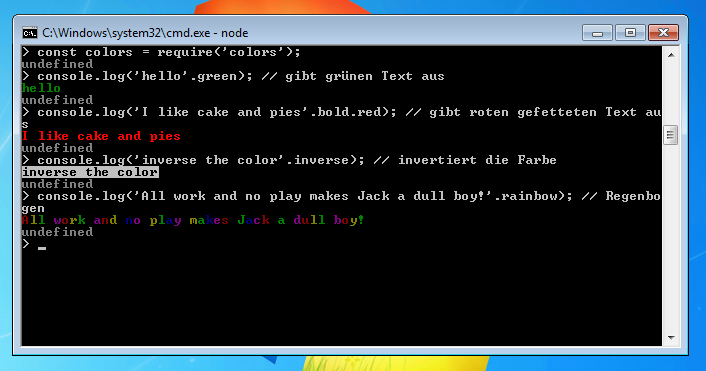
*console.log('hello'.green); // gibt grünen Text aus*

*console.log('I like cake and pies'.bold.red)*; *// gibt roten gefetteten Text aus*

*console.log('inverse the color'.inverse); // invertiert die Farbe*

*console.log('All work and no play makes Jack a dull boy!'.rainbow); // Regenbogen*

In der ersten Codezeile wird das Modul *colors* in die gleichnamige Konstante *colors* geladen. Die Zeilen 2 bis 5 beinhalten jeweils eine Funktion *console.log()*, deren Strings durch das *color*-Modul in unterschiedlicher Art hervorgehoben werden sollen. Möglich sind Farbanpassungen (z. B*. .red*, *.green*), Hervorhebungen (z. B. .*bold*, *.inverse*) und Sequenzen (z. B. *.rainbow*).



[Alt-Text: Node.js-Module: Das „colors“-Modul in der Anwendung]

[BU: Das Codebeispiel zeigt, wie das *colors*-Modul angewendet wird]

[Hinweis: In der Regel enthält jede Anwendung, die Sie mit Node.js entwickeln, ein eigenes *node\_modules*-Verzeichnis, das als Speicherort für Drittanbietermodule dient. Obwohl sich Module mit der Option *-g* oder *--global* prinzipiell auch global installieren lassen, bevorzugen Node.js-Entwickler im Allgemeinen die lokale Installation. Daraus ergibt sich der Nachteil, dass jedes Modul für jede Anwendung neu installiert werden muss. Wird ein bestimmtes Modul beispielsweise in zehn verschiedenen Anwendungen benötigt, muss dieses Modul zehn Mal auf der Festplatte gespeichert werden. Dies nimmt zusätzlichen Speicherplatz ein, verhindert jedoch Versionskonflikte, da jedes Anwendungsverzeichnis alle Abhängigkeiten enthält, die für den Programmablauf der entsprechenden Anwendung benötigt werden.]

### Eigene Node.js-Module

Der Anwendungscode, den Sie sich bei npmjs.com herunterladen können, unterscheidet sich im Grunde nicht von dem Code, den Sie auf Basis von Node.js selbst programmieren. Jeder, der die JavaScript-Laufzeitumgebung nutzt, hat die Möglichkeit, Anwendungen zu programmieren und diese als Module via npm mit der Community zu teilen. Wir zeigen Ihnen, wie Sie dabei vorgehen: Zunächst erstellen wir im Rahmen eines kurzen **Node.js-Server-Tutorials** mit nur acht Codezeilen einen eigenen Webserver und zerlegen diesen, um den Anwendungscode einzubinden. Anschließend bereiten wir das Programm für die **Veröffentlichung** vor.

#### Eigene Module erstellen

Eine Standardanwendung von Node.js ist die Entwicklung leichtgewichtiger Webserver. Ein simpler **„Hello World!“-Server** lässt sich dank des vorprogrammierten Moduls „http“ binnen Sekunden programmieren.

1. **Anwendungscode schreiben:** Alles, was Sie benötigen, um einen Webserver mit Node.js zu erstellen, sind die folgenden acht Zeilen JavaScript-Code. Geben Sie diese in einen Texteditor ein und speichern Sie die Anwendung als JavaScript-Datei in einem beliebigen Verzeichnis. In diesem Node.js-Server-Tutorial wird die Datei als *webserver.js* benannt und im Verzeichnis *my\_webserver* abgelegt.

*const http = require('http');*

*const port = 8080;*

*const server = http.createServer(function(req, res) {*

*res.writeHead(200, {*

*'Content-Type': 'text/html'*

*});*

*res.write('Hello World!');*

*res.end();*

*});*

*server.listen(port, () => {*

*console.log('Server is available under http://127.0.0.1:' + port + '/');*

*});*

In Codezeile 1 wird das **Webserver-Modul** *http* durch die Funktion *require()* geladen und in der gleichnamigen Konstante *http* gespeichert.

*const http = require('http');*

Anschließend wird in Zeile 2 die **Portnummer** 8080 definiert und in der Konstante *port* gespeichert.

*const port = 8080;*

Die folgenden Zeilen 4 bis 10 definieren mithilfe der Funktion *createServer()* des *http*-Moduls eine **Callback-Funktion**, die eingehende Browseranfragen (*Request, req*) entgegennimmt und eine Antwort zurückgibt (*Response, res*). Welche Antwort zurückgeschickt wird, definieren wir mithilfe der Funktionen *res.writeHead()* und *res.write()*. Während *res.write()* mit „Hello World!“die eigentliche Antwort enthält, stellt *res.writeHead()* einen Header bereit, der neben dem **HTTP-Statuscode 200** (OK) Meta-Informationen zur Antwort bereitstellt. Im aktuellen Beispiel wird die Antwort als Text klassifiziert, der im HTML-Format vorliegt. Anschließend wird die Verbindung via *res.end()* geschlossen.

*var server = http.createServer(function(req, res) {*

*res.writeHead(200, {*

*'Content-Type': 'text/html'*

*});*

*res.write('Hello World!');*

*res.end();*

*});*

Die *listen*-Funktion in Codezeile 12 dient dazu, den Webserver an die in Zeile 2 definierte Portnummer zu binden, und ermöglicht es zudem, einen Callback anzuhängen. In Zeile 13 nutzen wir diesen, um mit *console.log* eine Statusmeldung ins Terminal zu schreiben.

*server.listen(port, () => {*

*console.log('Server is available under http://127.0.0.1:' + port + '/');*

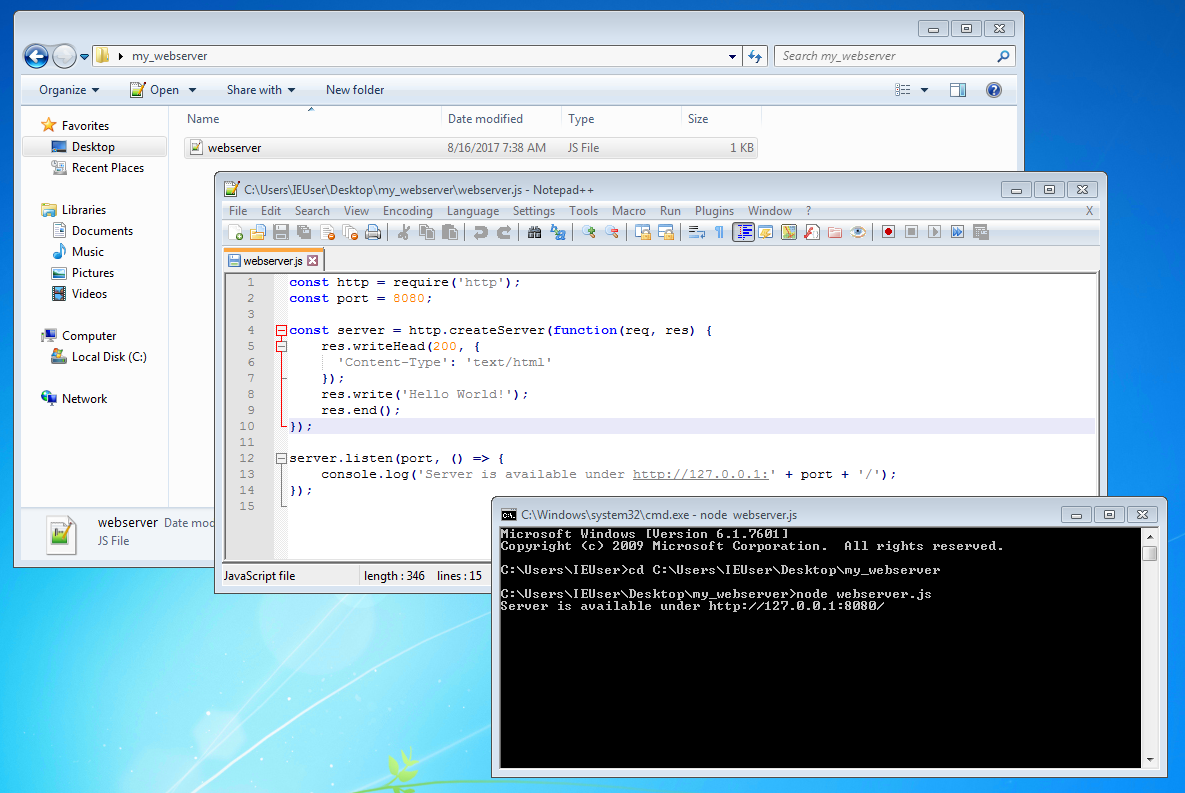
*});*

Im aktuellen Beispiel steht der Webserver unter der **Localhost-Adresse** *127.0.0.1:8080* innerhalb des eigenen Systems zur Verfügung.

