Skript - Webentwicklung mit Nodejs

▼ Inhaltsverzeichnis

Inhaltsverzeichnis
1. Historischer Abriss
    1.1 Die Anfänge: 2009 und die Einführung von Node.js
    1.2 Warum JavaScript?
    1.3 Das Wachstum von npm: Die Modulare Revolution
    1.4 Der Aufstieg und die Popularität
    1.5 Kontroversen und Forks
    1.6 Vereinigung und Gründung der Node.js Foundation
    1.7 Aktuelle Entwicklungen
    1.8 Die Zukunft von Node.js
2. Einführung in die Webentwicklung mit Node.js
    2.1 Ziele
    2.2 Nutzen
    2.3 Ursprünge
3. Node.js Installation und Konfiguration
    3.1 Node.js-Installationspaket herunterladen
    3.2 Node.js installieren
    3.3 Überprüfen der Installation und des PATH
    3.4 Node Package Manager - NPM
        3.4.1 Was ist NPM?
        3.4.2 Beispiele für die Verwendung von NPM
        3.4.2.1 Installation eines Pakets
        3.4.2.2 Globale Installation eines Werkzeugs:
        3.4.2.3 Erstellen einer package.json-Datei:
        3.4.2.4 Andere Projekte klonen und Abhängigkeiten installieren:
    3.5 Node Version Manager - nvm
        3.5.1 Eigenschaften
        3.5.1.1 Installation von Node-Versionen
        3.5.1.2 Wechseln zwischen Versionen
        3.5.1.3 Isolierte Umgebungen
        3.5.1.4 Einfaches Entfernen von Versionen
        3.5.2 Vorteile
        3.5.2.1 Flexibilität
        3.5.2.2 Konsistenz

3.5.2.3 Sicherheit
3.5.2.4 Vereinfachte Konfiguration
3.5.3 Installation von nvm auf Windows
3.5.4 Beispiele von Befehlen
4. Modulsystem von Node.js
4.1 Einführung
4.2 Der Nutzen von Modulen
4.3 Ein Modul in Node.js
4.3.1 exports
4.3.2 module.exports
4.3.3 Das moduleObjekt
4.3.4 Die require() Funktion
4.4 Modulauflösung
4.5 Caching von Modulen
4.6 Erstellung von Modulen
4.6.1 Einfache Funktionen exportieren
4.6.2 Ein gesamtes Objekt mit module.exports exportieren
4.6.3 Mischen von exports und module.exports
4.7 Beispielanwendung
5. Dateisystem mit Node.js: Lesen, Schreiben und Bearbeiten
5.1 Asynchron mit Callback
5.1.1 Lesen
5.1.2 Schreiben
5.1.3 Bearbeiten
5.2 Asynchron mit Promises
5.2.1 Lesen
5.2.2 Schreiben
5.2.3 Bearbeiten
5.3 Synchron
5.3.1 Lesen
5.3.2 Schreiben
5.3.3 Bearbeiten
6. Kommandozeilenargumente
7. Debugging-Werkzeugen und Fehlersuche
7.1 Debugging mit Visual Studio Code unter Windows
7.1.1 Voraussetzungen
7.1.2 Debugging von Node.js-Anwendungen
7.1.2.1 Konfigurationsdatei erstellen
7.1.2.2 Breakpoints setzen
7.1.2.3 Debuggen starten
7.1.3 Debugging von Frontend-Code in Chrome
7.1.3.1 Erweiterung installieren

7.1.3.2 Konfigurationsdatei erstellen
7.1.3.3 Chrome mit Remote-Debugging starten
7.1.3.4 Debuggen starten
7.1.4 Debugging-Funktionen
8. Asynchronität mit Callbacks und Promises
8.1 Grundlagen der Asynchronität in Programmiersprachen
8.2 Warum Asynchronität?
8.3 Die Event Loop
8.4 Callbacks: Das Rückgrat der Node.js Asynchronität
8.5 Promises und Async/Await: Evolution der Asynchronität
8.6 Nicht nur I/O: Andere Anwendungen der Asynchronität
8.7 Herausforderungen und Überlegungen
9. Express.js
9.1 Grundlagen
9.2 Integration von Express mit Node.js
9.3 Routing mit Express
9.4 Middleware: Das Fundament von Express
9.5 Sicherheit
10. Socket.io
10.1 Wie funktioniert socket.io
10.1.1 Initialisierung
10.1.2 Verbindung
10.1.3 Ereignisse
10.2 Beispiel
10.3 Namespaces
10.3.1 Beispiele zur Nutzung von Namespaces:
10.4 Beispiel für eine ToDo Anwendung mit Node.js, Express.js und socket.io
10.4.1 Einrichten des Projekts
10.4.2 Serverseitiger Code (server.js)
10.4.3 Clientseitiger Code
11. Einführung in der Verwendung von Datenbanken in Node.js
11.1 Warum braucht man eine Datenbank in einer Node.js Anwendung?
11.1.1 Dauerhafte Datenspeicherung
11.1.2 Große Datenmengen
11.1.3 Konkurrenter Datenzugriff
11.1.4 Such- und Abfragefunktionalität
11.1.5 Transaktionen
11.1.6 Skalierbarkeit
11.1.7 Datenintegrität und -sicherheit
11.1.8 Relationen zwischen Daten
11.1.9 Backup und Wiederherstellung
11.1.10 Middleware-Integration

11.2 MongoDB
11.2.1 MongoDB-Installation
11.2.2 Node.js-Anbindung an MongoDB
11.2.2.1 MongoDB Native Driver
Quellcodeverzeichnis

1. Historischer Abriss

1.1 Die Anfänge: 2009 und die Einführung von Node.js

Ryan Dahl, ein talentierter Entwickler, stellte Node.js im Jahr 2009 der Welt vor. Er war frustriert über die bestehenden Webtechnologien, insbesondere die Art und Weise, wie sie mit I/O (Eingabe/Ausgabe), insbesondere Netzwerkanfragen, umgingen. Das traditionelle, thread-basierte Modell war in vielen Situationen ineffizient, insbesondere wenn es um Echtzeit-Webanwendungen ging.

Daher schuf Dahl Node.js, ein ereignisgesteuertes, nicht-blockierendes I/O-System in JavaScript. Es basierte, und immer noch basiert, auf Googles V8 JavaScript-Engine und ermöglichte es Entwicklern, serverseitige Anwendungen in JavaScript zu erstellen.

1.2 Warum JavaScript?

JavaScript war traditionell eine Sprache für den Browser, die Webseiten interaktiv machte. Aber Dahl sah das Potenzial für mehr. Mit Node.js konnten Entwickler nun Frontend und Backend in einer einzigen Sprache schreiben, was eine engere Integration und schnellere Entwicklung ermöglichte.

1.3 Das Wachstum von npm: Die Modulare Revolution

Im Jahr 2010, nur ein Jahr nach der Einführung von Node.js, wurde der Node Package Manager (npm) eingeführt. npm revolutionierte die Art und Weise, wie Node.js-Entwickler Software schrieben. Es ermöglichte ihnen den Zugriff auf eine riesige Bibliothek von "Paketen" oder "Modulen", kleinen Codeblöcken, die bestimmte Aufgaben erfüllen.

Innerhalb kürzester Zeit explodierte das npm-Ökosystem. Entwickler aus der ganzen Welt begannen, ihre Module zu teilen, und die Gemeinschaft wuchs rasant.

1.4 Der Aufstieg und die Popularität

Durch die Effizienz von Node.js und die Unterstützung durch npm wuchs die Popularität der Plattform schnell. Große Unternehmen begannen, Node.js für ihre Backend-Systeme zu verwenden. PayPal, LinkedIn und viele andere wechselten zu oder integrierten Node.js in ihre Architekturen.

1.5 Kontroversen und Forks

Aber nicht alles war rosig. Mit der wachsenden Popularität kamen auch Meinungsverschiedenheiten in der Gemeinschaft, insbesondere in Bezug auf die Richtung von Node.js. Das führte 2014 zu einem bedeutenden Fork namens "io.js". Ein "Fork" tritt auf, wenn Entwickler eine existierende Software kopieren und in eine andere Richtung weiterentwickeln.

io.js brachte viele neue Features und Aktualisierungen mit, und für eine Weile gab es eine ernsthafte Spaltung in der Community.

1.6 Vereinigung und Gründung der Node.js Foundation

Glücklicherweise erkannten beide Lager den Wert der Zusammenarbeit, und 2015 vereinigten sich Node.js und io.js wieder unter dem Dach der neu gegründeten Node.js Foundation. Dies war ein bedeutender Schritt, der die Zukunft von Node.js sicherte und die Basis für sein anhaltendes Wachstum legte.

1.7 Aktuelle Entwicklungen

Seit seiner Einführung hat sich Node.js ständig weiterentwickelt. Jedes Jahr gibt es bedeutende Updates, die Leistungsverbesserungen, neue Features und

Sicherheitsupdates bringen.

Ein bedeutendes Merkmal von Node.js ist seine Asynchronität. Mit den Neuerungen in ECMAScript, insbesondere Promises und das async/await-Schema, wurde das Schreiben von asynchronem Code in Node.js einfacher und lesbarer.

1.8 Die Zukunft von Node.js

Die Zukunft von Node.js sieht rosig aus. Mit einem starken Fundament, einer aktiven Gemeinschaft und der kontinuierlichen Integration neuer Webtechnologien bleibt Node.js an der Spitze der serverseitigen JavaScript-Entwicklung.

Das Ökosystem um Node.js herum, insbesondere Tools wie Electron für Desktop-Anwendungen und diverse Frameworks wie Express.js für Webentwicklung, sorgt dafür, dass Node.js für viele Jahre relevant bleibt.

