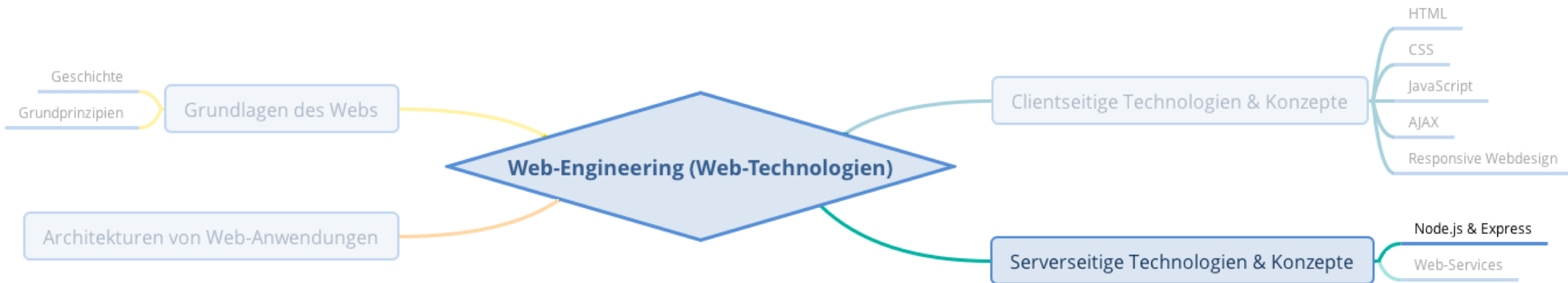




# **WEB- TECHNOLOGIEN**

**SERVERSEITIGE TECHNOLOGIEN:  
NODE.JS UND EXPRESS**

# THEMEN DER VERANSTALTUNG



# LERNZIELE

1. Ausgewählte Konzepte und Basistechnologien zur serverseitigen Web-Entwicklung mit Node.js und Express kennen und anwenden können
2. Serverseitige Architekturkonzepte verstehen



# ERINNERUNG: SERVERSEITIGE TECHNOLOGIEN

- **Node und Express** sind Beispiele für **serverseitige** Technologien
- Weitere Beispiele:
  - Common Gateway Interface ([CGI](#))
  - [PHP](#)
  - [Java EE](#)
  - [Ruby on Rails](#)
  - [ASP.NET](#)
  - [Django](#)


# ERINNERUNG: JAVASCRIPT

- JavaScript ist eine *im Browser* integrierte Programmiersprache
- Die Ausführung von JavaScript erfolgt über eine **JavaScript-Engine**
- Beispiele von Engines: [V8](#) , [SpiderMonkey](#) , [Rhino](#)



- *Serverseitige* Plattform für (Netzwerk)anwendungen (z.B. Web-Server, Netzwerktools, Kommandozeilentools, Web-Frameworks)
- Open Source ([MIT License](#) )  
→ <https://github.com/nodejs/node> 
- Verwaltet von der Node.js Foundation



- Basiert auf Google's [V8](#)  Engine
- Entwicklung mit Node.js erfolgt in JavaScript
- Damit ist es möglich, JavaScript sowohl client- als auch serverseitig einzusetzen (*Full Stack JavaScript*)



# NODE.JS: KURZE GESCHICHTE


- 2009 von Ryan Dahl erfunden und auf der [JSConf präsentiert](#) 
- Zunächst maßgeblich gesponsert von der Firma Joyent
- 2011: Node.js bekommt native Unterstützung für Windows
- 2012 zieht sich Ryan Dahl aus der Node.js-Entwicklung zurück → Joyent stellt nun die Maintainer



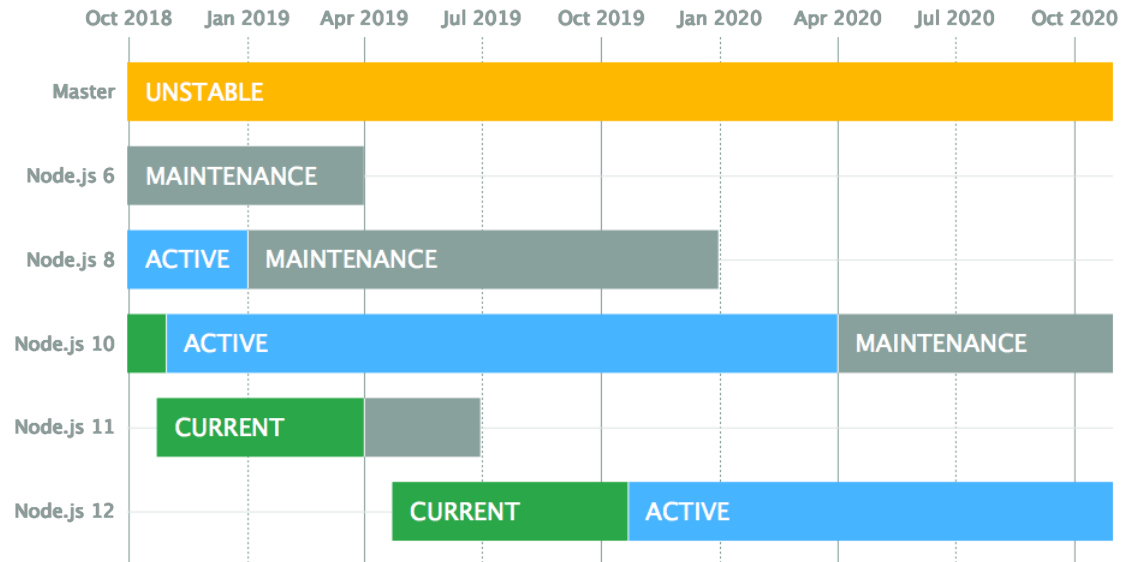
Bild: [David Calhoun](#)  
via [Wikimedia Commons](#) / CC BY 2.0