1. **Anwendungscode ausführen:** Um den Webserver zu starten, führen Sie die soeben erstellte JavaScript-Datei über die Windows-Eingabeaufforderung aus. Navigieren Sie dafür in das Verzeichnis *my\_webserver* und bestätigen Sie folgenden Befehl mit der Enter-Taste:

*node webserver.js*

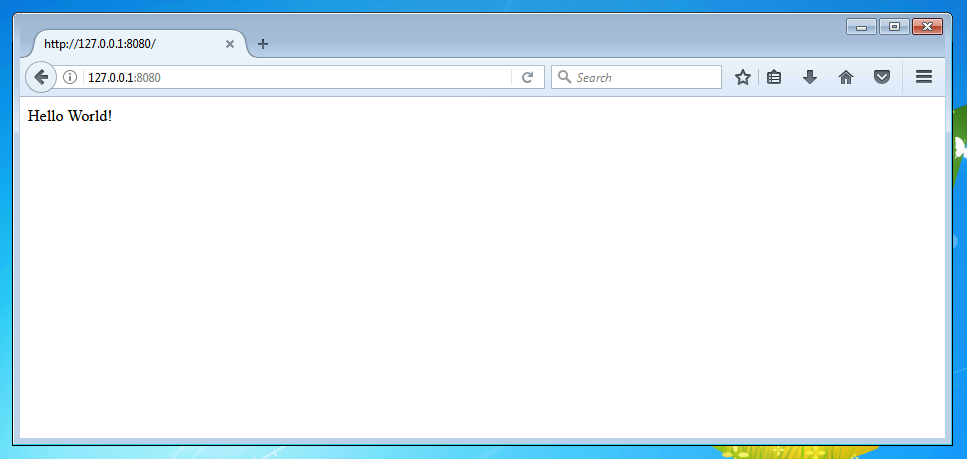
Haben Sie alle Angaben im Skript korrekt übernommen und den Dateipfad fehlerfrei angegeben, gibt die Konsole die Webadresse des Servers aus.



[Alt-Text: Terminalausgabe nach der Ausführung des Webserver-Skripts]

[BU: Node.js wird oft für die Einrichtung leichtgewichtiger Webserver eingesetzt]

1. **Webserver testen:** Der Webserver steht nun bereit, um Anfragen entgegenzunehmen. Testen können Sie dies durch die Eingabe der Webadresse *127.0.01:8080* in einen beliebigen Webbrowser. Wurde der Server korrekt programmiert, zeigt das Browserfester den Text „Hello World!“.



[Alt-Text: Node.js-Webserver testen: Die Server-Antwort im Browser-Fenster]

[BU: Gibt der Browser den Text „Hello World!“ aus, wurde der Webserver richtig programmiert]

Sie haben mit *webserver.js* Ihre erste eigene Anwendung geschrieben.Im nächsten Kapitel zeigen wir Ihnen, wie Sie eigene Anwendungen als Module in andere Anwendungen einbinden.

#### Eigene Module einbinden

Eigene Module werden ebenso wie Basismodule oder Drittanbietermodule über die *require*-Funktion eingebunden. Um dies zu verdeutlichen, **zerlegen** wir die soeben erstellte Webserver-Anwendung **in zwei einzelne Dateien**: eine Datei, die den Webserver-Code enthält, und eine Datei, die den Anwendungscode enthält. Anschließend binden wir den Anwendungscode als Modul in den Webserver-Code ein.

[Hinweis: Eine Aufteilung nach Funktionalität entspricht dem Design-Prinzip **Separation of Concerns (SoC)**. Dabei handelt es sich um ein grundlegendes Gestaltungsprinzip modular aufgebauter Computerprogramme, bei dem der Anwendungscode so strukturiert wird, dass jeder Teil, der eine neue in sich abgeschlossene Funktionalität bereitstellt, in eine eigene Datei ausgelagert wird.]

Um Ihren Webserver gemäß dem Prinzip Separation of Concerns (SoC) in zwei separate Anwendungen aufzuteilen, gehen sie wie folgt vor:

1. **Anwendungscode für das Handling von Requests und Responses schreiben:** Erstellen Sie eine JavaScript-Datei *handle.js* nach folgendem Vorbild und speichern Sie diese im Verzeichnis *my\_webserver*.

*const handle = function (req, res)* *{*

*res.writeHead(200, {*

*'Content-Type': 'text/html'*

*});*

*res.write('Hello World!');*

*res.end();*

*};*

*module.exports = handle;*

Die Datei *handle.js* enthält den Teil ihrer Webserver-Anwendung, der für das Handling von Requests und Responses zuständig ist. Sie umfasst somit im Wesentlichen die Callback-Funktion *function(req, res)*, die Sie im vorangegangenen Beispiel innerhalb der Funktion *http.createServer()* inline definiert haben. Diese Callback-Funktion wird in der Konstante *handle* gespeichert. Um die als *handle* definierte Callback-Funktion auch außerhalb von *handle.js* verwenden zu können, müssen Sie diese als Modul exportieren. Node.js stellt dazu das Objekt *module.exports* zur Verfügung, mit dem sich beliebige Elemente (z. B. Funktionen, Strings, Objekte etc.) exportieren lassen.

1. **Webserver-Code schreiben:** Erstellen Sie eine JavaScript-Datei *webserver1.js* nach folgendem Vorbild und speichern Sie diese in Ihrem Anwendungsverzeichnis *my\_webserver*.

*const http = require('http');*

*const port = 8080;*

*const handle = require('./handle');*

*const server = http.createServer(handle);*

*server.listen(port, () => {*

*console.log('Server is available under http://127.0.0.1:' + port + '/');*

*});*

Der Code entspricht im Wesentlichen dem der Webserver-Anwendung aus dem vorigen Beispiel. Doch statt das Handling von Requests und Responses inline in der*createServer*-Funktion zu definieren, verweisen Sie in Zeile 5 auf das zuvor erstellte Modul *handle* (die Dateiendung *.js* kann weggelassen werden). Damit die Ressource *handle* im Rahmen des Programmablaufs zur Verfügung steht, muss diese via *require()* als Modul eingebunden werden. Da sich beide Dateien im aktuellen Arbeitsverzeichnis befinden, verwenden wir den relativen Dateipfad *./handle*. Absolute Dateipfade werden in Node.js-Projekten in der Regel nicht verwendet.

1. **Webserver-Anwendung ausführen**: Öffnen Sie die Windows-Eingabeaufforderung, navigieren Sie in das Verzeichnis *my\_webserver* und führen Sie die JavaScript-Datei *webserver1.js* mit folgendem Befehl aus:

*node webserver1.js*

Sofern das Modul *handle* korrekt eingebunden wurde, erscheint die Statusmeldung: „Server is available under http://127.0.0.1:8080“*.* Rufen Sie diese Adresse auf, erhalten Sie: „Hello World!“.

#### Eigene Module veröffentlichen

Möchten Sie selbstentwickelte Anwendungen oder Programmteile auch anderen Node.js-Nutzern zur Verfügung stellen, empfiehlt sich eine **Veröffentlichung via npm**. Die npm-Registry dient der Community als zentrales Archiv für Drittanbietermodule. Prinzipiell kann jeder Node.js-Nutzer Module bereitstellen. Voraussetzung dafür ist eine **kostenlose Registrierung** bei npm.

[Hinweis: Zusätzlich zum kostenlosen Account bietet npm ein kostenpflichtiges Upgrade sowie eine Enterprise-Version an, die ein erweitertes Funktionsspektrum zur Verfügung stellen.]

Um sich bei npm zu registrieren, gehen Sie folgendermaßen vor:

1. Rufen Sie die npm-Startseite [Zur Startseite von npm] (https://www.npmjs.com/) auf.
2. Füllen Sie das Registrierungsformular aus und senden Sie dieses ab.
3. Bestätigen Sie Ihre E-Mail-Adresse.

Sie können sich nun über die Weboberfläche unter *www.npmjs.com* anmelden. Alternativ stellt npm den folgenden Befehl zur **Anmeldung über der Terminal** zur Verfügung:

*npm login*

Bei jedem Log-in über das Terminal werden Ihr Benutzername, Ihr Passwort sowie ihre E-Mail-Adresse abgefragt. Nachdem Sie sich über das Terminal angemeldet haben, können Sie beliebig viele Node.js-Module veröffentlichen.

**Voraussetzung für die Veröffentlichung** eines Moduls ist, dass dieses eine vollständige *package.json* enthält, in der sämtliche Metadaten zum Modul erfasst wurden. Zu den Standardangaben in dieser Metadaten-Datei gehören u. a. folgende Informationen:

* **name:** Der Name Ihres Moduls (Pflicht)
* **version:** Die Versionsnummer Ihres Moduls (Pflicht)
* **description:** Eine kurze Beschreibung Ihres Moduls
* **entry point:** Die Einstiegsdatei Ihrer Anwendung (z. B. index.js)
* **test command:** Ein Kommando, mit dem sich Tests starten lassen (falls vorhanden)
* **git repository:** URL zu einem Git-Repository, z. B. GitHub(falls vorhanden)
* **keywords:** Schlüsselwörter, die der Auffindbarkeit über die npm-Suchfunktion dienen
* **author:** Ihr Name
* **license:** Die Lizenz, unter der das Projekt veröffentlicht wird (z. B. MIT)

Node.js-Anwender, die die *package.json* nicht von Hand schreiben möchten, können auf folgenden Kommandozeilenbefehl zurückgreifen:

*npm init*

Dieser startet ein Kommandozeilenprogramm, das grundlegende Metadaten im interaktiven Modus abfragt und anschließend eine JSON-Datei nach folgendem Schema erstellt:

*{*

*"name": "module-name",*

*"version": "1.0.0",*

*"description": "An example module to illustrate the usage of a package.json",*

*"main": "index.js",*

*"scripts": {*

*"test": "echo \"Error: no test specified\" && exit 1"*

*},*

*"repository": {*

*"type": "git",*

*"url": "https://github.com/username/repo.git"*

*},*

*"keywords": [*

*"example"*

*],*

*"author": "John Doe",*

*"license": "MIT",*

*"bugs": {*

*"url": "https://github.com/username/repo/issues"*

*},*

*"homepage": "https://github.com/username/repo#readme"*

*}*

Beachten Sie: Die via *npm init* erstellte *package.json* muss in der Regel **manuell ergänzt** werden – beispielsweise um eine Liste der Drittanbietermodule, die für den Programmablauf benötigt werden. Diese werden unter *dependencies* (Abhängigkeiten) erfasst.

