

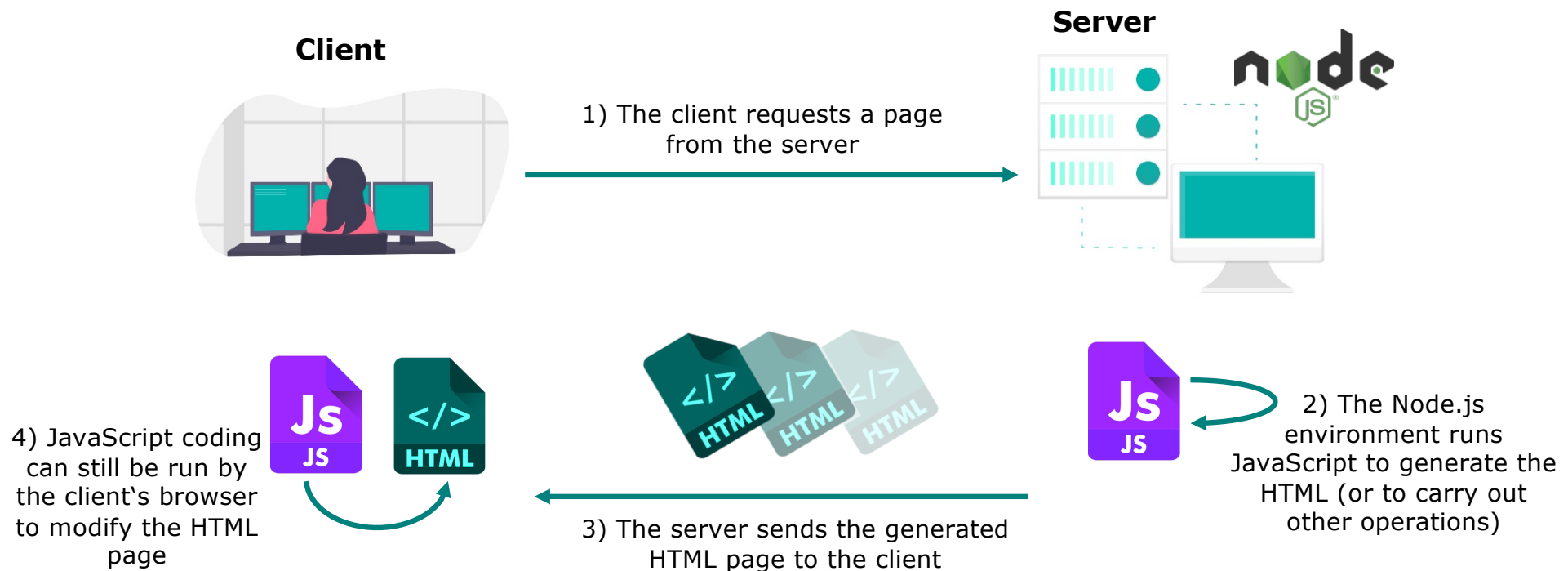


Node.js

Grundlagen, npm, Express.js

<https://nodejs.org/>

- Software, um JavaScript (**serverseitig**) auszuführen
 - nur eine Sprache zur Entwicklung notwendig
 - > gleiche Syntax, gleiche Mechanismen
 - > agiler Einsatz möglich
 - teilweise gleiche Bibliotheken für Server und Client!



- 2009: erste Veröffentlichung durch Ryan Dahl, gesponsert von Fa. Joyent
- 2010: Release von **express** (Web Application Server Framework)
- 2011: Einführung **npm** (Paketmanager für Javascript)
- 2015: Gründung Node.js Foundation
- 2020: npm wird von GitHub (Microsoft) gekauft
- 2020: aktuelle Version 12.16.2 (LTS)

- Plattform, um skalierbare Netzwerkanwendungen zu erstellen
 - Streaming
 - JSON-basierte REST-Dienste
 - Single-Page-Anwendungen
- basiert auf Googles JavaScript-Engine V8
- **eventbasiert, nicht-blockierend** (Event-Loop!!!)
- Opensource (MIT-Lizenz)
- <https://github.com/nodejs/node>

Ihr Code ist der König

- Könige arbeiten nicht, dafür haben sie Heerscharen von Dienern
- Morgens, wenn der König erwacht fragt ihn der Butler, was alles zu erledigen sei
 - > Der Butler ist, wenn man so möchte, die JavaScript-Engine node.js
 - > Die anderen Diener, die die Anweisungen des Butlers erledigen, sind andere Threads der node.js-Implementierung
- Der König gibt eine Liste von Aufträgen an den Butler, der diese auf die vielen anderen Diener verteilt (z.B. die Diener des Betriebssystems)
- Nun ist der König frei in seinem Handeln, bis einer der Diener seine Aufgabe erledigt hat und über sich zum Butler begibt und an den König berichtet
- Natürlich kann immer nur ein Diener nach dem anderen den König berichten
- Daher stellen sich alle Diener die etwas zu berichten haben in einer Schlange an
- Manchmal entscheidet der König, dass der Diener noch mehr zu erledigen hat und gibt ihm über den Butler neue Aufgaben

Es kann nur ein Diener nach dem anderen berichten!

```
var fs = require('fs'), sys = require('sys');

fs.readFile('eineDatei.txt', function(report) {
  sys.puts("Schau an was ich alle habe: " + report);
});

fs.writeFile('nocheineDatei.txt', 'etwas zu schreiben', function() {
  sys.puts("Und doch so viel zu tun");
});
```

Bei der Ausführung soll eine Datei gelesen und eine geschrieben werden. Beide Aufträge werden direkt erteilt. Dabei wird eine anonyme Funktion mitgeliefert, die als callback-Funktion dient. Diese wird aufgerufen, wenn die Aufgabe erledigt wurde.

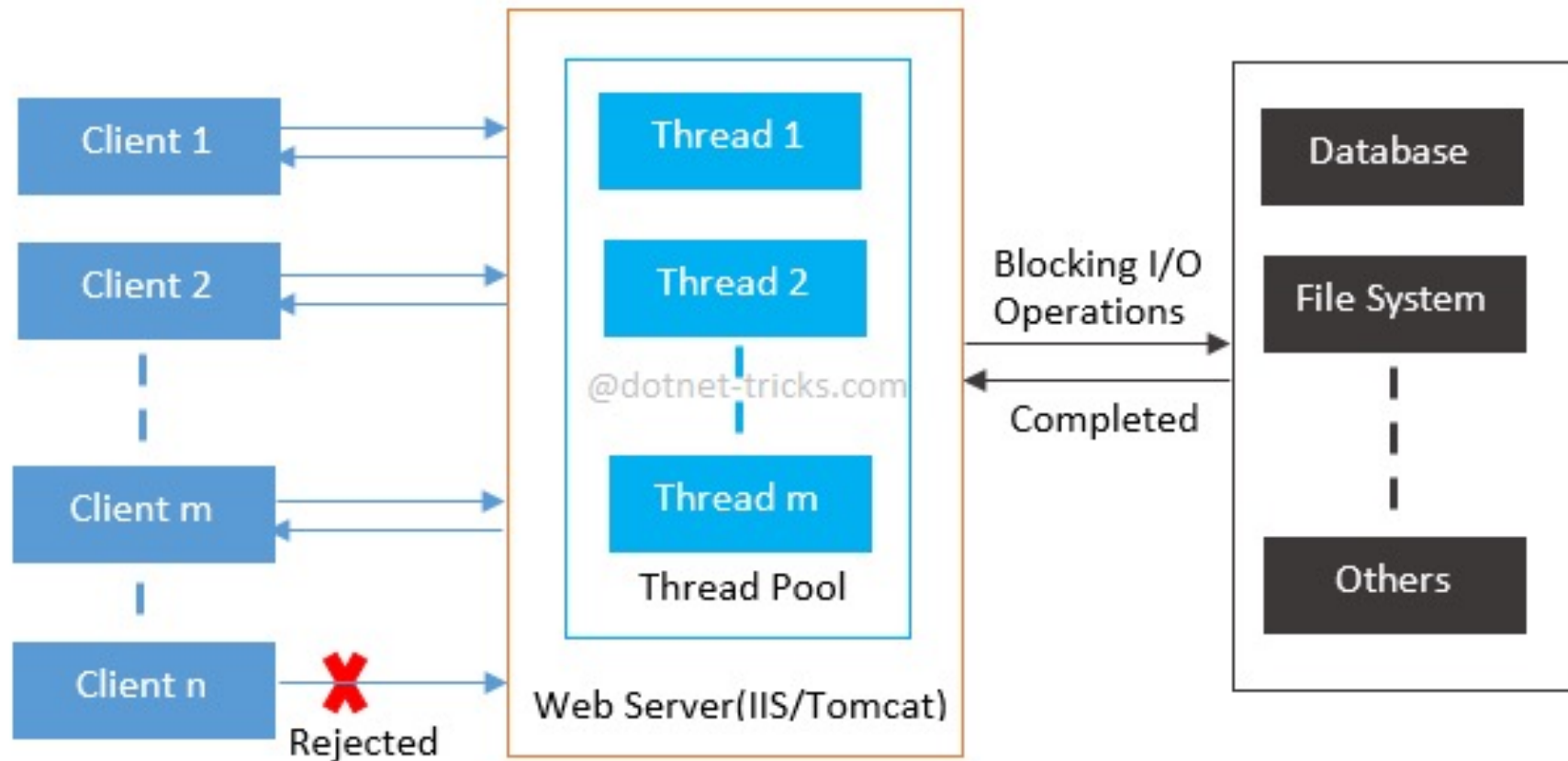
Node.js wird aber immer nur ein Event nach dem anderen bekommen und verarbeiten, also immer nur einen Callback!