2. Einführung in die Webentwicklung mit Node.js

2.1 Ziele

Die Webentwicklung hat das Ziel, Websites und Online-Anwendungen zu erstellen und zu pflegen. Mit Node.js können wir diese Ziele auf moderne und effiziente Weise erreichen:

- Interaktive Webseiten: In der heutigen digitalen Welt reicht es nicht mehr aus, nur statische Webseiten zu haben. Mit Node.js können Entwickler interaktive Webseiten erstellen, bei denen Benutzer Informationen in Echtzeit austauschen können, z.B. in Online-Spielen oder Chats.
- Schnelle und reaktionsfähige Anwendungen: Node.js verwendet ein ereignisgesteuertes, nicht-blockierendes I/O-Modell, was es ideal für Daten-intensive Echtzeitanwendungen macht. Dies sorgt dafür, dass Anwendungen reaktionsfähig bleiben, selbst wenn viele Benutzer gleichzeitig zugreifen.
- Modulare Entwicklung: Mit dem Node Package Manager (npm) können Entwickler auf tausende von Modulen zugreifen, die von der Community

entwickelt wurden. Dies erleichtert die Erstellung von Funktionen und spart Entwicklungszeit.

2.2 Nutzen

Die Verwendung von Node.js bringt mehrere Vorteile mit sich:

- Eine Sprache, zwei Anwendungsbereiche: Da Node.js auf JavaScript basiert, können Entwickler sowohl den Frontend (Teil der Webseite, den der Benutzer sieht) als auch den Backend (Datenverarbeitung, Datenbankzugriffe) in derselben Sprache programmieren.
- Aktive Community: Da Node.js sehr beliebt ist, gibt es eine große und aktive Gemeinschaft von Entwicklern. Das bedeutet, es gibt ständig Updates, viele Online-Ressourcen, Tutorials und Lösungen für häufige Probleme.
- Skalierbarkeit: Node.js ist ideal für skalierbare Anwendungen. Dies bedeutet, dass man mit der gleichen Codebasis sowohl wenige als auch sehr viele Benutzer bedienen kann.

2.3 Ursprünge

Ryan Dahl war frustriert über die damaligen Möglichkeiten, Echtzeitanwendungen im Web zu erstellen. Er sah das Potenzial von JavaScript, eine Sprache, die traditionell im Browser lief, um serverseitige Anwendungen zu ermöglichen.

Die Idee war, eine Umgebung zu schaffen, in der JavaScript außerhalb des Browsers laufen kann, um Webanwendungen effizienter und reaktionsfähiger zu machen. So entstand Node.js, und es revolutionierte die Art und Weise, wie moderne Webanwendungen entwickelt werden.

3. Node.js Installation und Konfiguration

3.1 Node.js-Installationspaket herunterladen

1. Besuche die offizielle Node.js-Website unter https://nodejs.org/.
2. Auf der Startseite siehst du Optionen zum Herunterladen von Node.js. Wähle die für deine Bedürfnisse passende Version. Für die meisten ist die empfohlene LTS-Version (Long Term Support) ideal.

3. Klicke auf die Version, um den Download zu starten. Das Installationspaket (.msi-Datei) wird heruntergeladen.

3.2 Node.js installieren

1. Sobald der Download abgeschlossen ist, öffne das Installationspaket, indem du darauf doppelklickst.
2. Das Installationsprogramm startet. Klicke auf „**Next**“.
3. Akzeptiere die Lizenzvereinbarung und klicke wieder auf „**Next**“.
4. Wähle den Installationsordner (standardmäßig ist das `C:\Program Files\nodejs\`). Klicke auf „**Next**“.
5. Lass alle Standardkomponenten ausgewählt (das sollte "Node.js runtime", "npm package manager" und "Online documentation shortcuts" beinhalten). Klicke auf „**Next**“.
6. Ein wichtiger Schritt: Achte darauf, dass die Option „Automatically install the necessary tools....“ nicht ausgewählt ist, es sei denn, du möchtest zusätzliche Tools installieren. Klicke auf „**Next**“.
7. Klicke auf „**Install**“, um die Installation zu starten.
8. Sobald die Installation abgeschlossen ist, klicke auf „**Finish**“.

3.3 Überprüfen der Installation und des PATH

Das Installationsprogramm von Node.js fügt Node und npm automatisch zum Windows `PATH` hinzu, sodass du es von der Kommandozeile aus überall ausführen kannst. Um zu überprüfen, ob die Installation erfolgreich war:

Öffne die Windows-Kommandozeile oder das Terminal (z.B. „cmd“ oder „PowerShell“).

Gib den folgenden Befehl ein, um die Node-Version zu überprüfen:

```
node -v
```

Gib den folgenden Befehl ein, um die npm-Version zu überprüfen:

```
npm -v
```

Wenn beide Befehle erfolgreich eine Versionsnummer zurückgeben, wurde Node.js richtig installiert und zum `PATH` hinzugefügt.

Jetzt kannst du Node.js und npm auf deinem Windows-Computer verwenden! Falls du Projekte starten oder Pakete aus dem npm-Repository installieren möchtest, bist du nun bestens ausgerüstet.

3.4 Node Package Manager - NPM

3.4.1 Was ist NPM?

NPM steht für "Node Package Manager". Es handelt sich um ein Werkzeug, das mit Node.js geliefert wird und zwei Hauptfunktionen erfüllt:

- Paketmanager: Mit npm können Entwickler JavaScript-Pakete von anderen teilen und nutzen. Es bietet einen einfachen Weg, Codebibliotheken zu installieren, die andere entwickelt haben, sodass du nicht alles von Grund auf neu schreiben musst.
- Werkzeug zur Projektverwaltung: NPM ermöglicht es Entwicklern, die Abhängigkeiten ihres Projekts (d.h. welche Pakete ihr Projekt benötigt) in einer Datei namens `package.json` zu definieren. Wenn jemand anderes das Projekt nutzen möchte, kann er einfach `npm install` ausführen, und alle benötigten Pakete werden automatisch installiert.

3.4.2 Beispiele für die Verwendung von NPM

3.4.2.1 Installation eines Pakets

Möchtest du die beliebte JavaScript-Bibliothek `lodash` in deinem Projekt verwenden? Anstatt den Code manuell herunterzuladen und in dein Projekt einzufügen, kannst du einfach folgenden Befehl ausführen:

```
npm install lodash
```

Jetzt steht dir `lodash` in deinem Projekt zur Verfügung.

3.4.2.2 Globale Installation eines Werkzeugs:

Einige Pakete sind nützliche Kommandozeilenwerkzeuge. Ein Beispiel ist "nodemon", ein Tool, das Node.js-Anwendungen überwacht und sie automatisch neu startet, wenn Änderungen erkannt werden. Um es global zu installieren (d.h. für alle deine Projekte zugänglich), würdest du ausführen:

```
npm install -g nodemon
```

3.4.2.3 Erstellen einer package.json-Datei:

Wenn du ein neues Projekt startest, möchtest du vielleicht eine package.json-Datei erstellen, um Informationen über dein Projekt und seine Abhängigkeiten zu speichern. Dazu würdest du:

```
npm init
```

ausführen und den Anweisungen folgen.

3.4.2.4 Andere Projekte klonen und Abhängigkeiten installieren:

Angenommen, du hast ein interessantes Projekt auf GitHub gefunden und möchtest es auf deinem Computer ausprobieren. Nachdem du es geklont hast, siehst du wahrscheinlich eine package.json-Datei im Hauptverzeichnis. Um alle Abhängigkeiten dieses Projekts zu installieren, navigiere einfach in das Verzeichnis und führe aus:

```
npm install
```

3.5 Node Version Manager - nvm

Der Node Version Manager, bekannt als nvm, ist ein essenzielles Werkzeug für jeden, der mit Node.js arbeitet, insbesondere wenn man unterschiedliche Versionen von Node.js für verschiedene Projekte handhaben muss. Es dient als Lösung, um mehrere Versionen von Node.js zu installieren und effektiv zu verwalten. In dieser Definition werden wir uns mit dem Hintergrund, den Funktionen und den Hauptvorteilen von nvm befassen.

Node.js hat sich im letzten Jahrzehnt als eine der führenden Plattformen für serverseitige Anwendungen etabliert. Während seiner Entwicklung hat Node.js viele Versionen durchlaufen, wobei jede Version ihre eigenen Verbesserungen, Funktionen und, in einigen Fällen, Abbrüche mit sich brachte. Dies führt dazu, dass Entwickler und Unternehmen oft mehrere Versionen von Node.js parallel verwenden müssen, je nach den Anforderungen ihrer Projekte. Die Verwaltung dieser Versionen manuell kann jedoch zu Komplikationen und Konflikten führen,

insbesondere wenn es um Abhängigkeiten, global installierte npm-Pakete und Systempfade geht. Hier kommt nvm ins Spiel.

3.5.1 Eigenschaften

3.5.1.1 Installation von Node-Versionen

Mit nur einem Befehl ermöglicht nvm die Installation beliebiger Node.js-Versionen. Dies bietet Entwicklern die Freiheit, mit neueren Versionen zu experimentieren, ohne ihre Hauptentwicklungsumgebung zu beeinträchtigen.

3.5.1.2 Wechseln zwischen Versionen

Ein entscheidendes Merkmal von nvm ist die Fähigkeit, nahtlos zwischen verschiedenen installierten Node-Versionen zu wechseln. Dies ist nützlich, wenn man zwischen Projekten wechselt, die unterschiedliche Node-Versionen benötigen.