*"dependencies": {*

*"express": "4.2.x",*

*"… "*

*}*

Im aktuellen Beispiel wird eine Abhängigkeit zum Drittanbietermodul *express* in der Version *4.2.x* ausgewiesen.

Erst die in der *package.json* hinterlegten Metadaten ermöglichen es, ein Modul via Name und Versionsnummer eindeutig zu identifizieren. Darüber hinaus stellt die *package.json* anderen Node.js-Anwendern alle Informationen bereit, die diese benötigen, um das Modul zu benutzen und die erforderlichen Abhängigkeiten bereitzustellen. Denn jedes Modul wird prinzipiell ohne *node\_modules*-Ordner in die Registry geladen. Kommt ein Drittanbietermodul zum Einsatz, nutzt Node.js stattdessen die Angaben unter *dependencies*, um erforderliche Module automatisch nachzuladen. Diese Metainformation sollten Sie somit immer angeben, wenn Sie eigene Drittanbietermodule in Ihre Anwendung einbinden.

Sofern Sie alle relevanten Metadaten in der *package.json* hinterlegt haben, steht der Veröffentlichung Ihrer Anwendung nichts mehr im Weg. Der **Upload in die npm-Registry** erfolgt in drei einfachen Schritten:

1. **Log-in:** Melden Sie sich über das Terminal bei npm an.

*npm login*

1. **Anwendung auswählen:** Navigieren Sie in das Verzeichnis der Anwendung, die Sie als Modul veröffentlichen möchten.
2. **Modul veröffentlichen:** Geben Sie folgenden Befehl ins Terminal ein und bestätigen Sie mit Enter:

*npm publish*

Der Befehl *npm publish* packt das ausgewählte Modul (Anwendungscode und *package.json*) als tar-Archiv und lädt dieses in die npm-Registry hoch.

**Beachten Sie:** Module, die einmal veröffentlicht wurden, lassen sich nur innerhalb der ersten 24 Stunden nach Veröffentlichung entfernen. Nutzen Sie dafür folgenden Befehl:

*npm unpublish*

Module, die länger als 24 Stunden online sind, lassen sich nutzerseitig nicht mehr löschen. Kontaktieren Sie in diesem Fall den npm-Support. Diese Unpublish-Policy soll verhindern, dass Node.js-Nutzer Anwendungen mit Abhängigkeiten zu Modulen entwickeln, die später nicht mehr zur Verfügung stehen.

## Asynchrone Programmierung mit Node.js: Callbacks, Events und Streams

Großer Vorteil von Node.js ist die eventgesteuerte Architektur, mit der sich Programmcode **asynchron** ausführen lässt. Dabei setzt Node.js auf **Single-Threading** und ein ausgelagertes Input/Output-System (I/O), das eine parallele Bearbeitung mehrerer Schreib- und Leseoperationen erlaubt.

* **Asynchrones I/O:** Zu den klassischen Aufgaben eines Servers gehören das Beantworten von Anfragen, das Speichern von Daten in eine Datenbank, das Lesen von Dateien von der Festplatte und der Aufbau von Verbindungen zu anderen Netzwerk-Komponenten. Diese Aktivitäten werden unter dem Kürzel „I/O“ (Input/Output) zusammengefasst. In Programmiersprachen wie C oder Java werden I/O-Operationen synchron ausgeführt. Es wird somit eine Aufgabe nach der anderen abgearbeitet. Dabei ist das I/O-System so lange blockiert, bis die aktuelle Aufgabe erledigt wurde. Node.js hingegen nutzt ein asynchrones I/O, bei dem Schreib- und Leseoperationen direkt an das Betriebssystem oder eine Datenbank delegiert werden. Dies ermöglicht es, eine große Anzahl an I/O-Aufgaben parallel durchzuführen, ohne dass es zum Blocking (einer Blockade) kommt, was Anwendungen auf Basis von Node.js und JavaScript in manchen Szenarien einen enormen Geschwindigkeitsvorteil verschafft.
* **Single-Threading:** Um Wartezeiten bei synchronem I/O zu kompensieren, setzen Serveranwendungen auf Basis klassischer serverseitiger Programmiersprachen auf zusätzliche Threads – mit den oben erwähnten Nachteilen eines Multithreading-Ansatzes. So startet beispielsweise Apache HTTP Server für jede eingehende Anfrage einen neuen Thread. Die Anzahl der möglichen Threads wird durch den verfügbaren Arbeitsspeicher begrenzt – und somit auch die Anzahl der Anfragen, die in einem synchronen Multithreading-System parallel beantwortet werden können. Node.js hingegen kommt aufgrund des ausgelagerten I/O-Systems mit nur einem Thread aus, wodurch sowohl die Komplexität als auch die Ressourcenauslastung deutlich reduziert werden.

Realisiert wird die asynchrone Abarbeitung von I/O-Operationen bei Node.js durch Konzepte wie **Callbacks**, **Events** und **Streams**.

[Hinweis: Beachten Sie: Die Kernanwendung Node.js startet durchaus mehrere Threads, lediglich sogenannter „Userland-Code“ – der Programmcode des jeweiligen Entwicklers – läuft auf Single-Threading-Basis.]

### Callbacks

Bei einem Callback handelt es sich um eine **Rückruf-Funktion**, die im Rahmen von Node.js immer dann zum Einsatz kommt, wenn eine Aufgabe ausgelagert und somit JavaScript-Code asynchron ausgeführt werden soll (z. B. im Fall einer I/O-Operation). Callback-Funktionen haben die Aufgabe, sich zu melden, sobald eine delegierte Operation abgeschlossen wurde, und das Ergebnis zurückzugeben. In der Regel werden Callbacks so implementiert, dass eine Fehlermeldung ausgegeben wird, sofern die delegierte Aufgabe nicht wie gewünscht ausgeführt werden konnte.

Dabei ist zu beachten, dass Callback-Funktionen **nicht zwangsläufig asynchron** sind. Zwar kommen Callbacks immer dann zum Einsatz, wenn Programmcode asynchron ausgeführt werden soll. Dass Callbacks immer auf asynchronen Code hindeuten, wäre im Umkehrschluss jedoch falsch.

Der Unterschied zwischen einer synchronen und einer asynchronen Ausführung von Programmcode lässt sich an Beispielen verdeutlichen. Gehen Sie folgendermaßen vor, um eine Datei synchron einzulesen:

1. **Textdatei erstellen:** Legen Sie eine Textdatei *input\_sync.txt* mit beliebigem Inhalt an – beispielsweise:

*This code example runs synchronously.*

Speichern Sie diese in einem beliebigen Verzeichnis auf Ihrem System.

1. **Anwendung schreiben**: Schreiben Sie eine einfache JavaScript-Anwendung nach folgendem Beispiel und legen Sie diese unter *app\_sync.js* im selben Verzeichnis ab.

*var fs = require('fs');*

*var data = fs.readFileSync('input\_sync.txt');*

*console.log(data.toString());*

*console.log('Program Ended');*

1. **Anwendung ausführen:** Öffnen Sie die Windows-Eingabeaufforderung, navigieren Sie in das Verzeichnis, in dem Sie die Anwendung und die Textdatei gespeichert haben, und führen Sie *app\_sync.js* mit folgendem Befehl aus:

*node app\_sync.js*

In der Konsole erhalten Sie diese Ausgabe:

*This code example runs synchronously.*

*Program Ended*

Der Programm-Code von *app\_sync.js* wird sequenziell, d. h. von oben nach unten, abgearbeitet. Zunächst werden die Daten aus der *input\_sync.txt* via *fs.readFileSync()* ins Programm geladen und anschließend mithilfe der Funktion *console.log()* auf die Standardausgabe geschrieben (in der Regel das Terminal). Erst wenn diese I/O-Operation abgeschlossen ist, fährt das Programm mit der zweiten *console.log*-Funktion fort, die den String „Program Ended“ ins Terminal schreibt.

Gehen Sie so vor, um eine **Callback-Funktion asynchron** auszuführen:

1. **Textdatei erstellen:** Legen Sie eine zweite Textdatei *input2.txt* mit beliebigem Inhalt an – beispielsweise:

*This code example runs asynchronously.*

Speichern Sie diese Textdatei im selben Verzeichnis wie die anderen Dateien.

1. **Anwendung schreiben**: Schreiben Sie eine zweite JavaScript-Anwendung nach folgendem Beispiel und legen Sie diese unter *app\_async.js* im selben Verzeichnis ab.