Die Event-Loop ist hierfür zuständig.

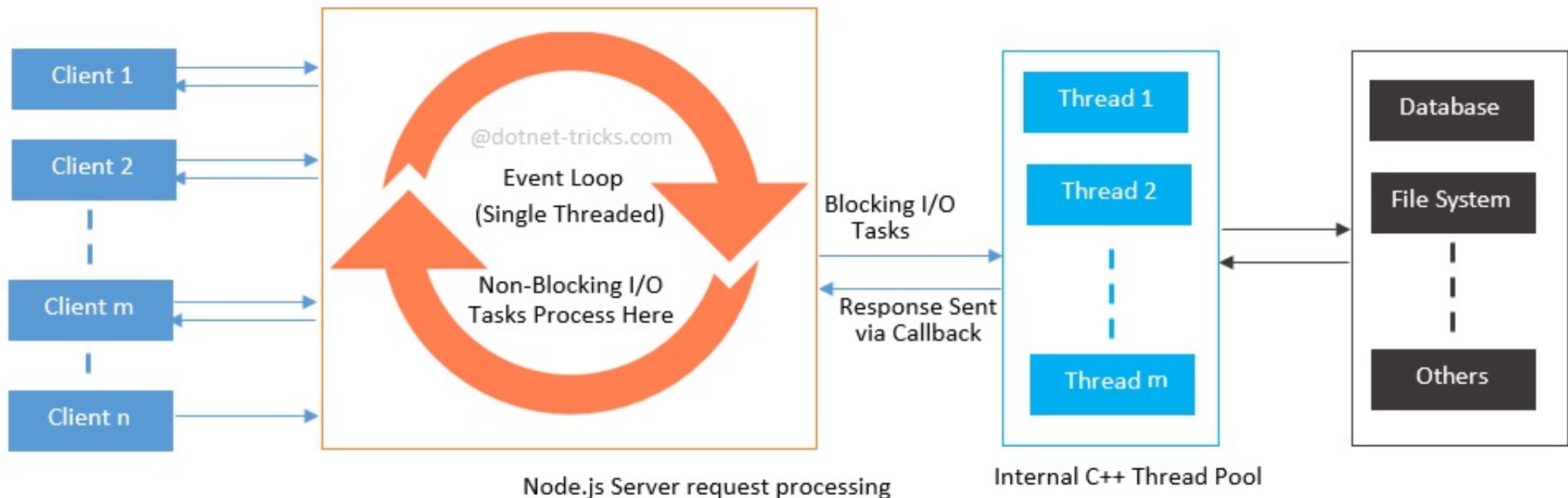
Node.js

threadbasiert vs. eventbasiert

- Webserver sind normalerweise **thread- oder prozessbasiert**
 - ein Thread oder Prozess pro Verbindung
 - sinnvoll bei komplexeren Anwendungen mit viel Rechenlast
 - Probleme:
 - Overhead durch Scheduling und Speicherbedarf
 - Synchronisation und Mandantentrennung (Isolation)
 - ..
- Node.js ist **eventbasiert**:
 - Aufträge, z.B. an das Betriebssystem werden mit einem Callback versehen. Wenn die Aufgabe erledigt ist, wird das als ein Event gesehen der sich in einer Event Queue einreihet. Diese wird gemäß FIFO abgearbeitet
 - Server wartet nicht bis die Aufträge abgeschlossen sind; stattdessen wird das Programm weiter abgearbeitet
 - In node.js verankerte Threads führen den Auftrag (blockierend) aus. Die Abgearbeitete IO-Operation wird im **Callback** dann abgearbeitet, wenn es in der Event-Queue an der Reihe ist
 - asynchron + single threaded

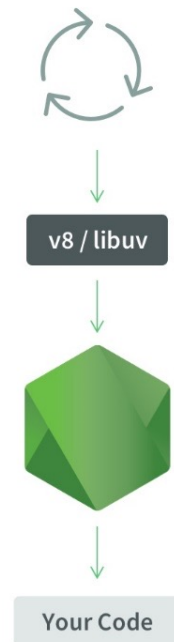


Multi-Threaded Web Server request processing

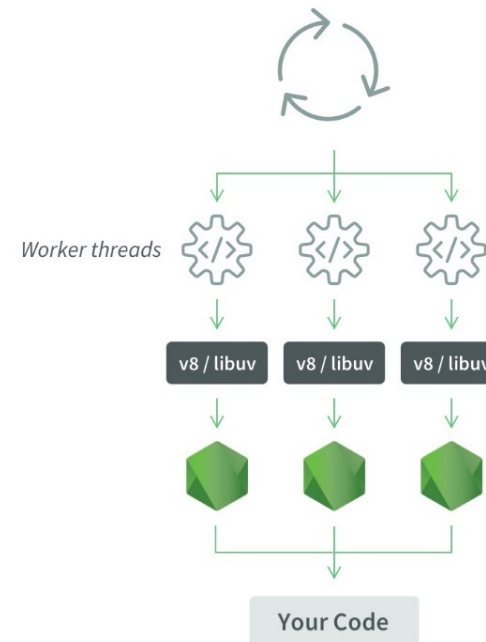


Die intern verwendete **libuv** stellt einen **Thread Pool** für blockierende Aufgaben bereit. **Dieser besteht standardmäßig aus 4 Threads.** Dies sollte gerade bei leistungsfähigen Servern erhöht werden. Anschaulich sind die Threads die Diener des Königs (node.js main Thread)