3.5.1.3 Isolierte Umgebungen

Jede mit **nvm** installierte Node-Version verfügt über ihre eigene isolierte Umgebung. Dies bedeutet, dass globale npm-Pakete, die in einer Version installiert sind, keinen Einfluss auf andere Versionen haben. Dies verhindert Konflikte und stellt sicher, dass jedes Projekt genau die benötigten Abhängigkeiten hat.

3.5.1.4 Einfaches Entfernen von Versionen

Das Deinstallieren einer Node-Version ist mit nvm genauso einfach wie das Installieren. Mit nur einem Befehl kann eine unerwünschte oder nicht mehr benötigte Version entfernt werden.

3.5.2 Vorteile

3.5.2.1 Flexibilität

Mit **nvm** ist es möglich, problemlos mehrere Versionen von Node.js auf einem Computer zu installieren. Dies ermöglicht den Wechsel zwischen verschiedenen Versionen je nach Projektanforderungen. Durch die Fähigkeit, verschiedene Versionen von Node.js zu installieren, können Entwickler die neuesten Funktionen von Node.js testen, ohne die bestehende Entwicklungsumgebung zu stören.

3.5.2.2 Konsistenz

In Teamumgebungen gewährleistet nvm, dass alle Mitglieder mit der gleichen Node.js-Version arbeiten. Dies minimiert unerwartete Verhaltensunterschiede und stellt sicher, dass alle die gleiche Umgebung nutzen. Da jede Node-Version und ihre globalen Pakete isoliert sind, werden mögliche Konflikte zwischen den Abhängigkeiten verschiedener Projekte vermieden.

3.5.2.3 Sicherheit

Bei bekannt gewordenen Sicherheitslücken ermöglicht nvm das rasche Aktualisieren auf eine sicherere Node.js-Version. Trotz der Möglichkeit zur Aktualisierung können ältere, bewährte Versionen von Node.js beibehalten werden. Dies ermöglicht es, bei Bedarf auf eine stabilere Version zurückzugreifen.

3.5.2.4 Vereinfachte Konfiguration

nvm bietet Anfänger in der Node.js-Welt einen unkomplizierten Einstieg, indem es für eine konsistente Einrichtung sorgt. Beim Wechsel eines Computers oder bei der Neukonfiguration einer Umgebung erleichtert nvm den Wiederaufbau der gewünschten Node.js-Umgebung, ohne tiefgreifende manuelle Einstellungen vornehmen zu müssen.

3.5.3 Installation von nvm auf Windows

1. Besuche die GitHub-Seite des Projekts: https://github.com/coreybutler/nvm-windows.
2. Lade die neueste Version von nvm-setup.zip aus dem Abschnitt "Releases" herunter.
3. Entpacke das Archiv und führe die nvm-setup.exe aus, um das Programm zu installieren.

Hinweis: Bevor du nvm auf Windows installierst, solltest du alle vorhandenen Node.js-Versionen und die zugehörige PATH-Umgebungsvariable entfernen, um Konflikte zu vermeiden.

3.5.4 Beispiele von Befehlen

Node-Versionen auflisten

```
nvm list
```

Eine bestimmte Node-Version installieren:

```
nvm install 14.17.0
```

Dies wird Node.js v14.17.0 installieren.

Zu einer anderen Node-Version wechseln:

```
nvm use 12.18.0
```

Das setzt Node.js v12.18.0 als aktive Version.

Node-Version deinstallieren:

```
nvm uninstall 12.18.0
```

Aktuelle verwendete Node-Version anzeigen:

```
nvm current
```

4. Modulsystem von Node.js

4.1 Einführung

Das Modulsystem stellt einen zentralen Bestandteil von Node.js dar und ermöglicht es, Code zu modularisieren und wiederzuverwenden. Es beruht auf der CommonJS-Modulspezifikation. Durch dieses System kann Code in verschiedenen Dateien organisiert werden, was besonders bei umfangreichen Projekten nützlich ist.

4.2 Der Nutzen von Modulen

In den Anfangsphasen von JavaScript war es gängige Praxis, Code in einer einzigen oder nur wenigen großen Dateien zu organisieren. Doch mit wachsender Komplexität wurde offensichtlich, dass dieser Ansatz Grenzen hat. Das Aufteilen des Codes in Module bietet Vorteile in Sachen Lesbarkeit, Wartbarkeit und Wiederverwendbarkeit.

4.3 Ein Modul in Node.js

In Node.js entspricht ein Modul einer Datei. Jede Datei verfügt über einen eigenen, privaten Gültigkeitsbereich. Das bedeutet, dass beispielsweise Funktionen oder Variablen, die in einem Modul definiert sind, standardmäßig in anderen Modulen nicht sichtbar sind. Um Code zwischen Modulen zugänglich zu machen, werden `exports` und `require()` genutzt.

4.3.1 exports

`exports` ist ein Objekt, das als Ablage für Funktionen oder Werte fungiert, die aus einem Modul heraus bereitgestellt werden sollen.

```
exports.meineFunktion = function() { return "Hallo, Welt!"; }
```

Im oberen Beispiel wurde eine neue Funktion zu dem exports-Objekt hinzugefügt. Die Funktion ist jetzt durch das exports-Objekt aufrufbar.

4.3.2 module.exports

Wenn ein gesamtes Objekt (etwa eine Funktion oder Klasse) bereitgestellt werden soll, kommt `module.exports` zum Einsatz.

```
module.exports = class MeineKlasse { ... };
```

4.3.3 Das moduleObjekt

Zusätzlich zu **exports** existiert das **module**-Objekt, das Informationen zum aktuellen Modul enthält. Tatsächlich ist exports nur ein Alias für module.exports.

4.3.4 Die require() Funktion

Mit **require** lassen sich Module importieren. Ruft man ein Modul mit **require()** auf, liest Node.js die betreffende Datei, führt sie aus und gibt anschließend das **exports**-Objekt dieser Datei zurück.

4.4 Modulauflösung

Node.js verfolgt eine bestimmte Reihenfolge bei der Modulauflösung:

1. Kernmodule: Dies sind in Node.js integrierte Module wie **fs**, **path** oder **http**. Sie haben immer Vorrang.

2. Datei- und Ordnermodule: Ist kein Kernmodul verfügbar, wird nach einer Datei oder einem Ordner mit dem Namen des Moduls gesucht.
3. node_modules und npm-Pakete: Kann das Modul weder als Kernmodul noch als Datei oder Ordner identifiziert werden, wird im node_modules-Ordner nach ihm gesucht.

4.5 Caching von Modulen

Module, die einmal mittels **require()**aufgerufen wurden, werden von Node.js gecacht. Das bedeutet, dass bei wiederholtem Aufruf der Code nur beim ersten Mal ausgeführt wird. Dieses Caching steigert die Performance, kann aber auch zu unerwarteten Ergebnissen führen.

4.6 Erstellung von Modulen

In Node.js spielt das Konzept der Modularisierung eine zentrale Rolle. Es ermöglicht Entwicklern, den Code in separate, wiederverwendbare Einheiten - sogenannte Module - aufzuteilen. Dies fördert nicht nur die Lesbarkeit und Wartbarkeit des Codes, sondern erleichtert auch das Testen einzelner Codeblöcke.

Ein Modul in Node.js ist grundsätzlich eine Datei oder ein Verzeichnis, das mit Hilfe der CommonJS-Spezifikation in andere Dateien eingebunden werden kann. Module können Funktionen, Objekte oder Werte exportieren, die dann in anderen Dateien oder Modulen wiederverwendet werden können.

Um ein eigenes Modul zu erstellen, benötigt man lediglich eine .js-Datei. In dieser Datei definiert man die gewünschten Funktionen, Klassen oder Werte. Um sie exportierbar zu machen, verwendet man entweder das **exports**-Objekt oder **module.exports**.

Das **exports**-Objekt ermöglicht es, einzelne Funktionen oder Werte zu exportieren, sodass sie in anderen Dateien direkt angesprochen werden können.

Mit module.exports hingegen kann ein gesamtes Objekt, wie z. B. eine Klasse oder ein Array, exportiert werden. Hierbei wird der gesamte Inhalt des Moduls durch das exportierte Objekt ersetzt.

Im Folgenden sind einige konkrete Beispiele gegeben, die den Prozess des Exportierens und Importierens von Modulen in Node.js verdeutlichen.