# NODE.JS: KURZE GESCHICHTE (2)

- Viele EntwicklerInnen waren unzufrieden mit der Führung des Projektes durch Joyent
- 2014 führt dies zum Bruch in der Community: Node.js wird unter dem Namen io.js geforked
- 2015 wird die unabhängige Node.js Foundation gegründet, Node.js und io.js werden wieder zusammengeführt
- Koordination erfolgt nun über das Technical Steering Committee (TSC)

# NODE.JS RELEASES



- Major Releases alle 6 Monate (gerade Versionen im April, ungerade im Oktober)
- Erscheint eine neue ungerade Version, so wird die vorherige gerade Version zur LTS-Version (*Long Term Support*)








# VERBREITUNG

→ [Stack Overflow Developer Survey 2018](#) 

Node.js wird u.A. eingesetzt von:

- Netflix
- LinkedIn
- PayPal
- Ebay

# NODE.JS ÖKOSYSTEM

- ~800.000 Bibliotheken/Pakete für Node.js (vgl. [npmjs.com](https://npmjs.com)  )
- Node.js bildet u.A. die Basis für:
  - Frameworks, z.B. [Express](#)  , [Meteor](#)  , [Koa.js](#)  , [Socket.io](#) 
  - Entwicklungswerkzeuge wie z.B. [Webpack](#)  , [Gulp](#) 
- Sehr gute Unterstützung in Entwicklungsumgebungen



## Zentrale Eigenschaften von Node.js:

- Asynchrone (nicht-blockierende) Ein- und Ausgabe  
→ Geeignet für hohe Anzahl von Zugriffen und hohen Durchsatz (z.B. Streaming)
- Ereignisgetrieben → Ressourcenschonend, optimale Auslastung
- Modularer Aufbau → Skalierbarkeit, gute Erweiterbarkeit

! Wir werden uns im Folgenden anschauen, was genau diese Eigenschaften bedeuten.

# JAVASCRIPT MIT NODE.JS AUSFÜHREN

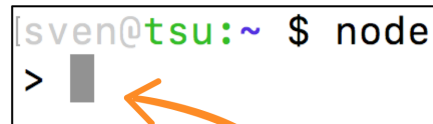
Node.js bietet zwei Modi, um JavaScript auszuführen:

1. interaktiv per REPL (Read-Eval-Print-Loop)
2. Ausführen von JavaScript-Dateien

# READ-EVAL-PRINT-LOOP (REPL)

## Interaktiver Modus auf Kommandozeile:

1. *Read*: vom Benutzer eingebene Kommandos werden eingelesen
2. *Eval*: die Kommandos werden evaluiert (ausgeführt)
3. *Print*: Ergebnisse werden auf der Kommandozeile ausgegeben
4. *Loop*: Starte wieder bei 1.



```
[sven@tsu:~ $ node  
> █
```

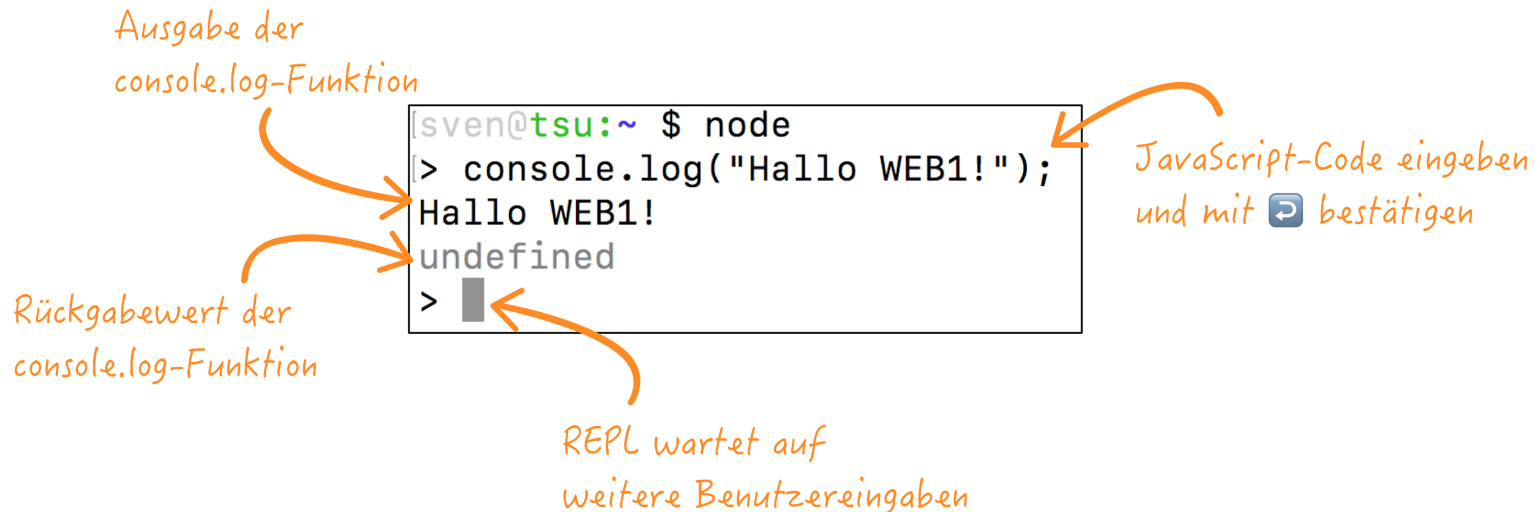
REPL durch Eingabe des  
Befehls „node“ starten