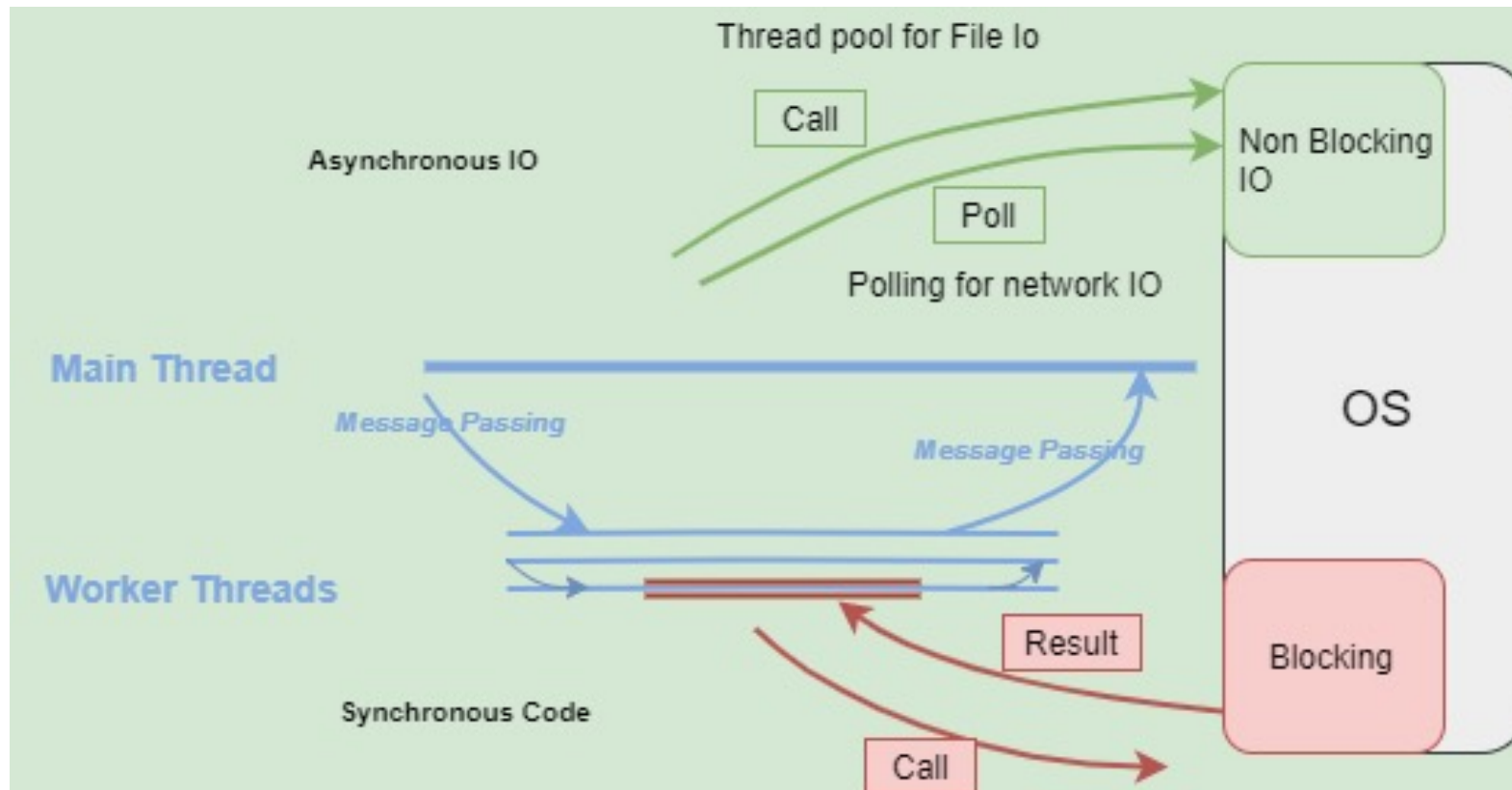
Standard Process Code



Process with Worker Threads



Tatsächlich gibt es mittels der Worker-Threads noch eine weitere Möglichkeit node.js zu skalieren. Anschaulich besitzt bei diesem Mechanismus jede Thread eine eigene JavaScript Engine mit Event-Loop und mit libuv-Thread-Pool-Anbindung



CPU-intensive Lasten sollte man über libuv lösen oder in Worker-Threads auslagern: **Never block the Event-Loop**

Node.js

Beispiel: Daten von Datei lesen



- blockierend:

```
let data = fs.readFileSync („data.csv“);
```

- nicht-blockierend:

```
fs.readFile („data.csv“, function(err, data) {  
    // do the processing here  
});
```

Hinweis:

Meistens geht man mit neueren Versionen dazu über, die asynchronen Aufgaben in eine asynchrone Funktion zu wrappen und mit await auf deren Beendigung zu warten (async-await/Promises kommen noch)

- Funktionalitäten, die in JavaScript-Dateien organisiert sind
 - Funktionen, Objekte, Variablen

Modularten

1. integrierte Module (Kernmodule)
 - müssen nicht installiert werden
2. lokale Module
 - Implementierung und Nutzung eigener Module
3. Third Party Module
 - Einbindung per Node Package Manager

Anwendungsschwerpunkte Node.js dynamische Webanwendungen

- File System
- HTTP(s)
- Path
- URL
- OS
- ...

Node.js

Third Party Module



Node Package Manager (**npm**)

- vielfältiger Projektmanager für Node.js
 - Konfiguration über `package.json`
 - Metainformationsverwaltung (Name, Beschreibung, Lizenz, ...)
 - Verwaltung des Development-Zyklusses
 - Starten des Projekts
 - Testen des Projekts
 - Bauen des Projekts
 - ...
 - Abhängigkeitsverwaltung von Third Party Modulen
 - installieren
 - updaten
 - Versions-Management
- Veröffentlichung eigener Module im npm-Repository

- wichtige Kommandozeilen-Befehle:
 - `npm init`
 - erzeugt interaktiv (mit Benutzerfragen) eine `package.json`, welche Informationen über das node.js Programm sowie alle erforderlichen Abhängigkeiten inkl. der Versionen enthält
 - `npm install <package name>`
 - Paket wird geladen und in das Unterverzeichnis `node_modules` des Projekt-Ordners installiert und in der `package.json` aufgenommen
 - `npm install`
 - installiert alle in `package.json` angegebenen Abhängigkeiten in den lokalen `node_modules` Ordner (ohne `-g` Option ist dies im aktuellen Verzeichnis und damit nur dort)
 - `-g` braucht üblicherweise Admin-Rechte (außer man nutzt einen Mac mit Homebrew)
- weitere npm Befehle: <https://docs.npmjs.com/cli-documentation/>

Node.js

Beispiel package.json

```
{  
  "name": "demo_server",  
  "description": "description",  
  "authors": "v.sander@fh-aachen.de",  
  "version": "1.0.0",  
  "main": "index.html",  
  "repository": {  
    "type": "git",  
    "url": "https://git.fh-aachen.de/demo_server.git"  
  },  
  "license": "SEE LICENSE IN /LISCENCE.MD",  
  "dependencies": {  
    "bootstrap": "^4.1.1",  
    "jquery": "^3.3.1",  
    "mustache": "^2.3.0",  
    "express": "^4.16.4",  
  }  
}
```

Relevant bei Veröffentlichung, sonst eher "private": true

Version ab der Kompatibilität gewährleistet ist

Default ist die npm-registry <https://registry.npmjs.org>

Node.js

Wichtige Third Party Module



- **Express**
 - serverseitiges Webframework
 - einfaches realisieren von Reaktion auf URL-Anfragen (Routing)
 - unser Standardeinstieg auf eine "GET /"-Anfrage des Browsers ist app.js (z.B. durch Start mit node app.js)
- **Sequelize**
 - ORM für SQL-basierte Datenbanken
 - einfache Persistenzschicht
- **Mustache**
 - Template-Engine
- Bootstrap
 - Frontend-Framework für responsives Design
- JSDoc
 - API-Dokumentations-Generator für Javascript

Node.js

Einbinden von Modulen

- Einbindung per `require`
- Beispiel 1:
 - Nutzung des integrierten Moduls `path`:

```
// app.js
const path = require('path')
```

- Beispiel 2:
 - Nutzung des 3rd-Party-Moduls `express`

```
// app.js
const express = require('express')
```

- **ACHTUNG!** Muss vorher per `npm` installiert worden sein (da 3rd-Party)!
- Hinweis: `require` sucht zunächst die integrierten Module und verwendet dann die Umgebungsvariable `NODE_PATH` um ggf. in verschiedenen Verzeichnissen nach der Datei zu suchen. Die Endung `.js` ist optional!