4.6.1 Einfache Funktionen exportieren

`mathModule.js`

```
exports.addieren = function(x, y) {
    return x + y;
};

exports.subtrahieren = function(x, y) {
    return x - y;
};
```

Ein anderes Modul kann diese Funktionen wie folgt verwenden:

`app.js`

```
const math = require('./mathModule.js');

console.log(math.addieren(5, 3));       // Gibt 8 aus
console.log(math.subtrahieren(5, 3));  // Gibt 2 aus
```

4.6.2 Ein gesamtes Objekt mit module.exports exportieren

`personModule.js`

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    begruessen() {
        return `Hallo, mein Name ist ${this.name} und ich bin
    }
}

module.exports = Person;
```

Und die Verwendung dieses Moduls in einem anderen:

`app.js`

```
const Person = require('./personModule.js');

const peter = new Person('Peter', 30);
console.log(peter.begruessen());  // Gibt "Hallo, mein Name i
```

4.6.3 Mischen von exports und module.exports

`mixedModules.js`

```
exports.sagHallo = function() {
    return 'Hallo Welt!';
}

const data = {
    titel: 'Mein Titel',
    inhalt: 'Inhalt der Daten'
};

module.exports = data;
```

Verwenden des Moduls:

`app.js`

```
const mixedData = require('./mixedModule.js');

console.log(mixedData.titel);  // Gibt "Mein Titel" aus

// 'sagHallo' ist hier nicht verfügbar, da 'module.exports' d
```

Es ist wichtig zu beachten, dass in einem Modul, wenn sowohl **exports** als auch **module.exports** verwendet werden, **module.exports** den Wert von exports überschreibt. Daher sollten sie nicht im selben Modul gemischt werden, es sei denn, es besteht eine klare Absicht dahinter.

Mit diesen Beispielen sollte deutlich werden, wie das Exportieren und Importieren von Modulen in Node.js funktioniert. Es ermöglicht eine klare Trennung und Wiederverwendbarkeit des Codes, was insbesondere bei großen Anwendungen von großem Vorteil ist.

4.7 Beispielanwendung

Mit dem unteren Beispiel wird angezeigt, wie man die Module in einer einfachen ToDo-Liste-Anwendung verwendet werden können.

Zuerst wird die HTML-Seite erstellt.

`index.html`

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initia
    <title>To-Do Liste</title>
    <link rel="stylesheet" href="styles.css">
</head>

<body>
    <div class="container">
        <input type="text" id="todoInput" placeholder="Neues 
        <button id="addButton">Hinzufügen</button>
        <ul id="todoList"></ul>
    </div>
    <script type="module" src="app.js"></script>
</body>

</html>
```

Die Seite hat eine Textbox zum Eingeben vom ToDo, einen Knopf zum Hinzufügen vom ToDo und eine Liste, die die ToDos auflistet. Natürlich ist eine Seite ohne Style zu geschmacklos, deswegen wird es auch ein bisschen CSS hinzugefügt.

`styles.css`

```
.container {
    width: 300px;
    margin: 50px auto;
    font-family: Arial, sans-serif;
```

```
}

button {
    margin-top: 10px;
    padding: 5px 15px;
}

ul {
    margin-top: 20px;
    padding: 0;
    list-style-type: none;
}
```

Dann wird das Modul für das Erstellen des ToDos in einem Listelement hinzugefügt

`todoModule.js`

```
export function addTodo(todo) {
    const ul = document.getElementById('todoList');
    const li = document.createElement('li');
    li.textContent = todo;
    ul.appendChild(li);
}
```

Und am Ende kommt die Haupt-JavaScript-Datei

`app.js`

```
import { addTodo } from './todoModule.js';

document.getElementById('addButton').addEventListener('click'
    const todoInput = document.getElementById('todoInput');
    const todoValue = todoInput.value.trim();

    if (todoValue) {
        addTodo(todoValue);
        todoInput.value = '';
    }
});
```

In diesem Beispiel wurde eine einfache To-Do-Liste erstellt. Die Logik zum Hinzufügen von To-Dos wurde in einem separaten Modul (todoModule.js) untergebracht. Das Haupt-JavaScript (app.js) importiert dann die benötigte Funktion (addTodo) aus diesem Modul.

Das **type="module"** Attribut im script-Tag gibt an, dass der Browser den Code als ES6-Modul behandeln soll. Das ermöglicht den Einsatz von **import** und **export** direkt im Browser ohne zusätzliche Werkzeuge wie Bundler oder Transpiler.

Es ist empfehlenswert, einen lokalen Server zu nutzen oder das Beispiel auf einem Webserver zu hosten.

5. Dateisystem mit Node.js: Lesen, Schreiben und Bearbeiten

Das Arbeiten mit dem Dateisystem ist eine der Stärken von Node.js, da es I/O-Operationen effizient mit seinem asynchronen Modell handhabt. Node.js bietet das **fs** (File System) Modul, welches eine Vielzahl von Methoden zur Arbeit mit Dateien bietet. Dieses Modul enthält sowohl synchrone als auch asynchrone Methoden.

Die meisten Methoden im **fs** Modul stehen in zwei Formen zur Verfügung: asynchron und synchron.

5.1 Asynchron mit Callback

Diese Funktionen sind nicht-blockierend und nutzen Callbacks oder Promises, um Operationen auszuführen. In der Regel haben sie den Postfix Async, wenn sie mit Promises arbeiten.

5.1.1 Lesen

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

fs.readFile(beispiel.txt', 'utf8', (err, data) => {
    if (err) {
        console.error('Es gab einen Fehler beim Lesen der Dat
        return;
    }
```

```
    console.log(data);
});
```

Hier wird fs.readFile() verwendet, um den Inhalt von beispiel.txt asynchron zu lesen. Der Callback wird aufgerufen, sobald die Datei vollständig gelesen ist, oder wenn ein Fehler auftritt.

5.1.2 Schreiben

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

const content = 'Dies ist ein Beispieltext.';

fs.writeFile(beispiel.txt', content, 'utf8', err => {
    if (err) {
        console.error('Es gab einen Fehler beim Schreiben in
        return;
    }
    console.log('Datei wurde erfolgreich geschrieben!');
});
```

Hier wird die Methode fs.writeFile() genutzt, um Daten in eine Datei asynchron zu schreiben. Der Callback wird aufgerufen, sobald der Schreibvorgang abgeschlossen ist oder ein Fehler auftritt.

5.1.3 Bearbeiten

Bei asynchroner Bearbeitung werden Callbacks oder Promises verwendet, um den Inhalt zu lesen, zu ändern und dann zurückzuschreiben. Das folgende Beispiel zeigt, wie das funktioniert

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

// 1. Lesen der Datei
fs.readFile('example.txt', 'utf8', (readErr, data) => {
    if (readErr) {
        console.error('Fehler beim Lesen der Datei:', readErr
        return;
    }
```

```
    // 2. Ändern des Inhalts
    const modifiedData = data.replace('alter Text', 'neuer Te

    // 3. Schreiben des geänderten Inhalts zurück in die Date
    fs.writeFile('example.txt', modifiedData, 'utf8', writeEr
        if (writeErr) {
            console.error('Fehler beim Schreiben in die Datei
            return;
        }
        console.log('Datei erfolgreich bearbeitet.');
    });
});
```

5.2 Asynchron mit Promises

5.2.1 Lesen

```
const fs = require('fs').promises;

async function readFile() {
    try {
        const data = await fs.readFile('beispiel.txt', 'utf8'
        console.log(data);
    } catch (error) {
        console.error('Fehler beim Lesen der Datei:', error);
    }
}

readFile();
```

In diesem Beispiel wird die Datei beispiel.txt asynchron gelesen und ihr Inhalt wird dann auf der Konsole ausgegeben.

5.2.2 Schreiben

```
const fs = require('fs').promises;
```

```
async function writeFile() {
    try {
        await fs.writeFile('beispiel.txt', 'Dies ist ein neue
        console.log('In Datei erfolgreich geschrieben.');
    } catch (error) {
        console.error('Fehler beim Schreiben in die Datei:',
    }
}

writeFile();
```

Hier wird ein neuer Inhalt in die Datei example.txt geschrieben.

5.2.3 Bearbeiten

```
const fs = require('fs').promises;

async function editFile() {
    try {
        // 1. Lesen der Datei
        const data = await fs.readFile('beispiel.txt', 'utf8'

        // 2. Ändern des Inhalts
        const modifiedData = data.replace('alter Text', 'neue

        // 3. Schreiben des geänderten Inhalts zurück in die
        await fs.writeFile('beispiel.txt', modifiedData, 'utf
        console.log('Datei erfolgreich bearbeitet.');

    } catch (err) {
        console.error('Fehler beim Bearbeiten der Datei:', er
    }
}

editFile();
```

In diesem Beispiel wird die async/await Syntax verwendet, die auf Promises aufbaut. Die Funktion editFile ist mit dem Schlüsselwort async markiert, was bedeutet, dass sie automatisch ein Promise zurückgibt. Innerhalb der Funktion

können asynchrone Operationen mit await angehalten werden, bis sie abgeschlossen sind, wodurch der Code so aussieht und sich so verhält, als wäre er synchron, obwohl er tatsächlich asynchron ist. Das Error-Handling erfolgt über einen try/catch-Block, der alle Fehler während der asynchronen Operationen abfängt. Diese Art des Arbeitens macht den Code oft sauberer und einfacher zu verstehen, insbesondere bei komplexeren asynchronen Abläufen, und ist ein großer Schritt weg von der sogenannten "Callback-Hölle".

5.3 Synchron

Diese Funktionen blockieren den Event-Loop und haben in der Regel den Postfix `Sync` am Ende ihres Namens.

In den meisten Fällen sollten Sie die asynchronen Methoden verwenden, um die Nicht-Blockierungs-Vorteile von Node.js zu nutzen. Die synchronen Methoden können nützlich sein, wenn Sie während des Startvorgangs Ihres Programms Dateioperationen ausführen müssen und nicht möchten, dass andere Operationen gleichzeitig stattfinden.

5.3.1 Lesen

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

try {
    const data = fs.readFileSync(beispiel.txt', 'utf8');
    console.log(data);
} catch (err) {
    console.error('Es gab einen Fehler beim Lesen der Datei!'
}
```

In diesem Beispiel verwendet man `fs.readFileSync()`, um den Inhalt der Datei synchron zu lesen. Das bedeutet, dass das Programm blockiert wird, bis die Datei vollständig gelesen ist.