REPL wartet auf  
Benutzereingaben

# READ-EVAL-PRINT-LOOP (REPL)

## Interaktiver Modus auf Kommandozeile:

1. *Read*: vom Benutzer eingegebene Kommandos werden eingelesen
2. *Eval*: die Kommandos werden evaluiert (ausgeführt)
3. *Print*: Ergebnisse werden auf der Kommandozeile ausgegeben
4. *Loop*: Starte wieder bei 1.





# READ-EVAL-PRINT-LOOP (REPL)

-  Gut geeignet zum schnellen Experimentieren
-  Nicht geeignet für "echte" Applikationen, da der eingegebene Code nicht persistent ist

# AUSFÜHREN VON JAVASCRIPT-DATEIEN

Üblicher Weg für Applikationen → Code ist in Dateien persistiert

Inhalt der Datei `app.js`:

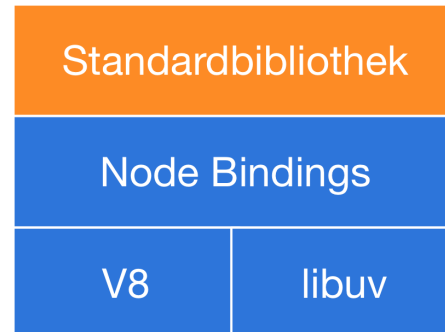
```
console.log( "Hallo WEB1!" );
```

```
[sven@tsu:~ $ node app.js  
Hallo WEB1! _
```