- Funktionalität eines Moduls muss aufgrund der Gültigkeitsregeln den Nutzern explizit zur Verfügung gestellt werden: **exports**
- Beispiel 3:
 - **Exportieren einer anonymen** Funktion in einem Modul:

```
// hellworld.js
module.exports = function () {
  console.log('bar!');
}
```

- Nutzung der Funktion in der Applikation:

```
// app.js
const hello = require('./helloworld.js');
hello();
```

- Hinweis: Wir werden ein solches Vorgehen bei der Verwendung des Express-Moduls nutzen

```
// app.js
var express = require('express');
var app = express();
```

- Beispiel 4:
 - Exportieren einer **benannten** Funktion in einem Modul:

```
// hellworld.js - Hier ohne module. - dafür mit .hello
exports.hello = function () {
    console.log('bar!');
}
```

- Nutzung der Funktion in der Applikation:

```
// app.js
const helloworld = require('./helloworld.js');
helloworld.hello();
```

- Hinweis: **bei exports muss** im Gegensatz zu module.exports immer **ein Bezug** (Properties: Klasse, Attribut, Funktion, ...) mit **angegeben werden**. Export ist genau für diesen Bezug da
- Exports regelt den Zugriff auf explizite „Properties“ (mehrere mit export {p1, p2, ...} möglich oder mehrere exports)

- Beispiel 5:
 - Exportieren eines Objektes in einem Modul

```
// Message.js  
exports.SimpleMessage = 'Hello World!'
```

oder:

```
module.exports.SimpleMessage = 'Hello World!'
```

- Nutzung des Objektes in der Applikation:

```
// app.js  
const msg = require('./Message.js')  
console.log(msg.SimpleMessage)
```

Node.js

Schreiben von Modulen

module.exports VS. exports

- exports ist ein Alias von module.exports

```
// cat.js
class Cat {
  makeSound() {
    return 'Meowww';
  }
}
```

```
module.exports = Cat;
//exports.Cat = Cat;
```

```
// app.js
const Cat = require('./cat.js');
const cat = new Cat();
//const cat = new Cat.Cat(); !
console.log(cat.makeSound())
```

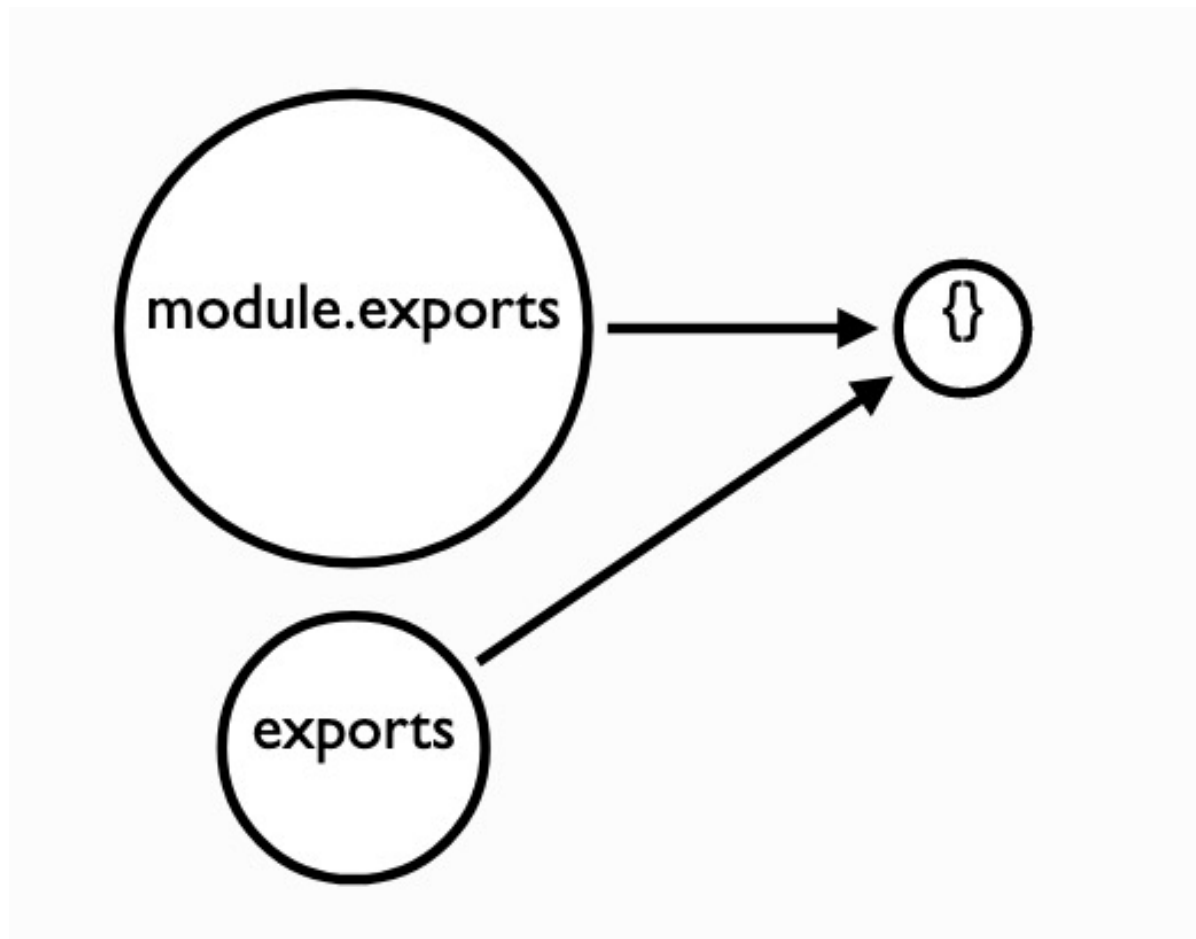
Bei Klassen besser
module.exports
verwenden!

geht so auch,
aber unschön

Node.js

Aufpassen beim Einbinden von Modulen

- Beide sind zunächst Referenzen auf das gleiche (leere) Objekt:



<https://blog.tableflip.io/the-difference-between-module-exports-and-exports/>

- ABER: Zuweisung direkt auf `exports` überschreibt das `exports`-Alias!

```
> module.exports.qux = "qux";
> exports
{ qux: "qux" }
> exports === module.exports
true
> exports = "wobble wibble wubble!";
> exports === module.exports
false
> exports
"wobble wibble wubble!"
> module.exports
{ qux: "qux" }
// module.exports is canonical
```

exports wird „umgebogen“, da keine Properties benutzt wurden

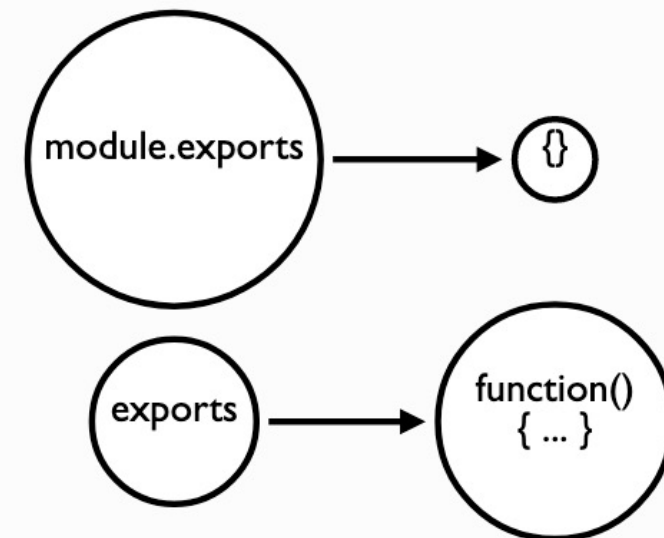
- Zuweisung immer auf `exports.property`

Zusammengefasst:

`module.exports` VS `exports`

- `module.exports` und `exports` sollten nicht gleichzeitig in einer Datei verwendet werden
 - von `exports` gesetzte Properties werden nicht mit exportiert, wenn vorher schon `module.exports` benutzt wurde!