5.3.2 Schreiben

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

const content = 'Dies ist ein Beispieltext.';
```

```
try {
    fs.writeFileSync('beispiel.txt', content, 'utf8');
    console.log('Datei wurde erfolgreich geschrieben!');
} catch (err) {
    console.error('Es gab einen Fehler beim Schreiben in die
}
```

In diesem Beispiel wird fs.writeFileSync() verwendet, um den Text synchron in beispiel.txt zu schreiben. Wie beim synchronen Lesen wird das Programm während des Schreibvorgangs blockiert.

5.3.3 Bearbeiten

Synchrones Bearbeiten blockiert den Event-Loop während jeder Operation. Das folgende Beispiel zeigt, wie man eine Datei synchron bearbeitet.

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

try {
    // 1. Lesen der Datei
    const data = fs.readFileSync('beispiel.txt', 'utf8');

    // 2. Ändern des Inhalts
    const modifiedData = data.replace('alter Text', 'neuer Te

    // 3. Schreiben des geänderten Inhalts zurück in die Date
    fs.writeFileSync('beispiel.txt', modifiedData, 'utf8');
    console.log('Datei erfolgreich bearbeitet.');
} catch (err) {
    console.error('Fehler beim Bearbeiten der Datei:', err);
}
```

In der Praxis kann die Wahl zwischen synchronem und asynchronem Bearbeiten von den spezifischen Anforderungen und dem Kontext abhängen. Asynchrone Methoden sind besonders nützlich, um den Event-Loop in leistungskritischen Anwendungen, wie Webservern, nicht zu blockieren. Synchrones Bearbeiten kann in Szenarien bevorzugt werden, in denen eine sequenzielle Ausführung erforderlich ist und die Blockierung des Event-Loops akzeptabel ist.

6. Kommandozeilenargumente

In Node.js werden Kommandozeilenargumente durch das eingebaute process-Objekt bereitgestellt. Insbesondere enthält process.argv ein Array mit den gesamten Kommandozeilenargumenten.

Das Array process.argv hat folgende Struktur:

- process.argv[0]: Pfad zur Node.js ausführbaren Datei.
- process.argv[1]: Pfad der gerade ausgeführten Datei.
- process.argv[2], process.argv[3], ...: Die restlichen Einträge sind die übergebenen Argumente, beginnend ab dem dritten Index.

Einfache Anzeige aller Kommandozeilenargumente:

`argv.js`

```
console.log(process.argv);
```

Wenn man dieses Skript mit node argv.js arg1 arg2 arg3 ausführt, bekommt man:

```
[  '/pfad/zu/node',   '/pfad/zu/argv.js',  'arg1',   'arg2',   'arg3']
```

Das direkte Arbeiten mit process.argv kann jedoch umständlich sein, da manuell die Argumente und deren Indizes behandelt werden müssen.

Ein anderes Beispiel ist die Summe von zwei Zahlen:

`summe.js`

```
const num1 = parseFloat(process.argv[2]);
const num2 = parseFloat(process.argv[3]);

console.log(`Die Summe von ${num1} und ${num2} ist ${num1 + n
```

Ausführen mit **'node summe.js 5 8'** gibt **'Die Summe von 5 und 8 ist 13.'** aus.

Für komplexere Anwendungen gibt es Pakete wie yargs oder commander, die helfen, Kommandozeilenargumente zu parsen und Anwendungen mit komplexeren Eingabeschemata zu erstellen.

`summeYargs.js`

```
const yargs = require('yargs');

const argv = yargs
  .option('a', {
      alias: 'num1',
      description: 'Erste Zahl',
      type: 'number'
  })
  .option('b', {
      alias: 'num2',
      description: 'Zweite Zahl',
      type: 'number'
  })
  .help()
  .alias('help', 'h')
  .argv;

const sum = argv.num1 + argv.num2;
console.log(`Die Summe ist: ${sum}`);
```

Ausführen mit ‘node sumYargs.js --num1=5 --num2=8’ gibt ‘**Die Summe ist: 13**’ aus.

In Node.js werden die Kommandozeilenargumente standardmäßig durch das process.argv-Array bereitgestellt. Während dieses Array für einfache Aufgaben ausreichend sein kann, kann es sich schnell als umständlich erweisen, wenn man mit einer größeren Anzahl von Argumenten oder komplexeren CLI-Szenarien arbeitet.

Für fortgeschrittene Anwendungen bieten Module wie yargs und commander Funktionen zum Parsen von Argumenten, Validierung, Generierung von Hilfe-Texten und vieles mehr. Sie ermöglichen es Entwicklern, leistungsstarke CLI-Tools und -Anwendungen mit Node.js auf intuitive und effiziente Weise zu erstellen. Bei der Erstellung von CLI-Tools ist es ratsam, die Bedürfnisse und Komplexität des Projekts zu berücksichtigen und dann zu entscheiden, ob man mit dem eingebauten process.argv arbeiten möchte oder auf spezialisierte Module zurückgreift.

7. Debugging-Werkzeugen und Fehlersuche

7.1 Debugging mit Visual Studio Code unter Windows

Visual Studio Code (VS Code) bietet eine integrierte Debugging-Funktionalität, die hilfreich ist, um Code in Echtzeit auszuführen, anzuhalten und zu überprüfen. Im Folgenden wird erläutert, wie man mit dieser Funktion in einer Windows-Umgebung arbeitet.

7.1.1 Voraussetzungen

- VS Code: Muss installiert sein.
- Node.js: Erforderlich für das Debuggen von Node.js-Anwendungen.
- Debugger für Chrome: Eine Erweiterung für VS Code, falls das Debuggen von Frontend-Code in Chrome geplant ist.

7.1.2 Debugging von Node.js-Anwendungen

7.1.2.1 Konfigurationsdatei erstellen

Nach dem Öffnen des Node.js-Projekts in VS Code geht man zum Debug-Menü in der linken Seitenleiste (Symbol, das wie ein Käfer aussieht). Hier wählt man "Konfiguration hinzufügen", dann "Node.js". Dies erstellt eine launch.json-Datei im Projektordner mit Debug-Konfigurationen.

7.1.2.2 Breakpoints setzen

Breakpoints werden gesetzt, indem links neben der Zeilennummer geklickt wird. Ein roter Punkt symbolisiert den Breakpoint.

7.1.2.3 Debuggen starten

Der Debugging-Prozess wird durch Klicken auf den grünen Pfeil im Debug-Menü gestartet. Bei Erreichen eines Breakpoints hält der Code an.

7.1.3 Debugging von Frontend-Code in Chrome

7.1.3.1 Erweiterung installieren

Die "Debugger für Chrome"-Erweiterung wird aus dem VS Code-Marktplatz installiert.

7.1.3.2 Konfigurationsdatei erstellen

Ähnlich wie oben, aber "Chrome" wird aus den Umgebungsoptionen gewählt.

7.1.3.3 Chrome mit Remote-Debugging starten

Damit eine Verbindung zwischen VS Code und Chrome möglich ist, muss Chrome mit Remote-Debugging gestartet werden. Dies kann durch die Debug-Konfiguration in VS Code gesteuert werden.

7.1.3.4 Debuggen starten

Der Frontend-Code wird in einem Chrome-Tab geöffnet und kann dort debuggt werden.

7.1.4 Debugging-Funktionen

- Watch: Erlaubt das Überwachen spezifischer Variablen in Echtzeit.
- Call Stack: Zeigt den aktuellen Aufrufstapel und ermöglicht das Navigieren zwischen den Ebenen.
- Steuerungen: Erlauben das schrittweise Durchlaufen des Codes sowie das Hineinspringen oder Überspringen von Funktionen.

Das Debugging in VS Code ermöglicht eine detaillierte Analyse des Codes während seiner Ausführung, was das Identifizieren und Beheben von Fehlern vereinfacht.

8. Asynchronität mit Callbacks und Promises

Die asynchrone Natur von Node.js ist ein entscheidender Faktor für seine Popularität und Effizienz als Plattform. Durch die Verwendung eines asynchronen, ereignisgesteuerten Modells kann Node.js Aufgaben in einer nicht-blockierenden Weise behandeln, was insbesondere bei I/O-intensiven Operationen vorteilhaft ist.

8.1 Grundlagen der Asynchronität in Programmiersprachen

Um den Wert von Node.js in Bezug auf Asynchronität zu verstehen, ist es hilfreich, einen Hintergrund darüber zu haben, wie Synchronität und Asynchronität in der Programmierung funktionieren. In einer synchronen Umgebung wird der Code sequenziell ausgeführt, von oben nach unten, wobei jede Operation den Fortschritt des Codes blockiert, bis sie abgeschlossen ist. Das ist in vielen Fällen in Ordnung, kann aber problematisch sein, wenn eine der Operationen – wie eine Datenbankabfrage oder eine HTTP-Anforderung – viel Zeit in Anspruch nimmt.

Asynchrone Programmierung wurde als Antwort auf dieses Problem entwickelt. Anstatt den Fortschritt zu blockieren, während sie auf das Ende einer langsamen Operation warten, gehen asynchrone Systeme weiter und führen andere Aufgaben aus. Wenn die langsame Operation abgeschlossen ist, wird eine zuvor festgelegte Funktion (oft als "Callback" bezeichnet) mit dem Ergebnis dieser Operation aufgerufen.

Synchroner Code wird sequenziell ausgeführt, wodurch zeitaufwändige Operationen alles andere blockieren können.

```
function fetchData() {
  // Nehmen wir an, dies dauert 5 Sekunden
  return "Daten geholt!";
}
console.log("Start");
const data = fetchData();
console.log(data);
console.log("Ende");
```

8.2 Warum Asynchronität?

Node.js wurde speziell für Netzwerkanwendungen entwickelt. Netzwerkaufgaben, wie z.B. das Abfragen von Daten aus einer Datenbank oder das Anfordern von Informationen von einem anderen Server, sind oft I/O-intensiv und können zeitintensiv sein. Ein blockierendes, synchrones Modell wäre hier suboptimal, da es den Server daran hindern würde, auf andere eingehende Anfragen zu reagieren, während er auf das Ende einer dieser langsamen I/O-Operationen wartet.