*var fs = require('fs');*

*fs.readFile('input\_async.txt', function (err, data) {*

*if (err) return console.error(err);*

*console.log(data.toString());*

*});*

*console.log('Program Ended');*

1. **Anwendung ausführen:** Führen Sie *app\_async.js* mit folgendem Befehl aus:

*node app\_async.js*

In der Konsole erhalten Sie folgende Ausgabe:

*Program Ended*

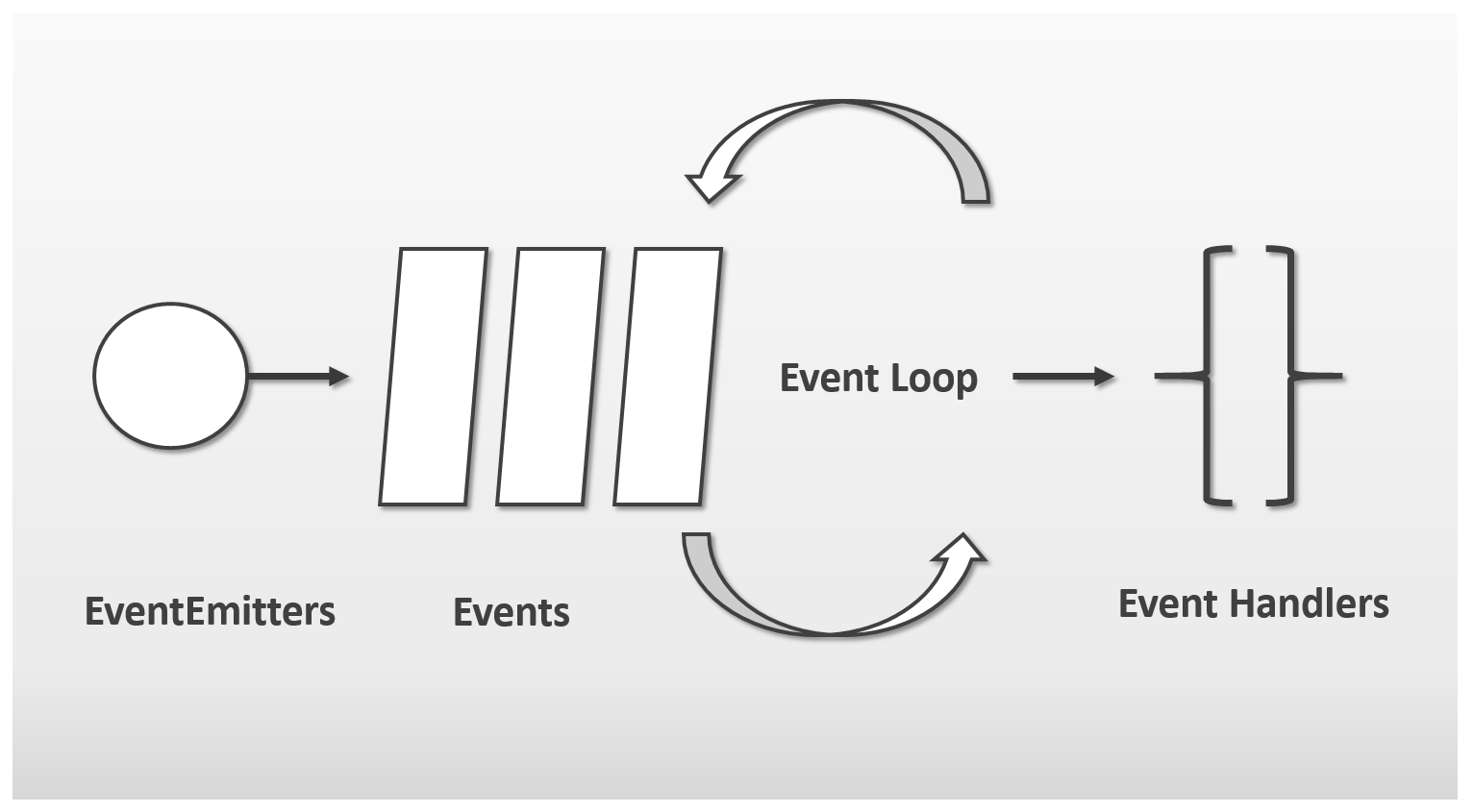
*This code example runs asynchronously.*

Auch der Programm-Code *app\_async.js* wird sequenziell von oben nach unten ausgeführt. Anders als bei *app\_sync.js* wartete das Programm jedoch nicht, bis die I/O-Operation in Zeile 2abgeschlossen ist, sondern fährt, nachdem die Aufgabe an das Dateisystem delegiert wurde, ohne Verzögerung mit der Ausführung des weiteren Programm-Codes fort. Da die I/O-Operation länger dauert als die Ausführung der *console.log*-Funktion in Zeile 6, erhalten wir zunächst den String „Program Ended“und erst danach das Ergebnis der I/O-Operation.

### Events

Während ein klassischer Callback im Rahmen des Programmablaufs lediglich einmal aufgerufen wird und somit auch nur einmal ein Ergebnis zurückgibt, bietet Node.js mit dem integrierten *events*-Modul die Möglichkeit, Auslöser und Antwort durch Events zu entkoppeln. Ein und derselbe Event-Handler kann somit problemlos auf unterschiedliche Event-Auslöser antworten. Zum Einsatz kommt dieses Konzept immer dann, wenn beim Eintreffen eines bestimmten Ereignisses automatisch eine damit verbundene Operation ausgeführt werden soll.

Die **eventgesteuerte Architektur von Node.js** basiert im Wesentlichen auf einem einzigen Thread, der sich in einer endlos laufenden Ereignisschleife befindet. Dieser **Event-Loop** hat die Aufgaben, auf Ereignisse zu warten und diese zu verwalten. Dabei können Ereignisse entweder als Aufgaben oder als Ergebnisse vorliegen. Registriert der Event-Loop eine Aufgabe, beispielsweise eine Datenbankabfrage, wird diese über eine Callback-Funktion an einen Prozess im Hintergrund ausgelagert. Die Bearbeitung der Aufgabe erfolgt somit nicht im selben Thread, in dem der Event-Loop läuft, sodass dieser umgehend zum nächsten Ereignis übergehen kann. Wurde eine ausgelagerte Aufgabe ausgeführt, werden die Ergebnisse des ausgelagerten Prozesses über die Callback-Funktion als neues Ereignis an den Event-Loop zurückgegeben. Infolgedessen kann dieser die Auslieferung des Ergebnisses anstoßen.



[Alt-Text: Schematische Darstellung der eventgesteuerten Architektur von Node.js]

[Bildunterschrift: Die eventgesteuerte Architektur von Node.js]

Um **Objekte** zu generieren, die Events auslösen und verarbeiten können, stellt Node.js im Basismodul *events* die Klasse ***EventEmitter*** zur Verfügung. Diese ermöglicht es Programmierern, eigene Klassen für eventauslösende Objekte abzuleiten sowie Objekte zu erzeugen, die automatisch die eventauslösende Funktion *emit()* erben.

In folgendem Beispiel binden wir das *events*-Modul über die Funktion *require()* ein und speichern dieses als *EventEmitter*-Klasse in einer Konstante ab. Anschließend nutzen wir *extends*, um aus der Klasse *EventEmitter* eine eigene eventauslösende Klasse *myEmitter* abzuleiten. Das eigentliche Event-Objekt erzeugen wir mithilfe des Operators *new.*

*const EventEmitter = require('events');*

*class MyEmitter extends EventEmitter {}*

*const myEmitter = new MyEmitter();*

*myEmitter.on('myEvent', function(a, b) {*

*console.log(a + b + a);*

*});*

*myEmitter.emit('myEvent', 1, 'und');*

Um Objekte mit der Fähigkeit auszustatten, **auf Events zu** **reagieren**, stellt die *EventEmitter*-Klasse die **Funktion *on()*** zur Verfügung. Diese nimmt zwei Parameter entgegen: Der erste Parameter enthält den Namen des Events, auf das gelauscht werden soll, und registriert das Objekt somit für das entsprechende Event. Der zweite Parameter besagt, wie auf das Event reagiert werden soll. Dazu wird Objekten mit dem zweiten Parameter ein sogenannter Event-Handler „angehängt“, eine Callback-Funktion, die immer dann aufgerufen wird, wenn das gewünschte Event eintritt.

Soll das Event-Objekt Events auslösen, kommt die Funktion *emit()* zum Einsatz. Die *emit-*Funktion erwartet einen Parameter, der den Namen des Events beinhaltet. Weitere optionale Parameter können Daten zum Event enthalten, die an die Event-Handler-Funktion weitergegeben werden.

### Streams

Bei einem Stream handelt es sich um die Abstraktion einer Datensequenz, die es Node.js erlaubt, **große Datenmengen** effizient zu verarbeiten. Konzeptionell lassen sich Streams mit Arrays vergleichen. Doch während Arrays alle zu verarbeitenden Daten zeitgleich als Werte im Arbeitsspeicher vorhalten, werden diese bei Streams Stück für Stück verarbeitet, sodass immer nur ein Teil der Daten in den Arbeitsspeicher geladen werden muss.

[Hinweis: **Arrays** sind Datentypen, in denen – anders als bei einfachen Objekten – mehrere Werte separat und in feststehender Reihenfolge gespeichert werden können. Technisch werden Arrays als zusammenhängende, linear adressierbare Bereiche von Speicherzellen im Arbeitsspeicher realisiert.]

Statt wie bei Arrays in räumlicher Abfolge werden Daten in Streams somit in zeitlicher Abfolge vorgehalten. Diese Art der Datenverarbeitung ermöglicht es Node.js, mit Datenmengen umzugehen, deren Größe den zur Verfügung stehenden Arbeitsspeicher deutlich überschreitet. Ein weiterer Vorteil ist die **Reduktion der Latenzzeit**. Um auf Daten aus einem Array zugreifen zu können, muss ein Programm warten, bis dieses komplett geladen ist. Streams hingegen stellen Daten kontinuierlich bereit, sobald die ersten Bytes eingetroffen sind, und laden den Rest im Hintergrund nach.