Wenn wir nach einem `module.export` mit `exports` z.B. einen anonyme Funktion exportieren, dann passiert das folgende



<https://blog.tableflip.io/the-difference-between-module-exports-and-exports/>

Node.js ECMAScript 6 import

- Neue Möglichkeit zur Einbindung von Modulen über import
- Import läuft asynchron und passt damit gut zu Promises (kommen noch)
- ES-Module haben die Endung .mjs
- **node --experimental-modules main.mjs**

REQUIRE

Require is Non-lexical, it stays where they have put the file.

It can be called at any time and place in the program.

You can directly run the code with require statement.

If you want to use require module then you have to save file with '.js' extension.

ES6 IMPORT AND EXPORT

Import is lexical, it gets sorted to the top of the file.

It can't be called conditionally, it always run in the beginning of the file.

To run a program containing import statement you have to use experimental module feature flag.

If you want to use import module then you have to save file with '.mjs' extension.

<https://www.geeksforgeeks.org/difference-between-node-js-require-and-es6-import-and-export/>

Node.js

einfachste Node.js Anwendung

- Node.js beinhaltet eine Skript-Engine für Javascript
 - funktioniert ähnlich wie Python, PHP, ...
 - Skripte werden top to bottom abgearbeitet
 - Aufpassen: Asynchron nicht blockierend

```
// demo.js
function concat(a,b) {
    return a+b;
}
let a = "Javascript ist";
let b = " aus dem Web nicht
mehr wegzudenken";
console.log(concat(a+b));
```

Applikation starten:

```
node demo.js
```

Output:

```
Javascript ist aus dem
Web nicht mehr
wegzudenken
```

Was wollen wir eigentlich?

- Daten bereitstellen
- Webseite bereitstellen (Frontend)
- Dienste (Services) bereitstellen (Backend)
- Web-Applikation (Frontend + Backend) bereitstellen

Klassisch

Anwendungsfall	Software (Beispiel)
Daten bereitstellen	Apache (htdocs) / php (Datenbank)
Webseite bereitstellen	Apache (htdocs)
Dienste (Services) bereitstellen	Tomcat → Java Servlet /php

Node.js: Ereignisgesteuert

- Node.js implementiert im **http-Modul** einen Webserver
- jede HTTP-Anfrage löst bei Node.js ein Ereignis aus
- Funktionen reagieren auf die Anfragen und verarbeiten die Daten (hier **handleRequest**)
- Einbindung des Moduls: `const http = require('http');`
- Nutzung des Moduls: `let server = http.createServer(handleRequest);`
- Starten des HTTP-Servers mittels `server.listen`-Methode, die faktisch nur das listen und nicht das accept macht:

```
server.listen(3001, function() {  
    console.log("Server listening on:  
                http://localhost:" + port);  
});
```

Node.js

Serverseitiges JavaScript

Node.js: erste Server-Applikation (app.js)

```
let http = require('http');
```

Server-Port

```
// port  
const port = 3001;
```

Request
Funktion

```
// request handle function  
function handleRequest(request, response){  
  response.end('request URL: http://localhost:  
+ port + request.url);  
}
```

create
Server

```
// create the server  
let server = http.createServer(handleRequest);
```

start
Server

```
// start the server  
server.listen(port, function(){  
  console.log("Server listening on: http://localhost: + port);  
});
```

Server im Terminal starten:

```
node purenodeserver.js
```

Zugriff auf Server via Web-Browser:

```
http://localhost:3001/Hallo
```

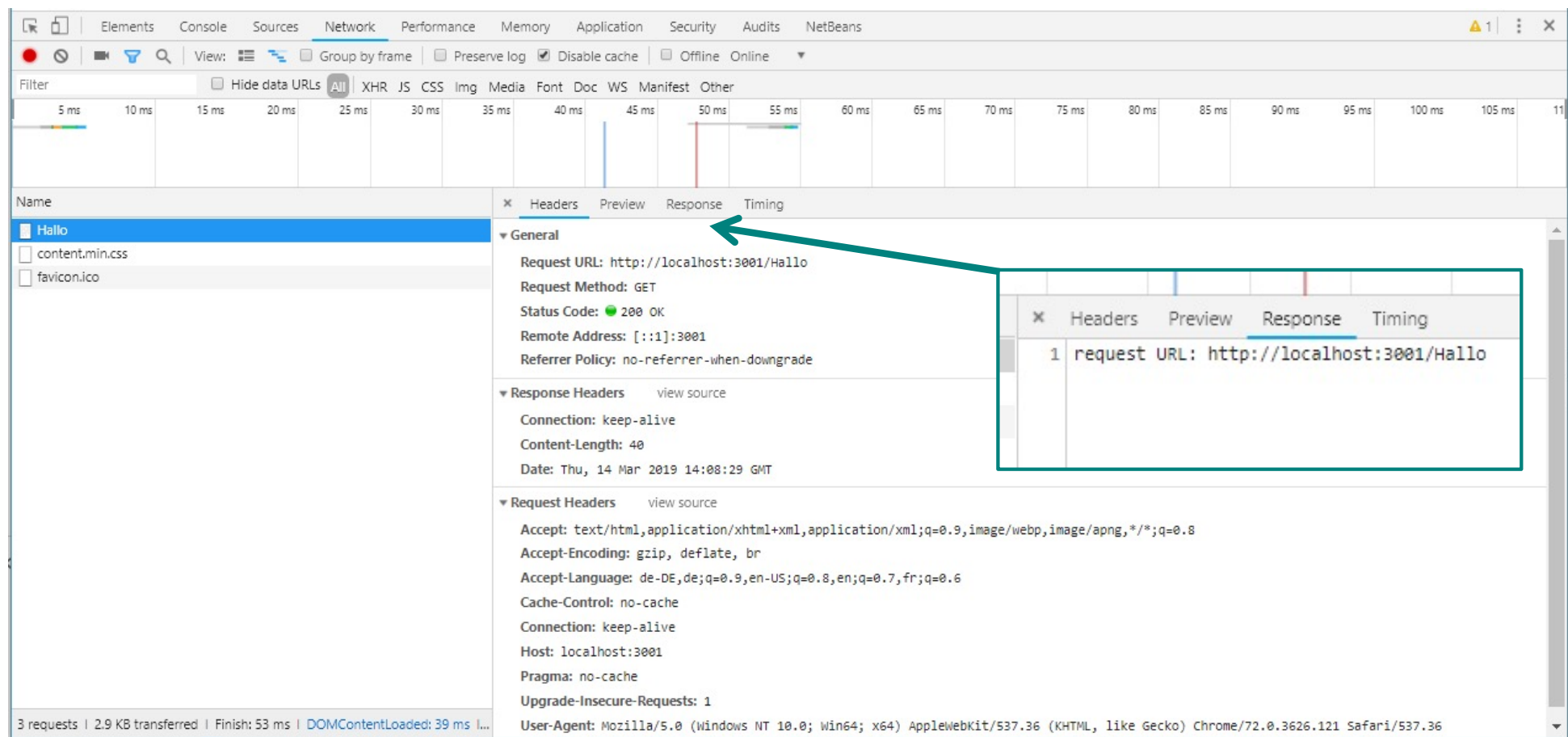
Browser-Ausgabe:

```
Request URL: http://localhost:3001/Hallo
```


Node.js

Serverseitiges JavaScript

Browser-Developertools (F12)