Name der auszuführenden  
Datei beim node-Kommando  
angeben

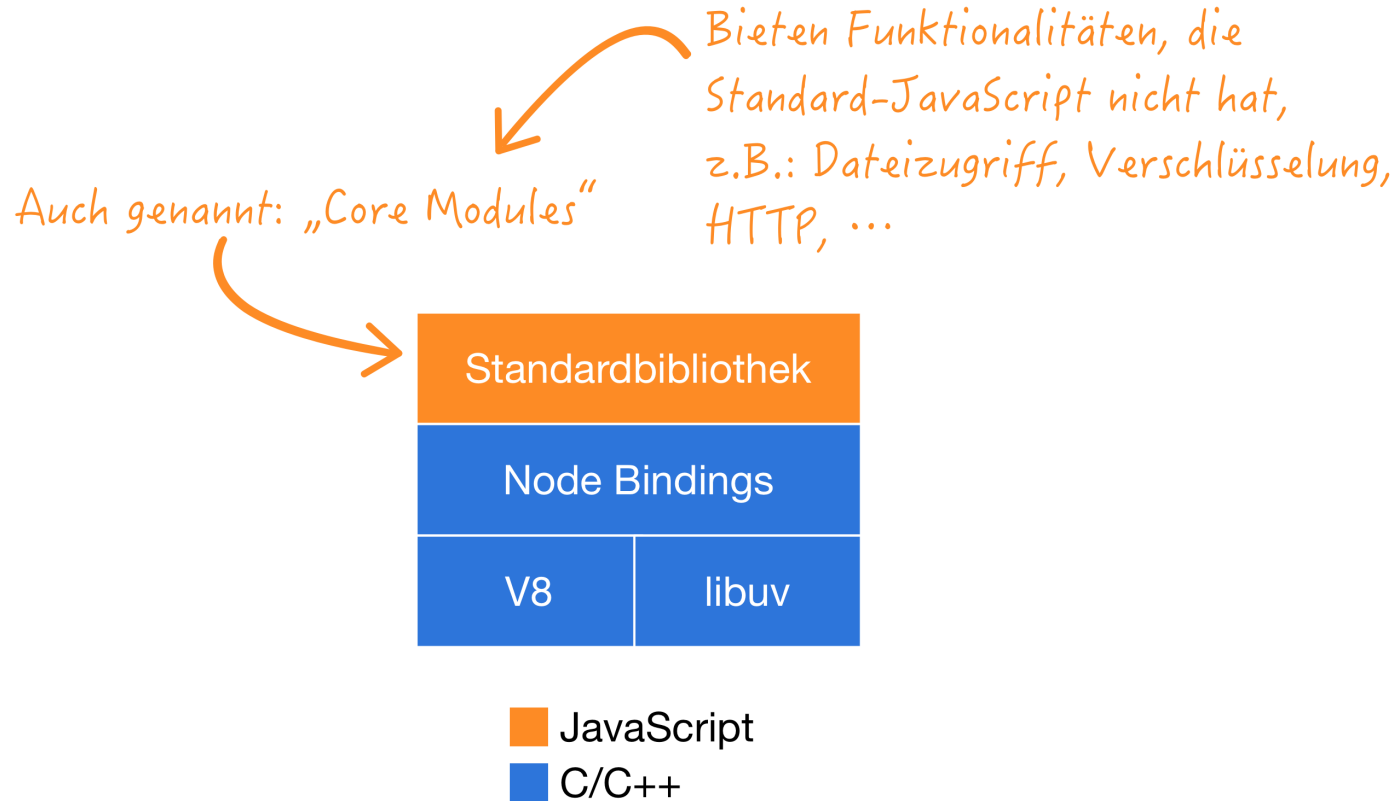
Ausgabe des Codes

# ARCHITEKTUR VON NODE.JS

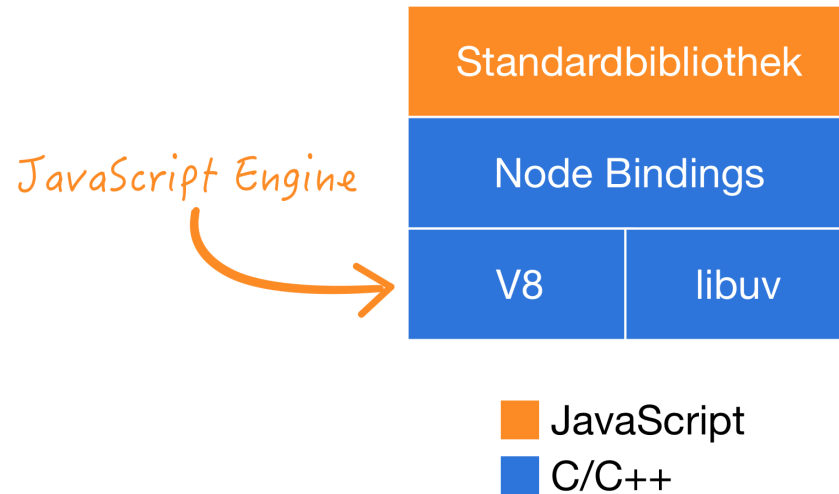


■ JavaScript  
■ C/C++

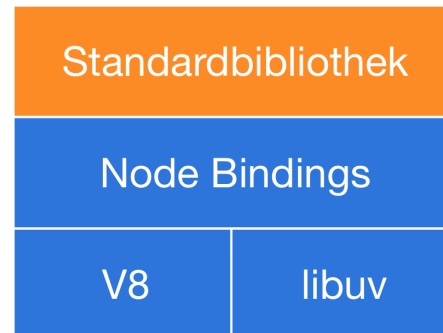
# ARCHITEKTUR VON NODE.JS



# ARCHITEKTUR VON NODE.JS



# ARCHITEKTUR VON NODE.JS

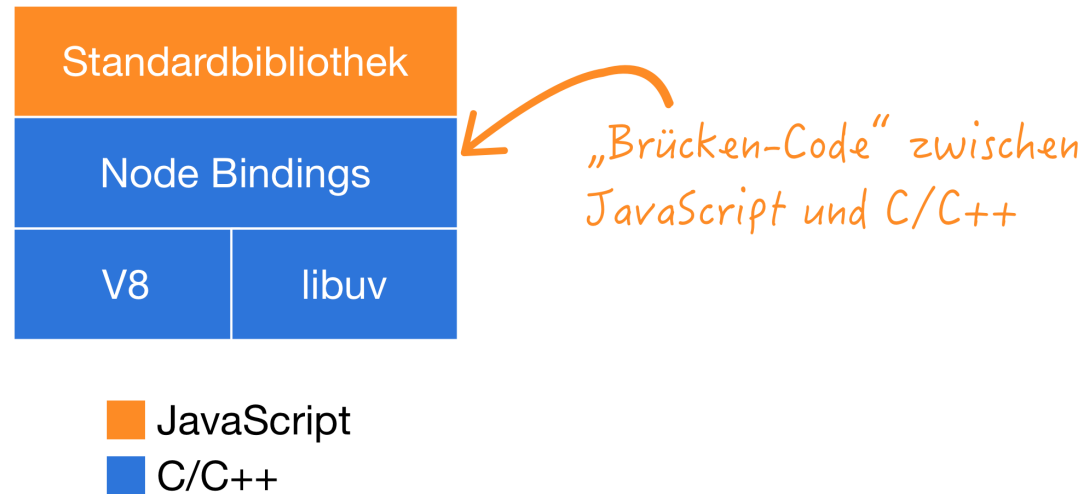


■ JavaScript  
■ C/C++

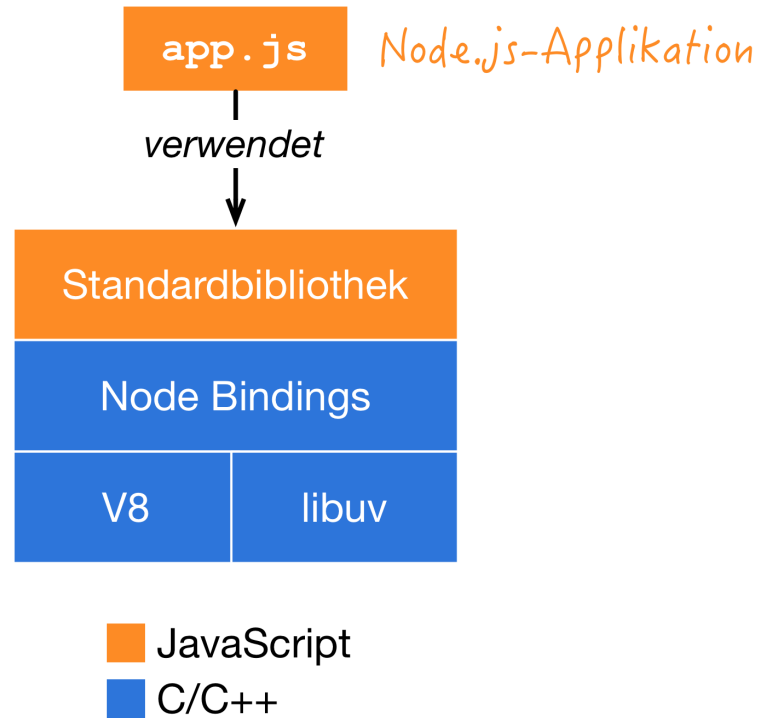
*Bibliothek, die eine Abstraktionsschicht  
für Betriebssystemfunktionen bietet*

*z.B. Dateizugriff, TCP-/UDP-Sockets, DNS, Threads*

# ARCHITEKTUR VON NODE.JS




# ARCHITEKTUR VON NODE.JS





# STANDBIBLIOTHEK

- Auch: Kernmodule (*Core Modules*)
- Bieten Standardfunktionalitäten für Node.js-Applikationen
- Dokumentation: <https://nodejs.org/api/> 
- Um die Funktionalitäten eines Kernmoduls zu verwenden, muss es über die (global verfügbare) `require`-Funktion eingebunden werden:

```
// Über die 'require'-Funktion wird das Modul mit dem Namen  
// $MODULNAME eingebunden. Die Funktionen des Moduls  
// können dann über die Variable "einModul" verwendet werden  
const einModul = require("$MODULNAME");
```

# KERNMODULE: BEISPIELE

Modulname	Zweck
buffer	Umgang mit Binärdaten
crypto	Verschlüsselung (OpenSSL)
dns	Namensauflösung
events	Ereignisbehandlung
fs	Zugriff auf das Dateisystem
http	Erzeugung von HTTP-Clients und -Servern
timers	Zeitabhängige Funktionen
url	Erzeugung und Parsen von URLs