Node.js nutzt daher das asynchrone, ereignisgesteuerte Modell, um sicherzustellen, dass es während der Ausführung von I/O-Operationen weiterhin Anfragen bearbeiten kann.

8.3 Die Event Loop

Im Zentrum von Node.js' asynchroner Architektur steht die Event Loop, die durch die libuv-Bibliothek implementiert wird. Es ist im Grunde eine Schleife, die kontinuierlich überprüft, ob neue Aufgaben vorhanden sind und ob bereits gestartete Aufgaben abgeschlossen wurden.

Wenn Sie einen asynchronen Code in Node.js ausführen, wie z.B. eine Datenbankabfrage, wird diese Aufgabe gestartet, und die Kontrolle wird sofort an die Event Loop zurückgegeben, anstatt auf das Ende der Abfrage zu warten. Die Event Loop überprüft kontinuierlich den Status dieser Abfrage und, sobald sie abgeschlossen ist, wird der bereitgestellte Callback ausgeführt.

8.4 Callbacks: Das Rückgrat der Node.js Asynchronität

Das primäre Konstrukt, das in den frühen Tagen von Node.js für Asynchronität verwendet wurde, ist der Callback. Ein Callback ist einfach eine Funktion, die als Argument an eine andere Funktion übergeben und aufgerufen wird, sobald die externe Funktion ihre Ausführung abgeschlossen hat.

Das Konzept ist einfach, aber bei komplexen Anwendungen kann es zu "Callback Hell" oder "Pyramid of Doom" führen, wo mehrere verschachtelte Callbacks den Code schwer lesbar und wartbar machen.

Mit den Callback-Funktionen, die übergeben und ausgeführt werden, sobald eine Aufgabe abgeschlossen ist.

```
function fetchData(callback) {
  setTimeout(() => {
    callback("Daten geholt!");
  }, 5000);
}

console.log("Start");
fetchData((data) => {
  console.log(data);
```

```
});
console.log("Ende");
```

8.5 Promises und Async/Await: Evolution der Asynchronität

Mit der Zeit wurde klar, dass eine bessere, strukturiertere Methode zur Behandlung asynchroner Abläufe benötigt wurde. Hier kamen Promises ins Spiel. Ein Promise ist ein Objekt, das das Ergebnis einer asynchronen Operation repräsentiert. Anstatt einen Callback als Argument zu übergeben, gibt die asynchrone Funktion sofort ein Promise-Objekt zurück, das später entweder erfüllt (erfolgreich abgeschlossen) oder abgelehnt (mit einem Fehler beendet) wird.

Dies führte zur Einführung von async/await in der Sprache. Dies sind spezielle Syntaxkonstrukte, die es ermöglichen, asynchronen Code zu schreiben, der fast wie synchroner Code aussieht. Eine Funktion, die mit async deklariert ist, gibt immer ein Promise zurück, und innerhalb einer solchen Funktion können Sie await verwenden, um auf das Ergebnis eines anderen Promises zu warten.

Promisses bieten eine sauberere Methode zur Handhabung von asynchronem Code. async/await ermöglicht es, asynchronen Code synchron aussehen zu lassen.

```
function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Daten geholt!");
    }, 5000);
  });
}

console.log("Start");
fetchData().then((data) => {
  console.log(data);
});
console.log("Ende");
```

Mit async/await:

```
async function fetchDataAndDisplay() {
  try {
    console.log("Start");
    const data = await fetchData();
    console.log(data);
    console.log("Ende");
  } catch (error) {
    console.error("Ein Fehler ist aufgetreten:", error);
  }
}

fetchDataAndDisplay();
```

8.6 Nicht nur I/O: Andere Anwendungen der Asynchronität

Während I/O-Aufgaben das häufigste Anwendungsgebiet für Asynchronität in Node.js sind, gibt es auch andere Fälle, in denen Asynchronität nützlich sein kann. Dies umfasst CPU-intensive Aufgaben, die aufgeteilt und über Zeiträume ausgeführt werden können, ohne den Hauptthread zu blockieren, und die Integration mit externen Systemen, die ihre eigene asynchrone Natur haben.

8.7 Herausforderungen und Überlegungen

Asynchronität ist mächtig, aber sie kommt auch mit ihren eigenen Herausforderungen. Fehlerbehandlung, Synchronisation von asynchronen Abläufen und das Verständnis des genauen Ablaufs von asynchronem Code können für Entwickler, die neu in diesem Paradigma sind, Herausforderungen darstellen.

Node.js' asynchrone Natur ist ein Schlüsselmerkmal, das es zu einer bevorzugten Plattform für die Entwicklung schneller, effizienter und skalierbarer Anwendungen gemacht hat. Durch das Verständnis und die richtige Nutzung der Tools und Muster, die Node.js für asynchrone Programmierung bietet, können Entwickler Anwendungen erstellen, die in der Lage sind, hohe Lasten mit geringem Overhead zu bewältigen.

9. Express.js

Express.js, oft schlicht als "Express" bezeichnet, ist ein Web-Framework, das innerhalb des Node.js-Ökosystems eine dominante Position einnimmt. Es fungiert als Schnittstelle zwischen Node.js und Webanwendungen und bietet Werkzeuge, die den Entwicklungsprozess von Webservern und APIs deutlich vereinfachen.

9.1 Grundlagen

Express stellt eine leichte, flexibel anpassbare Plattform zur Verfügung, die den spezifischen Anforderungen von Webanwendungen und APIs gerecht wird. Es bietet eine einfache API, um Routinen zu definieren, Requests zu verarbeiten und Responses an den Client zu senden.

Ein einfacher Express-Server.

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
    res.send('Hallo Welt!');
});

app.listen(3000, () => {
    console.log('Server läuft auf Port 3000');
});
```

9.2 Integration von Express mit Node.js

Während Node.js eine Plattform zum Erstellen und Ausführen von JavaScript außerhalb des Browsers bietet, stellt Express die notwendigen Mittel bereit, um den Server-Teil einer Webanwendung effizient zu gestalten. Express setzt vollständig auf den Funktionen von Node.js auf und erweitert diese, um Web-spezifische Anforderungen zu erfüllen.

9.3 Routing mit Express

Das Routing in Express.js bezieht sich auf die Definition von Endpunkten (URIs) für eine Anwendung und wie diese auf Client-Anfragen reagieren. Ein Router ist im Grunde genommen eine Mittelschicht, die bestimmt, was mit einer

eingehenden Anfrage passiert, basierend auf der HTTP-Methode (z.B. GET, POST, PUT, DELETE) und dem Pfad (z.B. /, /about, /api/products).

Im Kern wird durch das Routing in Express eine bestimmte Funktion, oft als "Handler" oder "Controller-Funktion" bezeichnet, aufgerufen, wenn eine bestimmte Anfrage auf eine bestimmte Route trifft.

```
const express = require('express');
const app = express();

// Ein Handler für GET-Anfragen auf den Pfad "/"
app.get('/', (req, res) => {
    res.send('Homepage');
});

// Ein Handler für GET-Anfragen auf den Pfad "/about"
app.get('/about', (req, res) => {
    res.send('Über uns');
});

app.listen(3000, () => {
    console.log('Server läuft auf Port 3000');
});
```

9.4 Middleware: Das Fundament von Express

In der Entwicklung von Webanwendungen mit Node.js, insbesondere beim Verwenden des Express.js-Frameworks, bezeichnet der Begriff "Middleware" eine Serie von Funktionen, die zwischen dem Erhalt einer Anfrage und dem Senden einer Antwort eingreifen. Middleware-Funktionen haben Zugriff auf das Anfrageobjekt (req), das Antwortobjekt (res) und die nächste Middleware-Funktion (next) in der Anwendungsanfrage-Antwort-Zykluskette.

Eine einfache Middleware, die das aktuelle Datum und die Uhrzeit für jede Anfrage protokolliert.

```
app.use((req, res, next) => {
    console.log('Anfrage empfangen am:', Date.now());
```

```
    next();
});
```

Die Hauptaufgabe von Middleware ist die Ausführung von Code, das Ändern von Anforderungen und Antworten, das Beenden des Anfrage-Antwort-Zyklus oder das Aufrufen der nächsten Middleware im Stapel. Wenn eine Middleware nicht die Antwort an den Client zurücksendet, muss sie next() aufrufen, um den Kontrollfluss an die nächste Middleware weiterzugeben.

Eine eigene Middleware zu schreiben ist einfach. Sie ist lediglich eine Funktion mit den Argumenten req, res und next.

```
function meineMiddleware(req, res, next) {
    // tue etwas mit req und/oder res
    next();
}
app.use(meineMiddleware);
```

Die Reihenfolge, in der Middleware-Funktionen aufgerufen werden, ist entscheidend. Middleware-Funktionen werden in der Reihenfolge aufgerufen, in der sie eingefügt und definiert wurden.

Express bietet zahlreiche vordefinierte Middleware-Funktionen, um gängige Aufgaben zu erleichtern:

- `express.static`: Stellt statische Dateien aus einem bestimmten Verzeichnis bereit, zB Middleware, die dazu verwendet wird, statische Ressourcen wie Bilder, CSS-Dateien und JavaScript-Dateien bereitzustellen.