Node.js

Backend-Frameworks – StateOfJS 2020

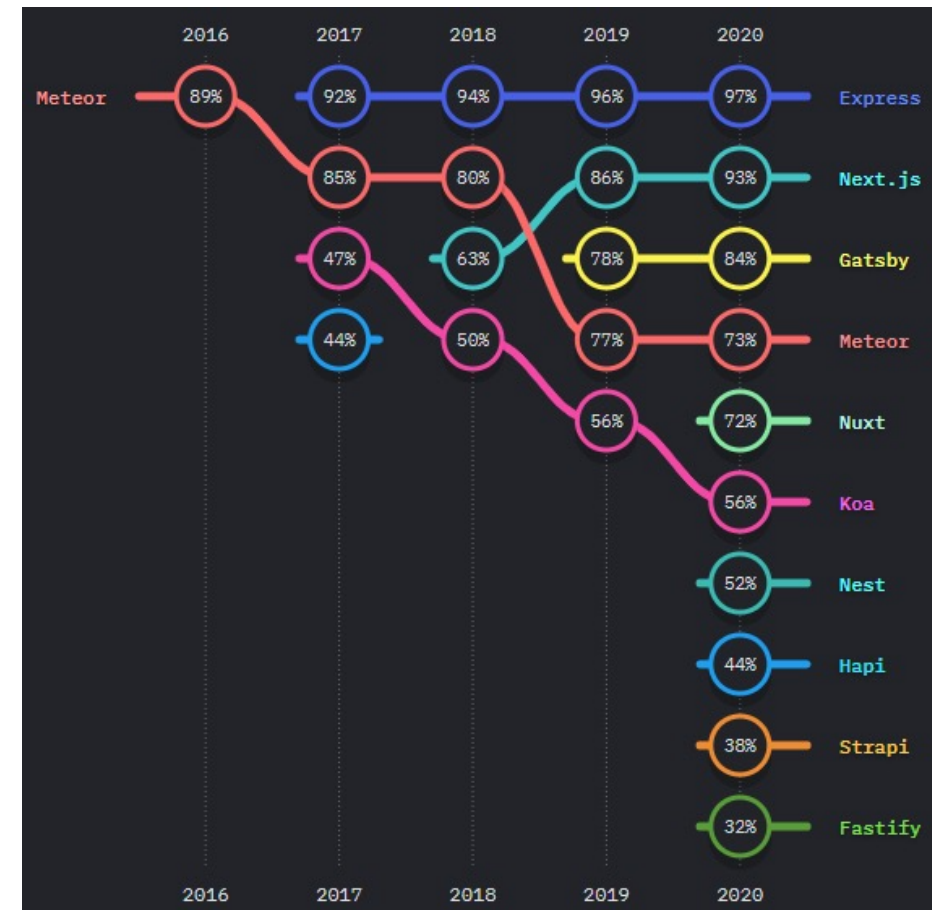
Usage

(would use again + would not use again) / total



Awareness

(total - never heard) / total



<https://2020.stateofjs.com/en-US/technologies/back-end-frameworks/>

Express.js

- Meistgenutztes Web-Framework für Node.js
 - Einbindung: `require('express');`
- Open Source, MIT-Lizenz
- Funktionalitäten
 - Routing
 - Middleware-Module
 - Ausliefern von statischen Dateien
 - Cookies und Session Manipulation
 - ...



- Läuft „hinter“ dem Node.js HTTP Server
- Express wird faktisch als Modul (Middleware) mit in die Bearbeitungskette der HTTP-Anfragen in node.js eingebunden

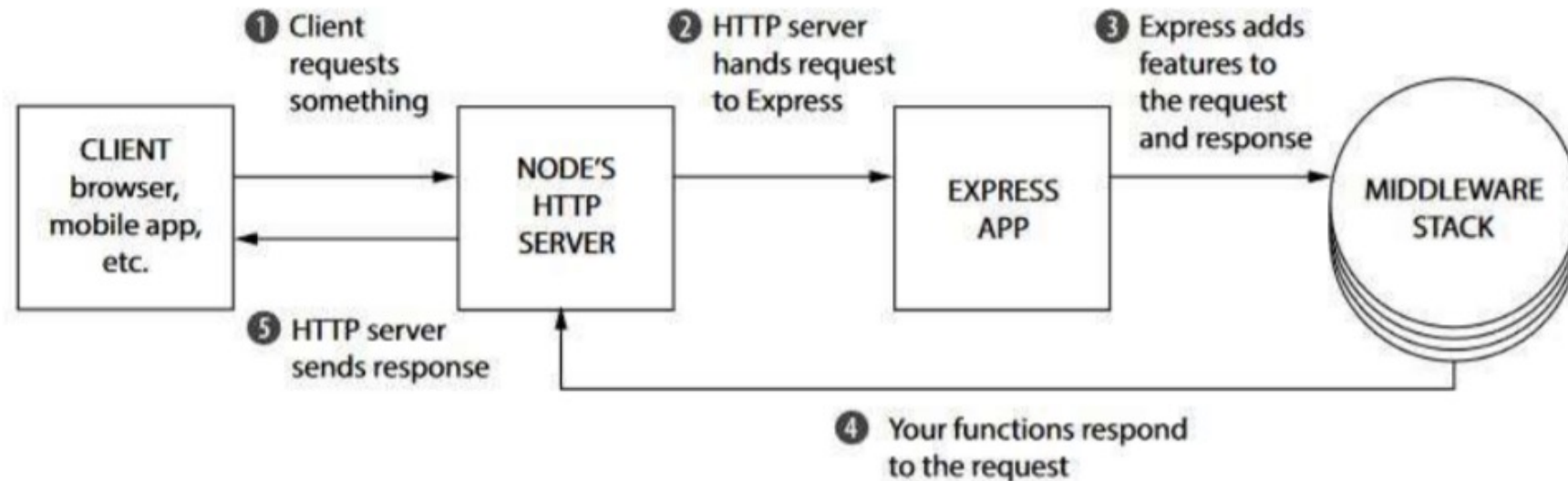


Abb.-Quelle: Evan M. Hahn: Express in Action - Writing, building, and testing Node.js applications, Manning Publications, 2016

vanilla Node.js (app.js)

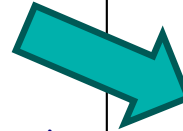
```
let http = require('http');

// port
const port = 3001;

// request handle function
function handleRequest(request, response){
  response.end('Hello World');
}

// create the server
let server = http.createServer(handleRequest);

// start the server
server.listen(port, function(){
  console.log("Server listening on:
http://localhost:" + port);
});
```



Express (app.js)

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

app.get('/', function(req, res){
  res.send('Hello World');
});

app.listen(3000, function(){
  console.log('Example app
listening on port 3000!');
});
```

Route



Hinweis: app.get bezieht sich auf eine HTTP-GET-Operation auf der angegebenen URL! app ist hier das aktivierte express-Modul

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

app.get('/', function(req, res, next) {
  next();
});

app.listen(3000);
```

HTTP method for which the middleware function applies.

Path (route) for which the middleware function applies.

The middleware function.

Callback argument to the middleware function, called "next" by convention.

HTTP **response** argument to the middleware function, called "res" by convention.

HTTP **request** argument to the middleware function, called "req" by convention.

Was sind Routen?