# KERNMODUL "fs"

- Das Modul "fs" [↗](#) ermöglicht lesenden und schreibenden Zugriff auf das Dateisystem (Dateien und Verzeichnisse)
- Die meisten Funktionen des Moduls gibt es in einer *synchronen* und einer *asynchronen* Variante
- Beispiele für Funktionen des "fs"-Moduls:

Funktion	Zweck
<a href="#">appendFile</a> <a href="#">↗</a> / <a href="#">appendFileSync</a> <a href="#">↗</a>	Daten an eine Datei anhängen
<a href="#">chmod</a> <a href="#">↗</a> / <a href="#">chmodSync</a> <a href="#">↗</a>	Dateiberechtigungen ändern
<a href="#">readFile</a> <a href="#">↗</a> / <a href="#">readFileSync</a> <a href="#">↗</a>	Gesamten Inhalt einer Datei einlesen
<a href="#">stat</a> <a href="#">↗</a> / <a href="#">statSync</a> <a href="#">↗</a>	Informationen zu einer Datei lesen (z.B. Größe, Zeitpunkt der letzten Änderung)
<a href="#">writeFile</a> <a href="#">↗</a> / <a href="#">writeFileSync</a> <a href="#">↗</a>	Inhalt in eine Datei schreiben



## Zentrale Eigenschaften von Node.js:

- Asynchrone (nicht-blockierende) Ein- und Ausgabe
- Ereignisgetrieben
- Modularer Aufbau

# SYNCHRONE IO<sup>\*</sup>

- Bei Ausführung einer *synchronen* IO-Operation wartet der Programmfluss, bis die Operation beendet ist (sequentielle Ausführung)
- Das Programm kann in dieser Zeit nichts anderes tun → die Operation ist blockierend (*blocking*)

<sup>\*</sup> IO = Input und Output (Ein- und Ausgabe, z.B. Dateisystem, Netzwerk etc.)

# BEISPIEL: SYNCHRONE IO MIT "fs"

Die Datei `name.txt` enthält die Zeichenkette "WEB1".