Das Konzept des Streams geht auf die Interprozesskommunikation in der UNIX-Shell zurück. Unter UNIX sind Streams über die Pipe (|) in die Shell implementiert. Diese ermöglicht es, Daten von einem Prozess an einen anderen zu übergeben. Auch Node.js bietet mit dem Basismodul *stream* eine API für die Datenverarbeitung via Stream. Anwendern stehen **zwei Basistypen von Streams** zur Verfügung: Readable Streams und Writable Streams. Auf diesen aufbauend, lassen sich darüber hinaus sogenannte Duplex- und Transform-Streams realisieren, die eine Kombination beider Basis-Streams darstellen.

|  |  |  |  |
| --- | --- | --- | --- |
| **Basis-Streams** | | **Kombinierte Streams** | |
| **Readable Streams** | **Writable Streams** | **Duplex-Streams** | **Transform-Streams** |
| Lesbare Streams, die zum Einsatz kommen, um Daten aus einer Datenquelle auszulesen | Beschreibbare Streams, die es ermöglichen, Daten in eine bestimmte Datenquelle zu schreiben | Kombination aus einem Readable Stream und einem Writable Stream  Beide Streams sind unabhängig voneinander | Kombination aus einem Readable Stream und einem Writable Stream  Beide Streams sind abhängig voneinander  Transform-Streams ermöglichen eine Modifikation und Transformation von Daten |

Für alle Stream-Typen stellt Node.js über das *stream*-Modul jeweils eine Basisklasse zur Verfügung, die Eigenschaften von der *EventEmitter*-Klasse erbt. Jeder Stream ist somit auch ein *EventEmitter*. Im Code werden Streams als Objekte realisiert, die in der Lage sind, Events auszulösen. Zu den wichtigsten Events für den Umgang mit Streams zählen *data-*, *end-*, *error-* und *finish-*Events.

|  |  |
| --- | --- |
| **Event** | **Beschreibung** |
| *data* | *data*-Events werden immer dann ausgelöst, wenn ein Readable Stream einen Datenblock bereitstellt. In der Regel wird der Datenfluss durch die Funktion *.pipe()* angestoßen. |
| *end* | *end*-Events werden von Readable Streams immer dann ausgelöst, wenn der letzte Datenblock eingetroffen ist und keine weiteren Daten mehr übrig sind, die vom Stream bereitgestellt werden können. |
| *error* | *error*-Events werden von Readable Stream und Writable Streams ausgelöst, wenn beim Lesen bzw. Schreiben der Daten ein Fehler aufgetreten ist. |
| *finish* | *finish*-Events werden von Writable Streams ausgelöst, sobald alle Daten vom zugrundeliegenden System verarbeitet wurden. |

Im Folgenden veranschaulichen wir den Umgang mit Streams am Beispiel der beiden Basis-Stream-Typen.

#### Readable Streams

Readable Streams kommen im Rahmen eines Programms zum Einsatz, um **Daten aus einer Quelle auszulesen**. Mögliche Datenquellen für diesen Stream-Typ sind das Dateisystem, Netzwerk-Endpunkte oder In-Memory-Datenstrukturen.

Wie alle Streams sind auch Readable Streams im JavaScript-Code als Objekte repräsentiert, die zunächst erzeugt werden müssen. Eine Funktion, umReadable Streamszu erzeugen, wird beispielsweise durch das *fs*-Modul bereitgestellt, das Node.js eine Interaktion mit dem Dateisystem ermöglicht. Im Beispiel zeigen wir Ihnen in drei Schritten, wie Sie Readable Streams nutzen, um Daten aus einer Textdatei auszulesen und in Form eines Strings ins Terminal zu schreiben.

1. **Inputdatei erstellen:** Erstellen Sie eine Textdatei *input.txt* mit beliebigem Inhalt und legen Sie diese in einem Verzeichnis ihrer Wahl ab.
2. **Anwendungscode schreiben:** Erstellen Sie eine JavaScript-Anwendung nach folgendem Vorbild und speichern Sie diese unter *read-stream.js* im selben Verzeichnis wie die *input.txt*.

*const fs = require('fs');*

*const MyReadableStream = fs.createReadStream('input.txt', {encoding: 'utf8' });*

*MyReadableStream.on('data' , data => {*

*console.log(data);*

*});*

*MyReadableStream.on('error', err => {*

*console.log(err.stack);*

*});*

*MyReadableStream.on('end', () => {*

*console.log('--- The End ---');*

*});*

Zunächst nutzen Sie die Funktion *require()*, um das *fs*-Modul zu laden. Dieses hält die Funktion *fs.createReadStream()* bereit, mit der Sie Streams für Lesezugriffe auf das Dateisystem erzeugen. Die Funktion *fs.createReadStream()* erwartet als ersten Parameter den Pfad zur Datei, die Sie auslesen möchten. Ein zweiter Parameter mit Optionen ist fakultativ. In der Regel kommt ein Encoding zum Einsatz.

Für das aktuelle Beispiel erzeugen wir einen Stream mit dem Namen *MyReadableStream* auf die Datei *input.txt* und nutzen dabei ein utf8-Encoding. Die Funktion *fs.createReadStream()* hat somit zwei Aufgaben: Zum einen sorgt sie im Rahmen des Programmablaufs dafür, dass die Daten blockweise aus der Datei in den Stream gelesen werden. Und zum anderen gibt sie vor, in welcher Kodierung die ausgelesenen Daten im Stream vorgehalten werden sollen. Wird das Encoding weggelassen, lädt *fs.createReadStream()* reine Binärdaten in den Stream.

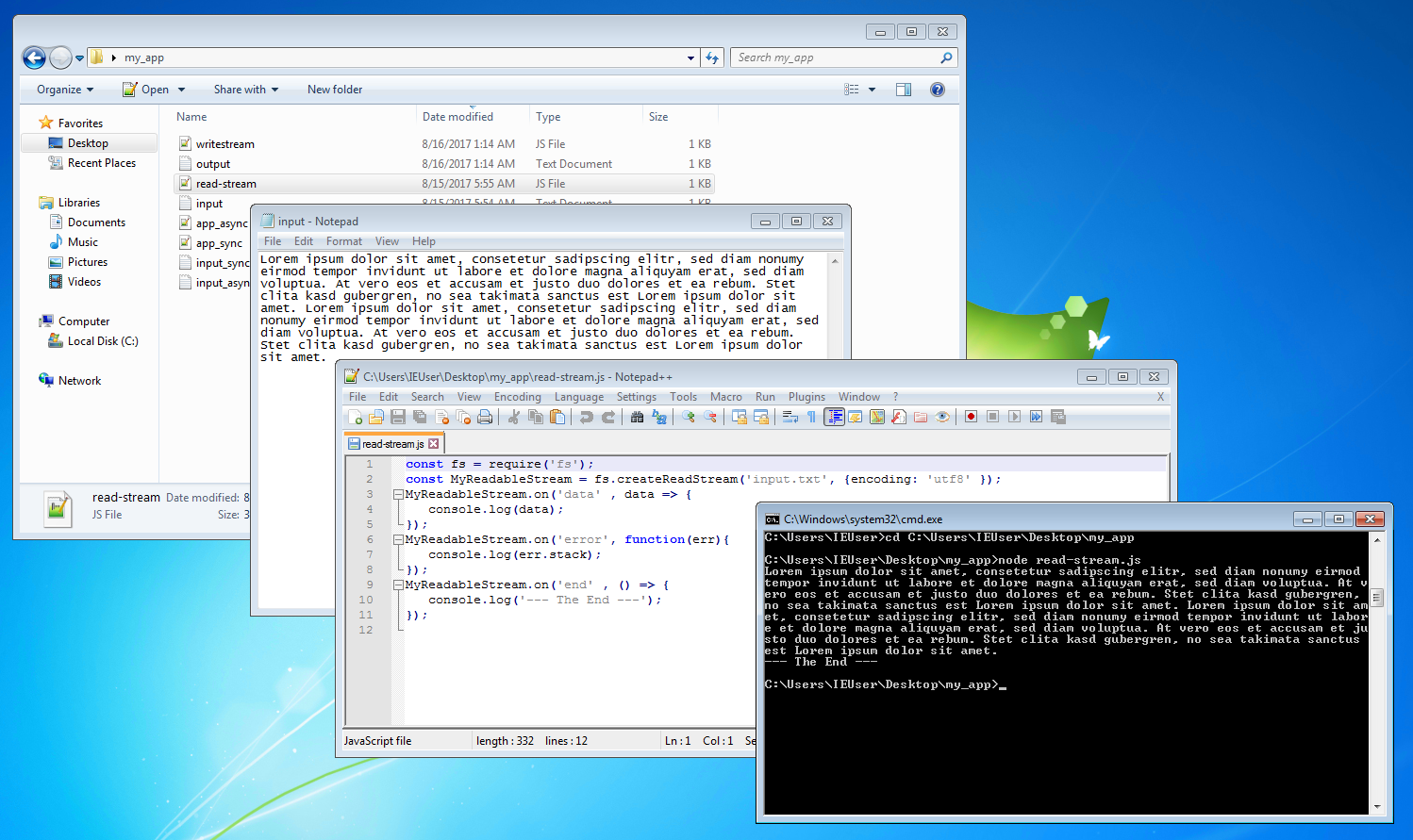
Das Streaming der Daten erfolgt Block für Block. Jedes Mal, wenn die ausgelesenen Daten eine von Node.js vordefinierte Blockgröße erreicht haben, wird an dem Stream *MyReadableStream* ein *data*-Event ausgelöst. Auf dieses Event reagieren Sie, indem Sie mithilfe der *on*-Funktion einen Callback anhängen. Im aktuellen Beispiel nutzen wir die Funktion *console.log()*. Die ausgelesenen Datenblöcke werden auf das Terminal ausgegeben.

Zusätzlich zu *data*-Events können Readable Streams *error*- und *end*-Events auslösen, auf die Sie ebenfalls reagieren sollten. Das *error*-Event wird bei Readable Streams immer dann ausgelöst, wenn Lesefehler auftreten. Es empfiehlt sich, eine *console.log*-Funktion via *on()* anzuhängen, die die Fehlermeldung im Terminal ausgibt. Zum *end*-Event kommt es, sobald die Datenquelle vollständig ausgelesen wurde und der Readable Stream keine weiteren Datenblöcke bereitstellen kann. Reagieren Sie auf das *end*-Event ebenfalls mit einer *console.log()*-Funktion, um sich eine Statusmeldung auf das Terminal ausgeben zu lassen.

1. **Anwendungscode ausführen:** Öffnen Sie die Windows-Eingabebestätigung, navigieren Sie in das gewählte Verzeichnis und starten Sie *read-stream.js* mit folgendem Kommando:

*node read-stream.js*

Das Programm *read-stream.js* sollte Ihnen nun den Inhalt der von Ihnen erstellten Datei *input.txt* in Textform auf das Terminal ausgeben – gefolgt von der Statusmeldung, dass der Stream beendet wurde (*--- The End ---*).