- Navigation innerhalb der Anwendung
- setzen sich zusammen aus HTTP-Methode und URL
 - URL:
`http://localhost/myModule/subRoutine`
 - zugehörige Route (bei `app.get`):
`GET /myModule/subRoutine`
 - das gilt genau nur für die Route (oder die zutreffenden Routen)
- **HTTP-Methode + URL** wird auf eine Callback-Funktion abgebildet:

```
// express
app.get('/myModule/subRoutine', function(req, res) {
    // ...
});
```


`app.js` (Einstiegspunkt der Applikation) sollte bei größeren Projekten eigentlich keine Implementierungsdetails der Routen enthalten

→ daher: Auslagerung der Routen abhängig von der Funktionalität in Dateien

→ Einbindung der Routen als Module in `app.js` (`require`)

//login.js

Beispiel: `module.exports = function (app) {`
 `app.get('/login', function (req, res, next) {`
 `res.render(...)`
 `});`
`};`

Hier wird eine Javascript Datei Aufgerufen (template), üblicherweise im views-Verzeichnis

//app.js ruft login.js auf und übergibt express (app)!
`const express = require('express');`
`let app = express();`
`require('./routes/login.js')(app)`


```
//login.js
```

```
module.exports = function (app) {  
  app.get('/login', function (req, res, next) {  
    res.render( ... )  
  });  
};
```



alle Express-konformen
Template-Engines exportieren
eine Funktion, die über `res.render()`
indirekt aufgerufen wird und so
Vorlagen in HTML überführt

```
//app.js
```


```
require('./routes/login.js')(app)
```

Wir nutzen hier also ein zusätzliches Modul (Template-Engine),
um die Antwort (HTML) über Vorlagen (Templates) zu erzeugen

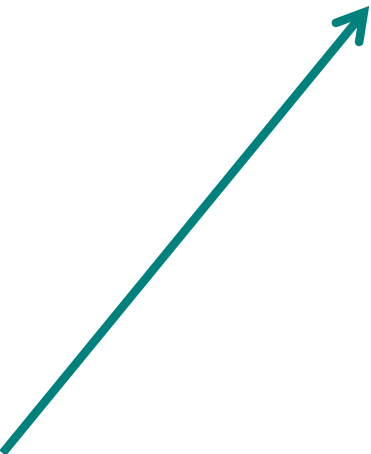
Damit wir die Vorlagen so auch nutzen können, müssen gewisse Einstellungen passen

- Require auf die verwendete Template-Engine
- Festlegen der Engine in Express für das render:
`app.set('view engine', 'mustache')`
- Festlegen eines Verzeichnisses, in dem sich die Vorlagendateien befinden. Beispiel: `app.set('views', '.__dirname + './views')`

Umgebungsvariable, die das Verzeichnis angibt, in dem node ausgeführt wird. Diese ist normalerweise automatisch gesetzt



Im Unterverzeichnis views sind die Template-Dateien (Default-Endung: .html)



Problem: Routen enthalten Strukturinformationen (absolute URLs). Code nicht wiederverwendbar/ schwerer zu Parametrisieren

Lösung: Router:

- Sammlung von Routen zur besseren Organisation der Applikation
- Übernahme der Hierarchie-Aufgabe: applikationsspezifische **Sub-Pfade** können einfach definiert und **von den Routen getrennt** werden
→ ermöglichen eine Wiederverwendbarkeit der Module

- ohne Router:

```
app.get('/wichtigeURL/first', ...  
app.get('/wichtigeURL/second', ...  
app.get('/wichtigeURL/third', ...
```

- mit Router:

```
const router = express.Router();  
router.get('/first', ...  
router.get('/second', ...  
router.get('/third', ...  
module.exports = router
```

```
//app.js  
app.use('/wichtigeURL', router)
```

```
const router = express.Router();
router.get('/first', ...
router.get('/second', ...
router.get('/third', ...
module.exports = router

//app.js
app.use('/first', router)
```

`app.use` ist wichtig, denn es bezieht sich auf ein Konzept in express:

Middleware!

In unserem Fall gilt: Wir binden die router-Funktion (Middleware) ein (mit all ihren get-Defintionen und setzen dabei den Pfad (/wichtigeURL) auf den sie wirkt!

Express.js

Router Beispiel

```
let express = require('express');
let app = express();
let router1 = express.Router();
let router2 = express.Router();
app.use('/Products', router1);
app.use('/Customers', router2);
router1.get('/', function(req, res){
    res.send('Products');
});
router2.get('/', function(req, res){
    res.send('Customers');
});
router2.get('/Count', function(req, res){
    res.send('50 Customer');
});
app.listen(3000, function(){ console.log('Example app
listening on port 3000!'); });
```

Router

URL Festlegung
für die
Middleware

Products/

Customers/

Customers/Count

Express.js

Router Beispiel - Refactoring

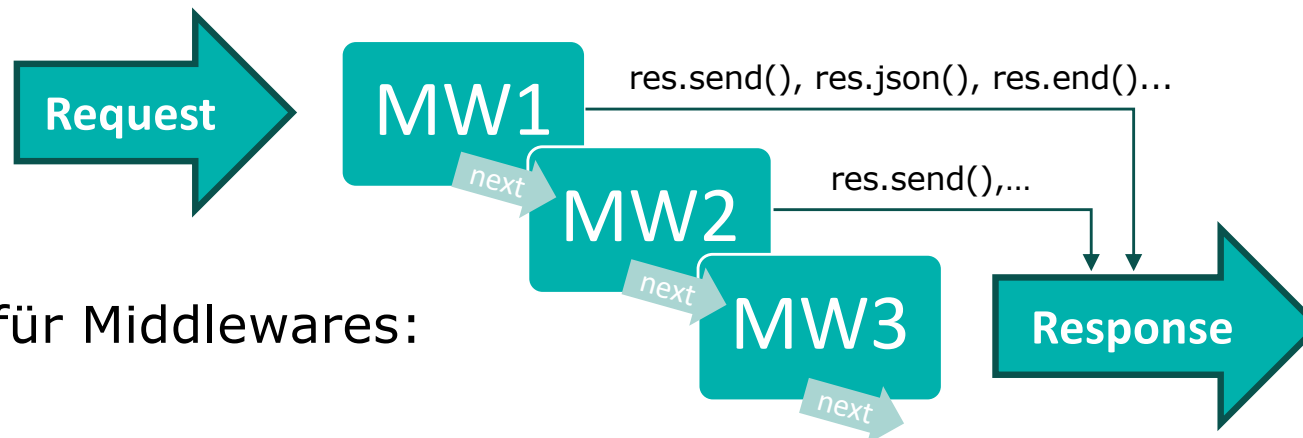
```
const express =  
require('express');  
let router2 = express.Router();  
  
router2.get('/', function(req,  
res){...});  
router2.get('/Count',  
function(req, res){...});  
module.exports = router2;
```

```
const express =  
require('express');  
let router1 =  
express.Router();  
  
router1.get('/',  
function(req, res){...});  
module.exports = router1;
```

```
const express = require('express');  
let app = express();  
let router1 = require('./Routes/router1.js');  
let router2 = require('./Routes/router2.js');  
app.use('/Products', router1);  
app.use('/Customers', router2);  
app.listen(3000, function(){ console.log('Example app  
listening on port 3000!'); });
```

Was sind Middlewares?