Datei `app.js`:

```
// fs-Modul einbinden
const fs = require("fs");

console.log("Hallo ");

// Inhalte der Datei "name.txt" synchron einlesen
const name = fs.readFileSync("name.txt", "utf-8");
// Diese Zeile wird erst ausgeführt, wenn die "readFileSync"-Funktion fertig ist
console.log(name);

console.log("!");
```

Ausgabe:

```
[sven@tsu:~ $ node app.js
Hallo
WEB1
!
```

# BEISPIEL: SYNCHRONE IO MIT "fs"

Erklärung: Das Programm wird sequentiell ausgeführt.

Programmcode:

```
const fs = require("fs"); // 1

console.log("Hallo ");

const name =
  fs.readFileSync("name.txt", "utf-8");
console.log(name);

console.log("!");
```

Kontrollfluss:

1

Ausgabe:

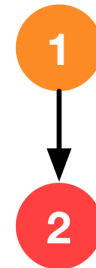
# BEISPIEL: SYNCHRONE IO MIT "fs"

Erklärung: Das Programm wird sequentiell ausgeführt:

Programmcode:

```
const fs = require("fs");  
  
console.log("Hallo "); // 2  
  
const name =  
    fs.readFileSync("name.txt", "utf-8");  
console.log(name);  
  
console.log("!");
```

Kontrollfluss:



Ausgabe:

Hallo



# BEISPIEL: SYNCHRONE IO MIT "fs"

Erklärung: Das Programm wird sequentiell ausgeführt:

Programmcode:

```
const fs = require("fs");  
  
console.log("Hallo ");  
  
const name =  
    fs.readFileSync("name.txt", "utf-8"); // 3  
console.log(name);  
  
console.log("!");
```

Ausgabe:  
Hallo

Kontrollfluss:



# BEISPIEL: SYNCHRONE IO MIT "fs"

Erklärung: Das Programm wird sequentiell ausgeführt:

Programmcode:

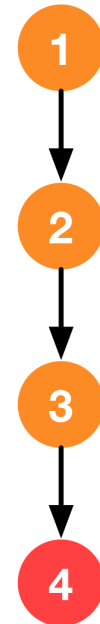
```
const fs = require("fs");  
  
console.log("Hallo ");  
  
const name =  
    fs.readFileSync("name.txt", "utf-8");  
console.log(name); // 4  
  
console.log("!");
```

Ausgabe:

Hallo

WEB1

Kontrollfluss:



# BEISPIEL: SYNCHRONE IO MIT "fs"

Erklärung: Das Programm wird sequentiell ausgeführt:

Programmcode:

```
const fs = require("fs");  
  
console.log("Hallo ");  
  
const name =  
    fs.readFileSync("name.txt", "utf-8");  
console.log(name);  
  
console.log("!"); // 5
```

Ausgabe:

Hallo

WEB1

!

Kontrollfluss:



# ASYNCHRONE IO

- Eine *asynchrone* IO-Operation wird *nebenläufig* ausgeführt
- Der eigentliche Programmfluss wartet nicht auf Beendigung der Operation (nachfolgender Code wird *direkt* ausgeführt)
- Die Operation ist nicht blockierend (*non-blocking*)

# BEISPIEL: ASYNCHRONE IO MIT "fs"

Die Datei `name.txt` enthält die Zeichenkette "WEB1".

Datei `app.js`:

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

console.log("Hallo ");

// Inhalte der Datei "name.txt" asynchron einlesen --> es wird nicht gewartet, bis
// das Einlesen beendet ist,...
fs.readFile("name.txt", "utf-8", function(err, name) {
    console.log(name);
});

// ...daher wird diese Zeile direkt ausgeführt
console.log("!");
```

Ausgabe:

```
[sven@tsu:~ $ node app.js
Hallo
!
WEB1
```

# BEISPIEL: ASYNCHRONE IO MIT "fs"

Erklärung: Das Lesen der Datei wird asynchron ausgeführt:

Programmcode:

```
const fs = require("fs"); // 1

console.log("Hallo ");

fs.readFile("name.txt", "utf-8", function(err, name) {
  console.log(name);
});

console.log("!");
```

Kontrollfluss  
(schematisch):

1

Ausgabe:

# BEISPIEL: ASYNCHRONE IO MIT "fs"

Erklärung: Das Lesen der Datei wird asynchron ausgeführt:

Programmcode:

```
const fs = require("fs");  
  
console.log("Hallo "); // 2  
  
fs.readFile("name.txt", "utf-8", function(err, name) {  
    console.log(name);  
});  
  
console.log("!");
```

Ausgabe:  
Hallo

Kontrollfluss  
(schematisch):



# BEISPIEL: ASYNCHRONE IO MIT "fs"

Erklärung: Das Lesen der Datei wird asynchron ausgeführt:

Programmcode:

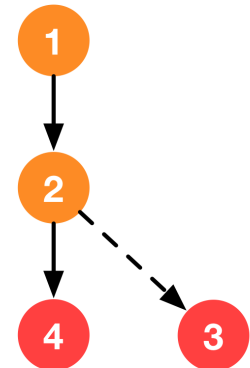
```
const fs = require("fs");  
  
console.log("Hallo ");  
  
fs.readFile("name.txt", "utf-8", function(err, name) { // 3  
  console.log(name);  
});  
  
console.log("!"); // 4
```

Ausgabe:

Hallo

!

Kontrollfluss  
(schematisch):





# BEISPIEL: ASYNCHRONE IO MIT "fs"

Erklärung: Das Lesen der Datei wird asynchron ausgeführt:

Programmcode:

```
const fs = require("fs");  
  
console.log("Hallo ");  
  
fs.readFile("name.txt", "utf-8", function(err, name) {  
    console.log(name); // 5  
});  
  
console.log("!");
```

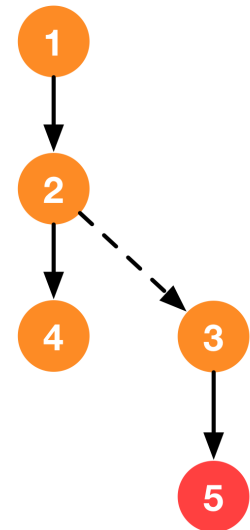
Ausgabe:

Hallo

!

WEB1

Kontrollfluss  
(schematisch):



# CALLBACKS

Wir betrachten folgenden Teil des Beispiels genauer:

```
fs.readFile("name.txt", "utf-8", function(err, name) {  
    console.log(name);  
});
```

- Über `readFile` lesen wir den Inhalt einer Datei asynchron ein
- Den gelesenen Inhalt wollen wir weiterverarbeiten (hier: Ausgabe auf die Konsole)

⚡ Problem: Wir wissen nicht, wann das Einlesen der Datei fertig ist - Wie bekommen wir mit, wann wir mit der Weiterverarbeitung starten können?

# CALLBACKS

Wir betrachten folgenden Teil des Beispiels genauer:

```
fs.readFile("name.txt", "utf-8", function(err, name) {  
    console.log(name);  
});
```

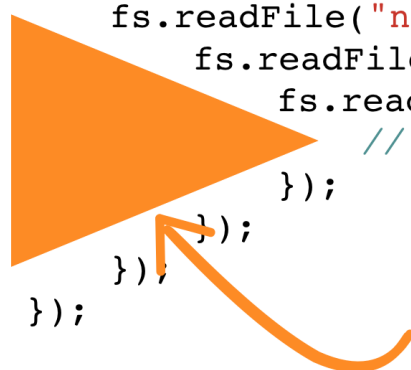
- Lösung: Wir geben `readFile` eine Funktion als Argument mit
- Diese Funktion wird von `readFile` aufgerufen, sobald das Einlesen der Datei beendet ist
- Eine solche Funktion wird *Callback*-Funktion genannt

# CALLBACKS (2)

- Callback-Funktionen sind ein sehr verbreitetes Muster zur asynchronen Programmierung in JavaScript
- Auch bei der Programmierung mit Node.js wird viel mit Callback-Funktionen gearbeitet

❗ Vorsicht: Zu tief verschachtelte Callbacks können zu sehr unleserlichem Code führen:

```
fs.readFile("name1.txt", "utf-8", function(err, name1) {  
  fs.readFile("name2.txt", "utf-8", function(err, name2) {  
    fs.readFile("name3.txt", "utf-8", function(err, name3) {  
      fs.readFile("name4.txt", "utf-8", function(err, name4) {  
        // [...]  
      });  
    });  
  });  
});
```




„Pyramid of Doom“, oder: „Callback Hell“

# SYNCHRONE VS. ASYNCHRONE IO

*Vor- und Nachteile?*

- **Verständlichkeit:** Synchron programmierter Code ist teilweise leichter zu verstehen als asynchron programmierter Code
- **Performanz:** Asynchrone IO nutzt die vorhandenen Ressourcen besser (diese werden nicht durch Warten blockiert) und erlaubt einen höheren Durchsatz

-  Das Performanz-Argument spielt insbesondere bei serverseitigen (Web-)Anwendungen eine Rolle, die sehr IO-lastig sind
- z.B. hohe Anzahl paralleler Anfragen, schnelle Auslieferung von Dateien, Video-Streaming

# KERNMODULE: BEISPIELE

Modulname	Zweck
buffer	Umgang mit Binärdaten
crypto	Verschlüsselung (OpenSSL)
dns	Namensauflösung
events	Ereignisbehandlung
fs	Zugriff auf das Dateisystem
http	Erzeugung von HTTP-Clients und -Servern
timers	Zeitabhängige Funktionen
url	Erzeugung und Parsen von URLs

# KERNMODUL "http"

- Das Modul "http" [↗](#) ermöglicht das Erstellen von HTTP-Servern und -Clients
- Für die Unterstützung von [HTTPS](#) [↗](#) und [HTTP/2](#) [↗](#) gibt es separate Kernmodule
- Nach Einbinden des Moduls stehen u.A. folgende zentrale Funktionen zur Verfügung:

Funktion	Zweck
<a href="#">createServer</a> <a href="#">↗</a>	Erstellt einen neuen HTTP-Server in Form eines <a href="#">Server</a> <a href="#">↗</a> -Objekts.
<a href="#">request</a> <a href="#">↗</a>	Erlaubt das Senden von HTTP-Anfragen (HTTP-Client).

# Server-OBJEKT

- Über die Funktion `createServer` des "http"-Moduls kann ein neues `Server` [↗](#)-Objekt erstellt werden:

```
// http-Modul einbinden
const http = require("http");
// Neues Server-Objekt erstellen
const server = http.createServer();
```

- Dieses `Server`-Objekt hat zunächst keine Funktionalität und muss daher weiter konfiguriert werden



# Server-OBJEKT: `listen`

- Mit Hilfe der `listen` [↗](#)-Funktion kann konfiguriert werden, wie der Server für Clients erreichbar sein soll
- Dazu bietet `listen` u.A. folgende Parameter:

Parameter	Zweck
<code>port</code>	<ul style="list-style-type: none"><li>▪ TCP-Port für den Server</li><li>▪ Darf nicht schon belegt sein</li><li>▪ Bei keiner Angabe wird ein beliebiger freier Port zugewiesen</li></ul>
<code>host</code>	<ul style="list-style-type: none"><li>▪ IP-Adresse für den Server</li><li>▪ Bei keiner Angabe wird <code>0.0.0.0</code> (IPv4) bzw. <code>::</code> (IPv6) zugewiesen (d.h. lokal erreichbar z.B. per <code>localhost</code>)</li></ul>
<code>callback</code>	<ul style="list-style-type: none"><li>▪ Callback-Funktion</li><li>▪ Wird ausgeführt, sobald Port und IP-Adresse zugewiesen wurden</li></ul>

- Man spricht hier auch vom *Binden* des Servers an einen Port und eine IP-Adresse

# BEISPIEL: MINIMALER HTTP-SERVER

Datei `webServer.js`:

```
const http = require("http");
const server = http.createServer();
// Server soll an Port 8042 sowie an lokale IP-Adressen
// (da keine Angabe) gebunden werden
server.listen(8042, function() {
    // Callback-Funktion, Ausgabe erfolgt nach erfolgreicher
    // Bindung des Servers
    console.log("Ich lausche nun auf http://localhost:8042");
});
```

Ausgabe:

```
sven@tsu:~ $ node webServer.js
Ich lausche nun auf http://localhost:8042
```

! Der Server lauscht zwar auf Verbindungen - er tut aber bisher noch nichts!

# REQUEST-LISTENER

- Mit einem *Request-Listener* können wir bestimmen, wie der Server auf eintreffende Anfragen reagieren soll
- Der *Request-Listener* ist eine Funktion, die wir beim Aufruf von `createServer` als Argument übergeben
- Diese Funktion wird immer aufgerufen, sobald eine Anfrage beim Server eintrifft
- Die Funktion hat zwei Parameter:


Parameter	Zweck
<code>request</code>	Ein Request-Objekt, das die eingetroffene Anfrage repräsentiert.
<code>response</code>	Ein Response-Objekt, das die Antwort des Servers repräsentiert.

# BEISPIEL: REQUEST-LISTENER

```
const http = require("http");  
// Request-Listener-Funktion wird als Argument bei  
// "createServer" gesetzt:  
const server = http.createServer(function(request, response) {  
    // [...]  
});  
  
server.listen(8042, function() {  
    console.log("Ich lausche nun auf http://localhost:8042");  
});
```




# Response-OBJEKT

- Durch Modifikation des Response-Objekts können wir die Antwort des Servers auf eine Anfrage bestimmen
- Das Response-Objekt bietet zu diesem Zweck u.A. folgende Funktionen:

Funktion	Zweck
<code>writeHead</code> 	Schreiben des Statuscodes und der Header der Antwort über folgende Parameter: <ol style="list-style-type: none"><li>1. <code>statusCode</code>: Statuscode als Zahl (z.B. 404)</li><li>2. <code>headers</code>: Objekt mit den Headern (z.B. <code>{ "Content-Type": "text/plain; charset=utf-8" }</code>)</li></ol>

# Response-OBJEKT (2)

- Fortsetzung: Funktionen des Response-Objekts:

Funktion	Zweck
<code>write</code> 	<ul style="list-style-type: none"><li>▪ Unterstützt eine Zeichenkette oder Binärdaten (als <code>Buffer</code>  - Objekt) als Argument</li><li>▪ Der übergebene Wert wird in den Body der Antwort geschrieben und als Teilnachricht (<i>Chunk</i>) zum Client geschickt.</li><li>▪ Kann zum Senden mehrerer Chunks mehrfach aufgerufen werden</li></ul>
<code>end</code> 	<ul style="list-style-type: none"><li>▪ Signalisiert dem Client, dass die Antwort komplett gesendet wurde</li><li>▪ Optional kann wie bei <code>write</code> noch ein letzter Chunk als Argument angegeben werden.</li></ul>

# BEISPIEL: EINFACHE HTML-ANTWORT

```
const http = require("http");

const server = http.createServer(function(request, response) {
  // Statuscode und Header schreiben
  response.writeHead(200, { "content-type": "text/html; charset=utf-8" });

  // HTML-Inhalt mittels Template-Literal erstellen
  const html = `<!DOCTYPE html>
    <html>
      <head>
        <title>Hallo WEB1</title>
        <meta charset="utf-8">
      </head>
      <body>
        <h1>Hallo WEB1!</h1>
      </body>
    </html>`;

  // Inhalt in einem einzigen Chunk schicken
  response.end(html);
});

server.listen(8042, function() {
  console.log("Ich lausche nun auf http://localhost:8042");
});
```