  ```
  app.use(express.static('public'));
  ```

- `express.json`: Parst eingehende Anfragen mit JSON-Inhalten.

9.5 Sicherheit

Express-Anwendungen müssen, wie alle Webanwendungen, vor einer Vielzahl von Webbedrohungen geschützt werden. Das Framework selbst und zahlreiche Middleware-Module bieten Sicherheitsfunktionen, um gängige Angriffe wie Cross-Site Scripting oder SQL-Injection zu verhindern.

10. Socket.io

Socket.io ist eine beliebte Bibliothek für Node.js, die es ermöglicht, Echtzeit-Webanwendungen zu erstellen. Es verwendet WebSockets und andere Transporttechnologien, um eine bidirektionale Kommunikation zwischen dem Client und dem Server zu ermöglichen. Das bedeutet, dass sowohl der Server als auch der Client aktiv Daten senden und empfangen können, ohne dass eine Seite eine Anfrage an die andere stellen muss.

10.1 Wie funktioniert socket.io

10.1.1 Initialisierung

Beim Starten des Servers wird ein Socket.io-Server erstellt, der auf Verbindungen von Clients wartet.

10.1.2 Verbindung

Ein Client stellt über ein socket.io-Client-Skript eine Verbindung zum Socket.io-Server her. Diese Verbindung wird normalerweise über WebSockets hergestellt, kann aber bei Bedarf zu anderen Transportmethoden wie Long Polling zurückfallen.

10.1.3 Ereignisse

Nachdem eine Verbindung hergestellt wurde, können sowohl der Server als auch der Client Ereignisse auslösen und darauf hören. Dies ermöglicht eine Zwei-Wege-Kommunikation.

10.2 Beispiel

Server (server.js) mit Express und socket.io:

```
const express = require('express');
const http = require('http');
const socketIo = require('socket.io');

const app = express();
const server = http.createServer(app);
const io = socketIo(server);
```

```
io.on('connection', (socket) => {
    console.log('Ein Benutzer hat sich verbunden.');

    // Hören auf ein 'chat message'-Ereignis vom Client
    socket.on('chat message', (msg) => {
        io.emit('chat message', msg); // Nachricht an alle ve
    });

    socket.on('disconnect', () => {
        console.log('Ein Benutzer hat die Verbindung getrennt
    });
});

server.listen(3000, () => {
    console.log('Server läuft auf http://localhost:3000');
});
```

Client (index.html) mit socket.io-Client:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Chat mit Socket.io</title>
</head>
<body>
    <ul id="messages"></ul>
    <form id="form" action="">
        <input id="input" autocomplete="off" /><button>Senden
    </form>

    <script src="/socket.io/socket.io.js"></script>
    <script src="https://code.jquery.com/jquery-3.6.0.min.js"
    <script>
        const socket = io();
        $('form').submit(() => {
            socket.emit('chat message', $('#input').val());
            $('#input').val('');
```

```
            return false;
        });

        socket.on('chat message', (msg) => {
            $('#messages').append($('<li>').text(msg));
        });
    </script>
</body>
</html>
```

Das Beispiel stellt eine einfache Echtzeit-Chat-Anwendung dar, die Socket.io verwendet, um eine bidirektionale Kommunikation zwischen dem Server und dem Client zu ermöglichen. Hier sind die Schritte, wie die Verbindung und die Kommunikation in diesem Beispiel aufgebaut und gehandhabt werden:

1. Server-Setup:
    a. Ein Express-Server wird erstellt und http wird verwendet, um einen neuen HTTP-Server basierend auf der Express-App zu erstellen
    b. socket.io wird mit diesem HTTP-Server initialisiert, um einen Socket.io-Server zu erstellen.
2. Warten auf Client-Verbindungen:
    a. Der socket.io-Server wartet nun auf eingehende Verbindungen von Clients.
    b. Bei einer neuen Verbindung wird ein 'connection'-Ereignis ausgelöst.
3. Client-Verbindung:
    a. Der Client lädt das Socket.io-Client-Skript, welches eine Verbindung zum Socket.io-Server herstellt.
    b. Dies geschieht durch den Aufruf von io(), der versucht, eine Verbindung zum Host der aktuellen Webseite herzustellen.
4. Ereignishandhabung auf dem Server:
    a. Wenn ein Client eine Verbindung herstellt, wird ein neuer Socket für diese spezifische Verbindung erstellt.
    b. Der Server hört dann auf Ereignisse, die aus diesem Socket kommen.

c. Im Beispiel hört der Server auf das 'chat message'-Ereignis, welches ausgelöst wird, wenn ein Client eine Nachricht sendet.

d. Wenn ein 'chat message'-Ereignis empfangen wird, sendet der Server diese Nachricht an alle verbundenen Clients.

5. Ereignishandhabung auf dem Client:

   a. Der Client hört auf das 'chat message'-Ereignis, welches vom Server gesendet wird.

   b. Wenn der Client dieses Ereignis empfängt, wird die Nachricht zur Liste der Nachrichten hinzugefügt.

6. Nachrichten senden:

   a. Der Client hat ein Eingabefeld und einen Button, um Nachrichten zu senden.

   b. Wenn der Benutzer auf "Senden" klickt, wird das 'chat message'-Ereignis an den Server gesendet.

   c. Der Server empfängt dieses Ereignis und sendet es dann an alle verbundenen Clients, einschließlich des sendenden Clients.

7. Verbindungsabbruch:

   a. Wenn ein Client die Webseite verlässt oder die Verbindung aus einem anderen Grund abbricht, wird ein 'disconnect'-Ereignis auf dem Server ausgelöst.

   b. Der Server kann auf dieses Ereignis reagieren, z. B. um eine Meldung in der Konsole auszugeben, dass ein Benutzer die Verbindung getrennt hat.

Das obige Beispiel zeigt die Grundlagen von Socket.io, wie man Ereignisse sendet und auf sie reagiert. In größeren Anwendungen könnten weitere Funktionen und Ereignistypen hinzugefügt werden, um komplexere Interaktionen und Datenflüsse zu ermöglichen.

10.3 Namespaces

Namespaces in Socket.io dienen der Organisation von Kommunikationskanälen. Sie bieten die Möglichkeit, verschiedene Endpunkte oder Pfade zu erstellen, unter denen Sockets kommunizieren können. Die Trennung in unterschiedliche

Namespaces ist sinnvoll, wenn verschiedene Bereiche oder Komponenten einer Anwendung jeweils eigene Echtzeit-Kommunikationskanäle erfordern.

Ein Namespace repräsentiert eine benannte Endpunkt-URL. Der primäre Namespace in socket.io wird als Root-Namespace bezeichnet und durch \ dargestellt. Auf der Serverseite wird ein Namespace durch die Methode 'of' des Socket.io-Objekts definiert. Jeder Namespace kann individuelle connection- und disconnect-Ereignislistener sowie weitere benutzerdefinierte Ereignisse enthalten. Die Kommunikation innerhalb eines Namespaces ist von anderen Namespaces isoliert. Ein Socket, der mit einem bestimmten Namespace verbunden ist, empfängt standardmäßig keine Nachrichten aus anderen Namespaces, es sei denn, er stellt auch dort eine Verbindung her.

10.3.1 Beispiele zur Nutzung von Namespaces:

Serverseitige Definition von Namespaces:

```
const io = require('socket.io')(3000);

// Haupt-Namespace
io.on('connection', (socket) => {
  console.log('Verbindung zum Haupt-Namespace hergestellt');
});

// Definition eines benutzerdefinierten Namespaces "/chat"
const chat = io.of('/chat');
chat.on('connection', (socket) => {
  console.log('Verbindung zum Chat-Namespace hergestellt');
  socket.on('message', (data) => {
    chat.emit('message', data);
  });
});

// Definition eines weiteren Namespaces "/news"
const news = io.of('/news');
news.on('connection', (socket) => {
  console.log('Verbindung zum News-Namespace hergestellt');
  socket.emit('item', 'Nachricht des Tages');
});
```

Clientseitige Verbindung zu Namespaces:

- Ohne Spezifikation eines Namespaces verbindet sich der Client mit dem Haupt-Namespace

```
const socket = io('http://localhost:3000');
```

- Für den /chat-Namespace:

```
const chatSocket = io('http://localhost:3000/chat');

chatSocket.on('message', (data) => {
  console.log(data);
});
```

- Für den /news-Namespace:

```
const newsSocket = io('http://localhost:3000/news');

newsSocket.on('item', (data) => {
  console.log(data);
});
```

Zusammenfassend ermöglichen Namespaces in Socket.io eine strukturierte und modulare Kommunikation. Sie sind insbesondere in komplexeren Anwendungen von Vorteil, in denen unterschiedliche Kommunikationskanäle für verschiedene Bereiche oder Komponenten erforderlich sind.

10.4 Beispiel für eine ToDo Anwendung mit Node.js, Express.js und socket.io

Diese Anwendung ermöglicht es Benutzern, Aufgaben hinzuzufügen, und alle Benutzer werden in Echtzeit über neue Aufgaben benachrichtigt.