[Alt-Text: Terminalausgabe nach der Ausführung der JavaScript-Anwendung *read-stream.js*]

[Bildunterschrift: Die Anwendung *read-stream.js* erzeugt einen Readable Stream auf die Textdatei input.txt]

#### Writable Streams

Writable Streams **nehmen Daten blockweise entgegen** und ermöglichen es Ihnen, diese in einen beliebigen Zielspeicherort – beispielsweise eine Datei oder einen anderen Stream – zu schreiben. Wie Readable Streams sind auch Writable Streams Ableitungen der *EventEmitter*-Klasse und somit in der Lage, Events auszulösen. Diese spielen beim Umgang mit Writable Streams jedoch eine untergeordnete Rolle.

Zentrale Elemente dieses Stream-Typs sind stattdessen die Funktionen *write()* und *end()*, die in jedem Fall implementiert sein müssen. Die Funktionsweise des Writable Streams veranschaulichen wir am Beispiel einer einfachen Dateioperation, bei der wir Textelemente aus dem Programm-Code einer JavaScript-Anwendung mithilfe eines Streams in eine Textdatei schreiben.

1. **Anwendungscode schreiben:** Erstellen Sie eine JavaScript-Anwendung nach folgendem Vorbild und speichern Sie diese unter *write-stream.js* in einem beliebigen Verzeichnis ab.

*const fs = require('fs');*

*const MyWritableStream = fs.createWriteStream('output.txt', { defaultEncoding: 'utf8' });*

*MyWritableStream.write('Hello ');*

*MyWritableStream.write('World!\n');*

*MyWritableStream.end();*

*MyWritableStream.on('finish', function() {*

*console.log("Write completed.");*

*});*

*MyWritableStream.on('error', function(err) {*

*console.log(err.stack);*

*});*

Wie Readable Streams lassen sich auch Writable Streams mithilfe des moduls *fs* erzeugen, das sie über die Funktion *require()* in Ihre Anwendung laden. Nutzen Sie dazu die Funktion *fs.createWriteStream()*. Auch diese Funktion erwartet als ersten Parameter einen Pfad bzw. einen Dateinamen, der angibt, wo die Streaming-Daten gespeichert werden sollen.

Wenn die Datei *output.txt* noch nicht existiert, wird diese im Rahmen der Programmausführung automatisch erstellt. Andernfalls überschreibt der Writable Stream eine bereits existierende Datei mit gleichem Namen. Setzen Sie auch bei Writable Streams den optionalen Parameter für das Encoding. Anders als beim Readable Stream kommt dazu das Keyword *defaultEncoding* zum Einsatz.

Sobald der Writable Stream auf die Datei *output.txt* erzeugt wurde, können Sie die Funktion *write()* nutzen, um Daten blockweise in den Stream zu schreiben.Solange derWritable Streamgeöffnet ist, kann die *write-*Funktionbeliebig oft aufgerufen werden. Dies ermöglicht Ihnen, Daten nach und nach in den Stream zu schreiben. Möchten Sie keine weiteren Datenin den Stream schreiben, schließen Sie diesen mit der *end-*Funktion ab.

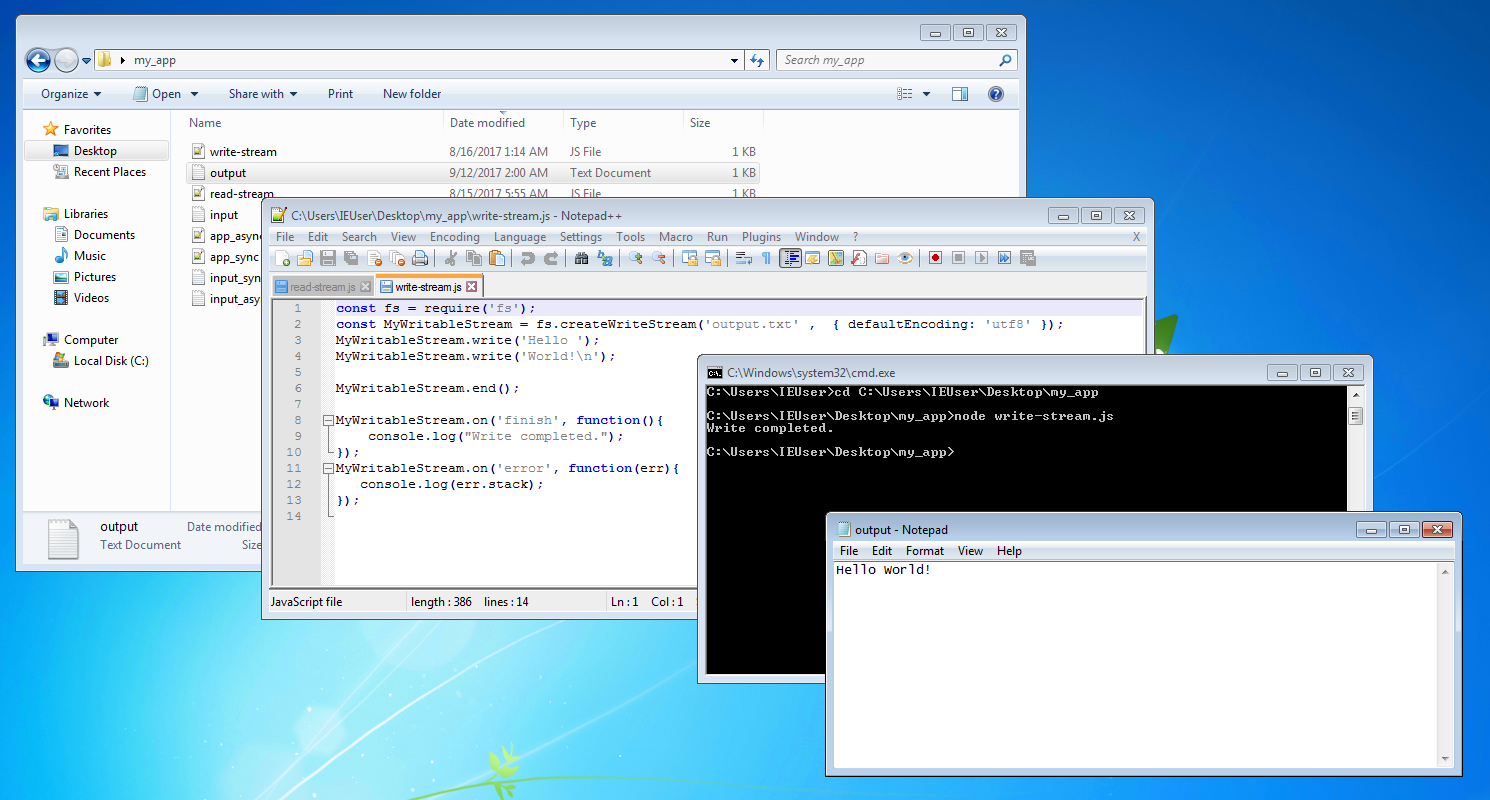
Auch Writable Streams lösen Ereignisse aus. In der Beispielanwendung reagieren wir auf das *finish*- und das *error*-Event. Das *finish*-Event wird immer dann ausgelöst, wenn alle Streaming-Daten erfolgreich in das Speicherziel geschrieben wurden. Möchten Sie sich in diesem Fall eine Statusmeldung (z. B. *Write completed*) ausgeben lassen, hängen Sie dem *finish*-Event via *on()* eine entsprechende *console.log()*-Funktion an.

Wie Readable Streams lösen auch Writable Streams *error*-Events aus, sofern beim Schreibprozess ein Fehler auftritt. Hängen Sie diesem Event ebenfalls eine *console.log*-Funktion an, um die Fehlermeldung als Terminalausgabe zu erhalten.

1. **Anwendungscode ausführen:** Öffnen Sie die Windows-Eingabeaufforderung, navigieren Sie in das gewählte Verzeichnis und starten Sie *write-stream.js* mit folgendem Kommando:

*node write-stream.js*

Sofern das Programm ohne Fehler durchgelaufen ist, sollten Sie die Ausgabe „Write completed“erhalten.In Ihrem Dateisystem wurde die Textdatei *output.txt* mit dem Inhalt „Hello World!“ erstellt.



[Alt-Text: Terminalausgabe nach Ausführung der JavaScript-Anwendung *write-stream.js*]

[Bildunterschrift: Die Anwendung *write-stream.js* erzeugt einen Writable Stream, der es ermöglicht, Daten via *write()* in die Datei *output.txt* zu schreiben.]

[Fazit: Die eventgesteuerte, nichtblockierende Architektur von Node.js setzt auf Konzepte wie Callbacks, Events und Streams. Diese bieten den Vorteil, dass Anwendungen nie untätig auf Ergebnisse warten müssen. So ist es beispielsweise möglich, verschiedene Datenbankabfragen gleichzeitig durchzuführen, ohne dass der Programmablauf aufgehalten wird. Auch ein Webseitenaufbau, der diverse externe Abfragen erfordert, lässt sich mit Node.js somit deutlich schneller bewerkstelligen als mit einem synchronen Bearbeitungsprozess.]

## Datenbankoperationen mit MySQL und MongoDB

Eine zentrale Funktionalität serverseitiger Anwendungen ist die **Interaktion mit Datenbanken**. Auch für Node.js stehen Drittanbietermodule bereit, mit denen sich Datenbanken ansprechen und Datenbankoperationen ausführen lassen.

Welche Datenbank-Management-Systeme dabei zum Einsatz kommen, ist Geschmackssache. Ein Großteil der Node.js-Anwender setzt jedoch auf **MongoDB** – und das aus gutem Grund. Bei MongoDB handelt es sich um eine dokumentenorientierte, nichtrelationale NoSQL-Datenbank. Die plattformübergreifende Software ist in C++ geschrieben und speichert Daten schemalos als BSON-Objekte, die in der Terminologie von MongoDB als **Dokumente** bezeichnet werden. Das Format BSON (Binary JSON) ist dem JSON-Format nachempfunden und stattet MongoDB mit einer nativen **JavaScript-Kompatibilität** aus. Für Anwender bedeutet das: Die Interaktion mit der Datenbank kann ebenso wie die Programmierung von Node.js-Anwendungen in JavaScript erfolgen. Eine zusätzliche Datenbanksprache wie SQL (Structured Query Language) wird nicht benötigt.

[Hinweis: JavaScript wird oft als Full-Stack-Sprache bezeichnet. Grund dafür ist, dass die Skriptsprache auf allen Ebenen der Webentwicklung zum Einsatz kommen kann: im Frontend (browserseitig), im Backend (auf dem Webserver) sowie im Zusammenhang mit JavaScript-kompatiblen Infrastrukturdiensten (beispielsweise Datenbanken).]

MongoDB wird der Gruppe der NoSQL-Datenbanken zugeordnet. Das Kürzel „NoSQL“ steht – anders als zu erwarten wäre – für „Not only SQL“ und kennzeichnet Datenbanken, die einem nichtrelationalen Ansatz folgen. Anders als relationale Datenbanken wie MySQL oder MariaDB nutzen diese **kein festgelegtes Tabellenschema**. Das bedeutet: Für Tabellen innerhalb von nichtrelationalen Datenbanken muss nicht festgelegt werden, welche Spalten es gibt oder welche Datentypen in diesen Spalten gespeichert werden können. Stattdessen werden die Daten ohne strukturelle Zuordnung in Tabellen abgelegt. Eine solche Datenbankarchitektur eignet sich vor allem für Anwendungen, bei denen **große Datenmengen** in kurzer Zeit verarbeitet werden müssen.

Der Einsatz von MongoDB bietet somit zahlreiche Vorteile. Mit Node.js sind Sie jedoch nicht auf dieses Datenbank-Management-System festgelegt. Sollte Ihr Webprojekt statt einer NoSQL-Datenbank eine SQL-basierte relationale Datenbank benötigen, lässt sich auch diese ohne weiteres als Modul einbinden.

Im Folgenden veranschaulichen wir den Umgang mit Datenbanken am Beispiel von MongoDB und der meistverwendeten relationalen Datenbank **MySQL**. Eine Gegenüberstellung beider Datenbank-Management-Systeme finden Sie im Grundlagenartikel zu MongoDB [MongoDB: Vorstellung und Vergleich mit MySQL] (https://hosting.1und1.de/digitalguide/websites/web-entwicklung/mongodb-vorstellung-und-vergleich-mit-mysql/).

### Datenbankoperationen mit MongoDB

Um mit einer Node.js-Anwendung auf MongoDB zugreifen zu können, benötigen Sie einen **Datenbanktreiber**. Das offizielle MongoDB-Modul für Node.js [Zur Modulbeschreibung auf www.npmjs.com] (https://www.npmjs.com/package/mongodb) steht per npm zur Verfügung und lässt sich wie oben beschrieben installieren:

*npm install mongodb*

Im Folgenden zeigen wir Ihnen, wie Sie MongoDB über eine Node.js-Anwendung ansprechen und im Rahmen des Programmablaufs Datenbankoperationen durchführen.

[Hinweis: Um das Codebeispiel auszuprobieren, müssen Sie eine Datenbank bereitstellen: Wie Sie dabei vorgehen, erfahren Sie in unserem MongoDB-Tutorial [MongoDB-Tutorial: Installation und Einstieg] (https://hosting.1und1.de/digitalguide/websites/web-entwicklung/mongodb-tutorial-installation-und-einstieg/). Eine komfortable Möglichkeit, Datenbanken zu Testzwecken einzurichten, bietet zudem die Container-Plattform Docker [Docker-Tutorial: Installation und erste Schritte] (https://hosting.1und1.de/digitalguide/server/konfiguration/docker-tutorial-installation-und-erste-schritte/).]

#### MongoDB ansprechen

Damit Ihre Anwendung mit der Datenbank interagieren kann, müssen Sie zunächst eine **Verbindung aufbauen**. Wie Sie dabei vorgehen, zeigt folgendes Codebeispiel:

*const mongodb = require ('mongodb');*

*const MongoClient = mongodb.MongoClient;*

*const connectionString = 'mongodb://localhost/myDatabase ';*

*MongoClient.connect(connectionString, {autoReconnect: true}, (err, database) => {*

*if (err) {*

*console.log('Failed to connect.', err.message);*

*process.exit(1);*

*}*

*console.log('Connected!')*

*});*

Zunächst rufen Sie das offizielle *mongodb-*Modul für Node.js über die Funktion *require()* auf und speichern dieses in der Konstante *mongodb*.

*const mongodb = require ('mongodb');*

Für die Interaktion mit der Datenbank stellt das Modul *mongodb* die Klasse ***MongoClient*** zur Verfügung. Auch diese speichern Sie in einer gleichnamigen Konstante. Diese fungiert später als Referenz und ermöglicht es, mit dem *MongoClient* zu interagieren.

*const MongoClient = mongodb.MongoClient;*

Anschließend definieren Sie einen sogenannten ***connectionString***.Dieser beinhaltet alle Informationen, die für den Zugriff auf die Datenbank benötigt werden: Benutzernamen und Passwort (optional), die IP-Adresse des Rechners, auf dem sich die Datenbank befindet (im aktuellen Beispiel localhost), der Port, über den Sie auf die Datenbank zugreifen möchten (standardmäßig 27017; kann weggelassen werden), sowie einen freiwählbaren Datenbanknamen.

Im nächsten Schritt übergeben Sie den *connectionString* dem *MongoClient*. Dazu sprechen Sie diesen über die zuvor definierte Konstante an und rufen die Funktion *connect()* auf.

*MongoClient.connect(connectionString, {autoReconnect: true}, (err, database) => {*

*if (err) {*

*console.log('Failed to connect.', err.message);*

*process.exit(1);*

*}*

*console.log('Connected!')*

*});*

Im aktuellen Beispiel werden dem *MongoClient* zusätzlich zum *connectionString* zwei weitere Parameter übergeben: das Objekt *autoReconnect: true*, das bei Verbindungsabbrüchen automatisch eine Neuverbindung einleitet, sowie ein Callback. Tritt ein Fehler auf, gibt dieser eine Meldung auf das Terminal aus und beendet das Programm. Sofern kein Fehler auftritt, ruft der Callback die abschließende *console.log*-Funktion auf, die Ihnen den Verbindungsstatus „Connected!“ auf das Terminal ausgibt.

[Hinweis: In der Praxis werden alle nun folgenden Codebeispiele in den oben dargestellten Callback nach der *console.log*-Zeile eingefügt.]

#### Collections ansprechen

In der Terminologie von MongoDB werden die schemalosen Tabellen innerhalb einer Datenbank **Collections** genannt. Zugriff auf eine Collection erhalten Sie durch die gleichnamige, vom *mongodb*-Modul bereitgestellte Funktion *collection()*.

In folgendem Beispiel greifen wir auf die Collection *users* zu.

*const users = database.collection('users')*

Beachten Sie: Wenn Sie eine Collection ansprechen, die noch nicht existiert, erzeugt MongoDB diese automatisch.

#### Dokumente einfügen

Datensätze werden in der Terminologie von MongoDB **Dokumente** genannt und als BSON-Objekte gespeichert. Gehen Sie wie folgt vor, um ein Dokument in einer Collection abzulegen.

Definieren Sie zunächst ein Dokument – beispielsweise das Dokument *user*:

*const user = {*

*firstName: 'John',*

*lastName: 'Doe'*

*};*

Speichern Sie das Dokument mithilfe der Funktion *insertOne()* in der eben angelegten Tabelle.

*users.insertOne(user, err => {*

*if (err) {*

*console.log('Failed to insert user.');*

*process.exit(1);*

*}*

*console.log('Successfully inserted user!')*

*database.close();*

*});*

Im aktuellen Beispiel speichern wir das Dokument *user* in der zuvor angelegten Tabelle *users* und hängen darüber hinaus eine Callback-Funktion an. Diese gibt im Fall eines Fehlers eine entsprechende Meldung aus und beendet das Programm. Wurde das Dokument erfolgreich in die Collection eingefügt, erhalten wir über eine *console.log*-Funktion die Statusmeldung„Successfully inserted user!“*.* Anschließend wird die Datenbankverbindung mithilfe der Funktion *close* geschlossen.

#### Dokumente abfragen

Datenbankabfragen laufen bei MongoDB über die ***find*-Funktion**. Mit dieser können Sie entweder einzelne oder alle Dokumente in einer Collection abfragen.

Mit folgendem Codebeispiel werden alle Dokumente der Collection *users* abgerufen und als Array auf das Terminal ausgegeben:

*users.find().toArray((err, documents) => {*

*if (err) {*

*console.log('Failed to find users.');*

*process.exit(1);*

*}*

*console.log(documents);*

*database.close();*

*});*

Möchten Sie ein bestimmtes Dokument einer Collection abrufen, nutzen Sie die Funktion *find()* in Kombination mit einem Suchparameter.

*user.find({*

*firstName: 'John'*

*}).toArray((err, documents) => {*

*if (err) {*

*console.log('Failed to find user.');*

*process.exit(1);*

*}*

*console.log(documents);*

*database.close();*

*});*

Im aktuellen Beispiel werden lediglich die Dokumente der Collection *users* ausgegeben, die einFeld *firstName* mit dem Inhalt *John* aufweisen.

Möchten Sie hingegen lediglich das erste Dokument abfragen, auf das das Suchkriterium zutrifft, verwenden Sie statt *find()* die Funktion *findOne()*.

*users.findOne({firstName: 'John'}, (err, document) => {…})*

#### Objekte löschen

Um Ihnen das Löschen von Dokumenten zu ermöglichen, bietet das Modul *mongodb* die **Funktionen *deleteMany()***und ***deleteOne()****.* Während *deleteOne()* lediglich das erste Dokument löscht, auf das das Suchkriterium zutrifft, löscht *deleteMany()* alle Dokumente mit entsprechender Übereinstimmung.