10.4.1 Einrichten des Projekts

Zuerst initialisieren wir ein neues Node.js-Projekt und installieren die benötigten Pakete

```
npm init -y
npm install express socket.io
```

10.4.2 Serverseitiger Code (server.js)

```
const express = require('express');
const http = require('http');
const socketIo = require('socket.io');

const app = express();
const server = http.createServer(app);
const io = socketIo(server);

let todos = [];

io.on('connection', (socket) => {
    console.log('Ein Benutzer ist verbunden');

    // Anfangsdaten an den Client senden
    socket.emit('initialData', todos);

    socket.on('addTodo', (todo) => {
        todos.push(todo);
        // Daten an alle Clients senden
        io.emit('updateTodos', todos);
    });

    socket.on('disconnect', () => {
        console.log('Ein Benutzer hat die Verbindung getrennt
    });
});

app.use(express.static('public'));

const PORT = 3000;
server.listen(PORT, () => {
```

```
    console.log(`Server läuft auf http://localhost:${PORT}`);
});
```

10.4.3 Clientseitiger Code

Erstelle einen public-Ordner mit einer index.html und einer app.js.

public/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initia
    <title>ToDo Anwendung</title>
</head>
<body>
    <input type="text" id="todoInput" placeholder="Neue Aufga
    <button onclick="addTodo()">Hinzufügen</button>

    <ul id="todoList"></ul>

    <script src="/socket.io/socket.io.js"></script>
    <script src="app.js"></script>
</body>
</html>
```

public/app.js

```
const socket = io.connect('http://localhost:3000');

socket.on('initialData', (todos) => {
    renderTodos(todos);
});

socket.on('updateTodos', (todos) => {
    renderTodos(todos);
});
```

```
function addTodo() {
    const todoInput = document.getElementById('todoInput');
    const todo = todoInput.value;
    if (todo) {
        socket.emit('addTodo', todo);
        todoInput.value = '';
    }
}

function renderTodos(todos) {
    const todoList = document.getElementById('todoList');
    todoList.innerHTML = todos.map(todo => `<li>${todo}</li>`
}
```

Nun, wenn Benutzer eine neue Aufgabe hinzufügen, wird diese in Echtzeit an alle verbundenen Clients gesendet.

Um die Anwendung zu starten, führen Sie node server.js aus und öffnen Sie einen Browser unter http://localhost:3000. Sie können mehrere Fenster öffnen, um die Echtzeitfähigkeiten zu testen.

Das ist natürlich eine sehr einfache Implementierung. Es können viele weitere Funktionen hinzugefügt werden, wie z.B. das Löschen von Aufgaben, das Bearbeiten von Aufgaben, Benutzeridentifikation, Datenbankanbindung und so weiter.

11. Einführung in der Verwendung von Datenbanken in Node.js

11.1 Warum braucht man eine Datenbank in einer Node.js Anwendung?

11.1.1 Dauerhafte Datenspeicherung

Ohne eine Datenbank würden alle Daten, die während der Laufzeit einer Anwendung erstellt oder geändert werden, verloren gehen, sobald die Anwendung gestoppt wird. Eine Datenbank ermöglicht es, Daten dauerhaft zu speichern und auch nach einem Neustart der Anwendung oder sogar des gesamten Servers wieder abzurufen.

11.1.2 Große Datenmengen

Eine Datenbank ermöglicht es, große Mengen an Daten effizient zu speichern, abzurufen und zu durchsuchen. Dateisystem-basierte Lösungen könnten bei großen Datenmengen ineffizient oder unpraktisch werden.

11.1.3 Konkurrenter Datenzugriff

Datenbanken sind so konzipiert, dass sie gleichzeitigen Zugriff von mehreren Benutzern oder Systemen unterstützen, ohne Dateninkonsistenzen oder -konflikte zu verursachen.

11.1.4 Such- und Abfragefunktionalität

Moderne Datenbanken bieten fortgeschrittene Abfragefunktionalitäten, die es ermöglichen, komplexe Abfragen auf den Daten auszuführen, um genau die benötigten Informationen effizient abzurufen.

11.1.5 Transaktionen

Viele Datenbanksysteme unterstützen Transaktionen, was bedeutet, dass mehrere Operationen als eine einzelne Einheit betrachtet werden können. Wenn ein Teil der Transaktion fehlschlägt, werden alle Änderungen rückgängig gemacht, was die Datenintegrität sicherstellt.

11.1.6 Skalierbarkeit

Datenbanksysteme, insbesondere NoSQL-Datenbanken, sind oft für horizontale Skalierbarkeit ausgelegt. Dies ermöglicht es Anwendungen, mit wachsenden Datenmengen und Benutzerzahlen umzugehen.

11.1.7 Datenintegrität und -sicherheit

Datenbanken können dazu beitragen, die Integrität der Daten zu gewährleisten, indem sie Datentyp-Beschränkungen, eindeutige Einschränkungen, Fremdschlüsselbeziehungen und andere Regeln einhalten. Darüber hinaus bieten sie Sicherheitsmechanismen wie Benutzerberechtigungen und Verschlüsselung.

11.1.8 Relationen zwischen Daten

In relationalen Datenbanken können komplexe Beziehungen zwischen verschiedenen Datensätzen oder Tabellen definiert werden, die das Abfragen und Organisieren der Daten erleichtern.

11.1.9 Backup und Wiederherstellung

Datenbankmanagementsysteme bieten in der Regel Möglichkeiten zum Sichern der Daten und zum Wiederherstellen im Falle eines Datenverlustes.

11.1.10 Middleware-Integration

Viele moderne Web-Frameworks und Middleware, insbesondere in der Node.js-Welt (z. B. Express mit Mongoose für MongoDB oder Sequelize für SQL-Datenbanken), bieten eine nahtlose Integration mit Datenbanken. Dies erleichtert den Entwicklern das Erstellen und Verwalten von Datenbankoperationen.

Zusammenfassend ermöglicht eine Datenbank in einer Node.js-Anwendung, Daten sicher, effizient und dauerhaft zu speichern und zu verwalten, und bietet gleichzeitig eine Reihe von Werkzeugen und Funktionen, um Daten effektiv abzurufen und zu manipulieren.

11.2 MongoDB

MongoDB ist eine dokumentenorientierte Datenbank, was bedeutet, dass sie Informationen in Form von "Dokumenten" speichert, die in JSON-ähnlichen Strukturen (genannt BSON) organisiert sind. Diese Eigenschaft macht MongoDB besonders geeignet für Anwendungen, die in JavaScript geschrieben sind, da JSON (JavaScript Object Notation) nativ in der Sprache verwendet wird.

11.2.1 MongoDB-Installation

Bevor MongoDB mit Node.js verwendet werden kann, muss die Datenbank selbst installiert sein. Dies kann durch den Download von der offiziellen MongoDB-Website https://www.mongodb.com/try/download/community oder über einen Paketmanager erfolgen.

11.2.2 Node.js-Anbindung an MongoDB

Es gibt verschiedene Möglichkeiten, Node.js mit MongoDB zu verbinden. Eine der gebräuchlichsten Methoden ist die Verwendung des MongoDB Native Drivers oder des Mongoose ODM (Object Document Mapper).

11.2.2.1 MongoDB Native Driver

Dies ist der offizielle Treiber von MongoDB für Node.js. Er ermöglicht direkte CRUD-Operationen (Create, Read, Update, Delete) auf der Datenbank.

▼ Quellcodeverzeichnis

3.1 Prüfen der Node-Version

3.2 Prüfen der npm-Version

3.3 Installation von lodash

3.4 Installation von nodemon

3.5 Erstellen einer package.json-Datei

3.6 Andere Projekte klonen und Abhängigkeiten installieren

3.7 Auflisten der Node-Versionen

3.8 Installation einer bestimmten Node-Version

3.9 Wechseln zu einer bestimmten Node-Version

3.10 Deinstallieren einer bestimmten Node-Version

3.11 Aktuelle verwendete Node-Version anzeigen

4.1 Bsp. zu exports

4.2 Bsp. zu module.exports

4.3 Bsp. Einfache Funktionen exportieren mathModule.js

4.4 Bsp. Einfache Funktionen exportieren app.js

4.5 Bsp. Ein gesamtes Objekt mit module.exports exportieren apersonModule.js

4.6 Bsp. Ein gesamtes Objekt mit module.exports exportieren app.js

4.7 Bsp. Mischen von exports und module.exports mixedModules.js

4.8 Bsp. Mischen von exports und module.exports app.js

4.9 Beispielanwendung index.html

4.10 Beispielanwendung styles.css

4.11 Beispielanwendung todoModule.js

4.12 Beispielanwendung app.js

5.1 Asynchron mit Callback / Lesen

5.2 Asynchron mit Callback / Schreiben

5.3 Asynchron mit Callback / Bearbeiten

5.4 Asynchron mit Promises / Lesen

5.5 Asynchron mit Promises / Schreiben

5.6 Asynchron mit Promises / Bearbeiten

5.7 Synchron / Lesen

5.8 Synchron / Schreiben

5.9 Synchron / Bearbeiten

6.1 Kommandozeilenargumente argv.js

6.2 Kommandozeilenargumente summe.js

6.3 Kommandozeilenargumente summeYargs.js

8.1 Bsp. Grundlagen der Asynchronität

8.2 Bsp. Callbacks

8.3 Bsp. Promises

8.4 Bsp. async/await

9.1 Bsp. Express.js Grundlage

9.2 Bsp. Routing mit Express

9.3 Bsp. Middleware 1

9.4 Bsp. Middleware 2

9.5 Bsp. express.static

10.1 Bsp. mit Express und socket.io / server.js

10.2 Bsp. mit Express und socket.io / index.html

10.3 Bsp. zur Nutzung von Namespaces / Serverseitige Definition

10.4 Bsp. zur Nutzung von Namespaces / Clientseitige Verbindung zu Namespaces

10.5 Bsp. zur Nutzung von Namespaces / chat-Namespace

10.6 Bsp. zur Nutzung von Namespaces / news-Namespace

10.7 Bsp. für eine ToDo Anwendung mit Node.js, Express.js und socket.io / Einrichten des Projekts

10.8 Bsp. für eine ToDo Anwendung mit Node.js, Express.js und socket.io / server.js

10.9 Bsp. für eine ToDo Anwendung mit Node.js, Express.js und socket.io / index.html

10.10 Bsp. für eine ToDo Anwendung mit Node.js, Express.js und socket.io / app.js