Die Funktionen *deleteMany()* und *deleteOne()* erwarten den Parameter *filter*.

*const filter = {*

*firstName: 'John'*

*lastName: 'Doe'*

*};*

Nutzen Sie die *deleteOne*-Funktion nach folgendem Beispiel, um das erste Dokument zu löschen, das das Suchkriterium erfüllt:

*users.deleteOne(filter, err => {*

*if (err) {*

*console.log('Failed to insert user.');*

*process.exit(1);*

*}*

*console.log('Successfully deleted one document!');*

*database.close();*

*});*

Sollten Sie alle Dokumente löschen wollen, die die in der *filter*-Konstante definierten Kriterien erfüllen, ersetzen Sie die *deleteOne-*Funktion durch *deleteMany()*:

*collection.deleteMany(filter, err => {*

*if (err) {*

*console.log('Failed to insert user.');*

*process.exit(1);*

*}*

*console.log('Successfully deleted documents!');*

*database.close();*

*});*

#### Dokumente aktualisieren

Möchten Sie Dokumente aktualisieren, nutzen Sie die **Funktionen *updateOne()*** und ***updateMany()***. Diese erwarten zwei Parameter: einen *filter*-Parameter, mit dem Sie definieren, welche Dokumente Sie aktualisieren möchten, und einen *update*-Parameter, mit dem Sie definieren, womit Sie die ausgewählten Dokumente überscheiben möchten. Während *updateOne()* lediglich das erste Dokument überschreibt, auf das das Suchkriteriumzutrifft, überschreibt *updateMany()* alle Dokumente mit entsprechender Übereinstimmung.

Möchten Sie ein Dokument aktualisieren, definieren Sie zunächst die Parameter *filter* und *update*.

*const filter = {*

*firstName: 'John',*

*lastName: 'Doe'*

*};*

*const update = {*

*firstName: 'John Thomas',*

*lastName: 'Doe'*

*};*

Anschließend rufen Sie die Funktion *updateOne()* auf.

*users.updateOne(filter, update, err => {*

*if (err) {*

*console.log('Failed to update user.');*

*process.exit(1);*

*}*

*console.log('Successfully updated!');*

*database.close();*

*});*

[Hinweis: Die Parameter *filter* und *update* können bei Bedarf auch inline und somit innerhalb der Funktion *updateOne()* definiert werden.]

Im aktuellen Beispiel ersetzen wir das erste Dokument mit den Feldern *firstName: 'John'* und *lastName: 'Doe'* miteinem Dokument mit denFeldern *firstName: 'John Thomas'* und *lastName: 'Doe'*. Beachten Sie, dass **Dokumente immer komplett überschrieben** werden. Der Parameter *update* gibt somit an, wie das ausgewählte Dokument nach der Aktualisierung aussehen soll. Möchten Sie lediglich einzelne Datenfelder eines Dokuments ersetzen, müssen Sie ein **partielles Update** durchführen. Nutzen Sie dazu den Update-Operator *$set.*

*const filter = {*

*firstName: 'John',*

*lastName: 'Doe'*

*};*

*const update = {*

*$set: { firstName: 'John Thomas' }*

*};*

*users.updateOne(filter, update, err => {*

*if (err) {*

*console.log('Failed to update user.');*

*process.exit(1);*

*}*

*console.log('Successfully updated!');*

*database.close();*

*});*

Im aktuellen Beispiel gibt der Operator *$set* an, dass im ersten Dokument, auf das das Filterkriterium zutrifft, lediglich der Wert im Datenfeld *firstName* durch *John Thomas* ersetzt werden soll.

Möchten Sie mehrere Datensätze mit *updateMany()* aktualisieren, muss immer ein Update-Operator verwendet werden. Weitere Informationen zu den **Update- und Abfrage-Operatoren von MongoDB** finden Sie in der Dokumentation des Datenbank-Management-Systems [Zur offiziellen MongoDB-Dokumentation] (https://docs.mongodb.com/manual/reference/operator/).

### Datenbankoperationen mit MySQL

Um eine Node.js-Anwendung mit einer MySQL-Datenbank zu verbinden, benötigen Sie ebenso wie bei MongoDB einen Datenbanktreiber. Diesen stellen diverse Drittanbietermodule zur Verfügung, die sich kostenlos über npm beziehen lassen. Ein offizielles MySQL-Modul für Node.js gibt es nicht. Wir empfehlen das **Modul *mysql*** von Douglas Wilson [Zur Modulbeschreibung auf www.npmjs.com] (https://www.npmjs.com/package/mysql).

Die Installation erfolgt nach diesem Schema:

*npm install mysql --save*

#### MySQL ansprechen

Folgendes Codebeispiel zeigt, wie Sie das Modul *mysql* nutzen, um eine **Datenbankverbindung** zu einer Node.js Anwendung aufzubauen.

*const mysql = require('mysql');*

*const connection = mysql.createConnection({*

*host: "localhost",*

*user: "yourusername",*

*password: "yourpassword"*

*});*

*connection.connect(function(err) {*

*if (err) {*

*console.log('Failed to connect.', err.message);*

*process.exit(1);*

*}*

*console.log('Connected!');*

*});*

Zunächst laden Sie das Modul *mysql* in Ihre Anwendung und speichern dieses in der Konstante *mysql*. Beachten Sie, dass auch dieses zuvor via npmauf ihrem System installiert werden muss.

*const mysql = require('mysql');*

Anschließend nutzen Sie die im *mysql*-Modul enthaltene Funktion *createConnection()*, um die **Verbindungsdaten** zur Datenbank zu definieren. Benötigt werden folgende Angaben: der Name des Servers (oder die IP-Adresse), auf dem die Datenbank liegt, Ihr Benutzername, Ihr Passwort sowie der Name der Datenbank, auf die Sie zugreifen möchten.

*const connection = mysql.createConnection({*

*host: "localhost",*

*user: "yourusername",*

*password: "yourpassword",*

*database: "yourdb"*

*});*

Damit Sie später auf die Funktion *createConnection()* sowie auf die als Parameter hinterlegten Verbindungsdaten zugreifen können, speichern Sie diese in der Konstante *connection.*

Der eigentliche **Verbindungsaufbau** erfolgt in Zeile 9. Dazu sprechen Sie die Konstante *connection* an und rufen die vom *mysql-*Modul bereitgestellte Funktion *connect()* auf.Dieser hängen Sie eine Callback-Funktion an, die eine Statusmeldung auf das Terminal ausgibt.

*connection.connect(function(err) {*

*if (err) {*

*console.log('Failed to connect.', err.message);*

*process.exit(1);*

*}*

*console.log('Connected!')*

*});*

Im Falle eines Fehlers erhalten Sie eine Fehlermeldung als Terminalausgabe. Anschließend wird das Programm beendet. War der Verbindungsaufbau erfolgreich, gibt die Anwendung den Status „Connected!“ *aus.*

#### Datenbankoperationen durchführen

Im Rahmen von Datenbankoperationen sprechen Sie ebenfalls die Konstante *connnection* an,die Sie im vorhergehenden Beispiel definiert haben, um auf *createConnection()* sowie auf die Parameter für den Verbindungsaufbau zuzugreifen. Alle Datenbankoperationen erfolgen mithilfe der Funktion *query()*. Diese erwartet als ersten Parameter ein **SQL-Statement**, das angibt, welche Operation durchgeführt werden soll und auf welche Daten sich diese bezieht. Die eigentliche Interaktion mit der Datenbank erfolgt somit in der Datenbanksprache SQL. Das SQL-Statement kann inline definiert oder in eine Variable/Konstante ausgelagert werden.

[Hinweis: Während MongoDB – wie für Node.js typisch – asynchron läuft, arbeitet MySQL synchron. Vergleichen Sie:

*connection.connect();*

*connection.query(…);*

und

*connection.connect(xy => {xy.query()})*

Dieser Umstand ist nicht der Datenbank geschuldet – es handelt sich um eine Entscheidung der Modulentwickler.]

Inline:

*connection.query('SELECT column FROM table', function(error, results, fields) {*

*if (err) {*

*console.log('Failed to select column Form table.', err.message);*

*process.exit(1);*

*}*

*console.log('The solution is: ', results[0].column;*

*});*

Ausgelagertes SQL-Statement:

*const sql = 'SELECT column FROM table';*

*connection.query(sql, function(err, results, fields) {*

*if (err) {*

*console.log('Failed to connect.', err.message);*

*process.exit(1);*

*}*

*console.log('The solution is: ', results[0].column);*

*});*

Der zweite Parameter der *query-*Funktion ist ein Callback mit drei Argumenten:

* **err** speichert Fehler, die während der Ausführung des SQL-Statements auftreten.
* **results** enthält die Ergebnisse der Abfrage.
* **fields** enthält Detailinformationen zu den betroffenen Datenfeldern.

Detaillierte Informationen dazu, wie Sie eine MySQL-Datenbank einrichten und SQL-Statements formulieren, um Datenbankoperationen durchzuführen, entnehmen Sie unserem MySQL-Tutorial für Einsteiger [MySQL-Lernen leicht gemacht] (https://hosting.1und1.de/digitalguide/server/knowhow/mysql-lernen-leicht-gemacht/).

**Teaser**

Mit der Software-Plattform Node.js nutzen Sie JavaScript serverseitig und profitieren auch bei der Entwicklung von Serveranwendungen von den Möglichkeiten der asynchronen, eventbasierten Programmierung. Wir geben Ihnen einen Überblick über den modularen Aufbau von Node.js, erklären, wie Sie Drittanbietermodule mithilfe des integrierten Paket-Managers npm nachinstallieren oder eigene Programmmodule entwickeln. Erstellen Sie mit nur acht Codezeilen Ihren eigenen Webserver und erfahren Sie, wie Sie Verbindungen zu Datenbanken auf Basis von MongoDB oder MySQL aufbauen.