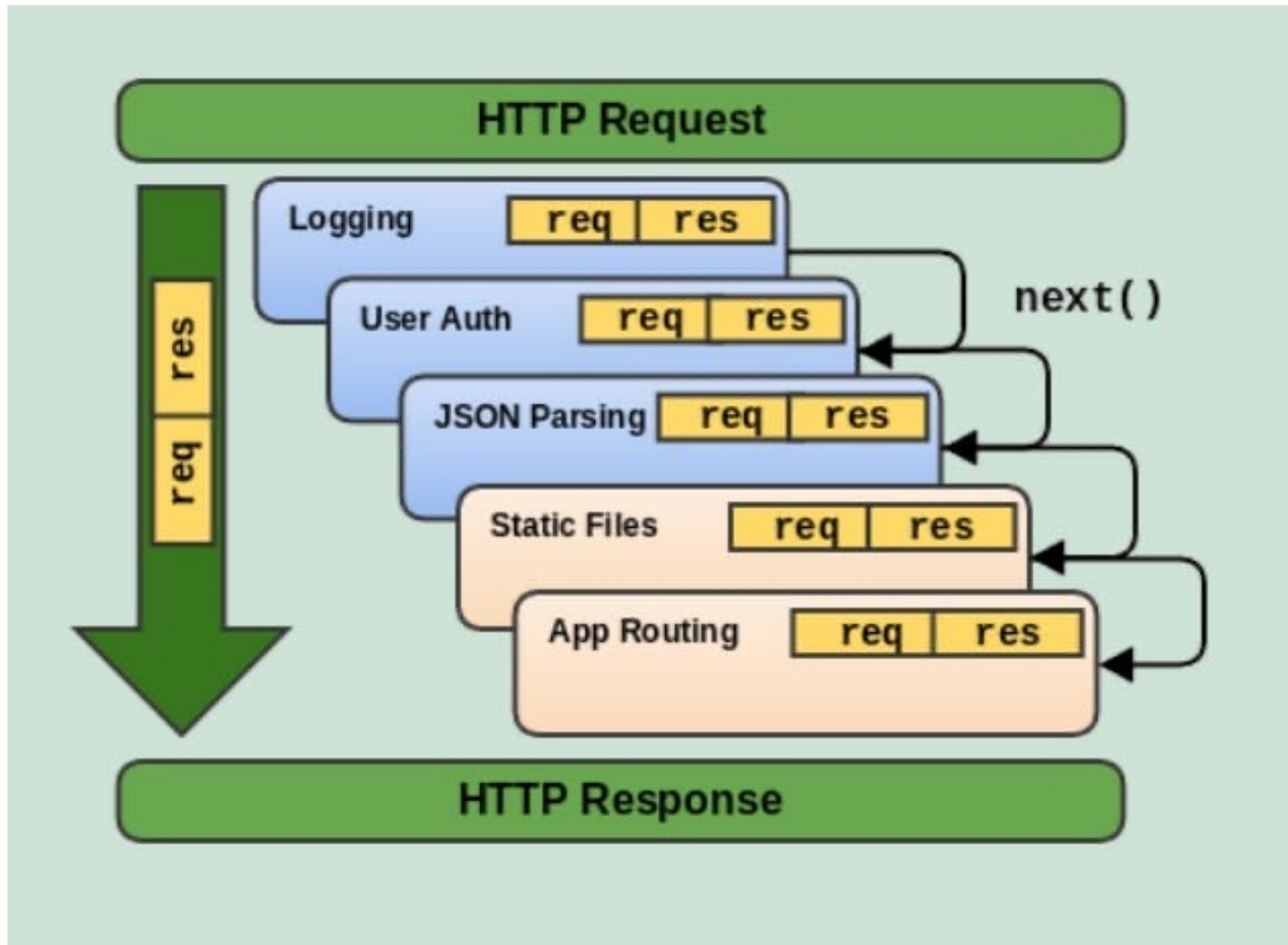
- Stack von Funktionen mit beliebigem Inhalt
- werden zwischen dem Eingehen eines Requests und dessen Bearbeitung durch einen Handler geschaltet
- haben Zugriff auf Request, Response und nächste Middlewarefunktion (über Variable next). Aufpassen: `res.send()` verschickt die Nachricht



- Beispiele für Middlewares:
 - Parsing
 - Autorisierung
 - Logging
 - Error Handling
- Verwendung der Middleware per `app.use`

Express Middleware

<https://dev.to/ghvstcode/understanding-express-middleware-a-beginners-guide-g73>



Express Middleware

<https://expressjs.com/en/guide/routing.html>

```
const express = require('express')
const router = express.Router()

// middleware that is specific to this router
router.use((req, res, next) => {
  console.log('Time: ', Date.now())
  next()
})
// define the home page route
router.get('/', (req, res) => {
  res.send('Birds home page')
})
// define the about route
router.get('/about', (req, res) => {
  res.send('About birds')
})

module.exports = router
```

```
const birds = require('./birds')

// ...

app.use('/birds', birds)
```

birds.js

Express.js Middlewares

```
let express = require('express');  
let app = express();  
const user = 'Volker';  
const password = '1234'
```

Definition
einer
Middleware

```
function middleHandler(req, res, next){  
  console.log(req.originalUrl);  
  next();  
}
```

```
app.use('/', function(req, res, next){  
  // Authorization test  
  let query_user = req.query.user;  
  let query_pw = req.query.pw;  
  if(query_user === user && query_pw === password){  
    next();  
  }else{  
    res.send('Zugriff verweigert');  
  }  
});
```

Eigene
Middleware
für alle Routen

```
app.get('/', middleHandler, function(req, res){  
  res.send('Zugriff gestattet');  
});  
app.listen(3002);  
console.log('start server on port 3002');
```

Benutzung zweier
Middleware
Beendet durch
Middleware ohne next

Express.js

Aufruf mit GET

```
let express = require('express');  
let app = express();  
const user = 'Volker';  
const password = '1234'
```

```
function middleHandler(req, res, next){  
  console.log(req.originalUrl);  
  next();  
}
```

Dann das

```
app.use('/', function(req, res, next){  
  // Authorization test  
  let query_user = req.query.user;  
  let query_pw = req.query.pw;  
  if(query_user === user && query_pw === password){  
    next();  
  }else{  
    res.send('Zugriff verweigert');  
  }  
});
```

Erst wird das
ausgeführt

```
app.get('/', middleHandler, function(req, res){  
  res.send('Zugriff gestattet');  
});  
app.listen(3002);  
console.log('start server on port 3002');
```

Zuletzt das

Express.js

Aufruf mit POST

```
let express = require('express');  
let app = express();  
const user = 'Volker';  
const password = '1234'
```

```
function middleHandler(req, res, next){  
  console.log(req.originalUrl);  
  next();  
}
```

```
app.use('/', function(req, res, next){  
  // Authorization test  
  let query_user = req.query.user;  
  let query_pw = req.query.pw;  
  if(query_user === user && query_pw === password){  
    next();  
  }else{  
    res.send('Zugriff verweigert');  
  }  
});
```

```
app.get('/', middleHandler, function(req, res){  
  res.send('Zugriff gestattet');  
});
```

```
app.listen(3002);  
console.log('start server on port 3002');
```

Nur das wird
ausgeführt

[app.METHOD](#) (<pfad>, <middleware>) und [app.use](#) (<pfad>, <middleware>)

Mögliche Angaben für den Pfad:

- Eine String-Darstellung des Pfades
- Ein Pfad Muster (Path Pattern, Untermenge von regulären Ausdrücken)
- Ein regulärer Ausdruck
- Ein Array mit einer Kombination der angegebenen Möglichkeiten

Unterschiede zwischen **app.METHOD (Routing von Anfragen)** und **app.use (Einbindung von Middlewares)**:

- Bei `app.use()` gilt der Pfad auch für Subpfade
 - `app.use('/', ...)` gilt für `'/'` aber auch für `'/products'`
- `app.METHOD()` unterstützt [Pfadparameter](#) (Route parameters)
 - `app.get('/products/:id', ...)` gilt für `/products/42` und `"42"` ist über `req.params.id` auslesbar (**Achtung** ist immer ein String)
 - Der Pfadparameter ist hier exakt gemeint und gilt ohne reguläre Ausdrücke nicht rekursiv

Express ermöglicht den Aufbau einer Verarbeitungskette durch Middlewares

- `app.METHOD()` bindet einen Handler (ggf. Kette) also Middleware **für eine Route** und eine bestimmte Methode (z.B. GET, POST, PUT, PATCH, DELETE) ein
- `app.use()` bindet einen Handler (ggf. Kette) also Middleware für alle Anfragen an einen Pfad und alle Subpfade (ohne Pfad heißt `'/'`). Anschaulich ist das ein Mount der Middleware in dem Pfad
- Die Reihenfolge der Einbindung im Source-Code bestimmt die Ausführungsreihenfolge
- Ohne Aufruf von `next()` bricht die Kette ab!
- `app.all('*')` **entspricht** `app.use('/')`
- **Vorsicht!**
- `app.use('/api')` **springt bei den URLs `/api` und `/api/resource` an,
`app.all('/api/*')` **aber nur bei `/api/resource` und nicht `/api`****

Express.js

Routen - Middlewares

```
const express = require('express')
const app = express()
app.all('/api/*', function(req, res, next) {
  console.log('only applied for routes that begin with /api')
  next()
})
```

Hinweis:

Die Middlewares senden ihre Antwort typischerweise mittels `res.send()` oder `res.json()`

Diese Aufrufe schließen den Antwort-Stream, so dass keine weiteren Ausgaben, z.B. in anderen Middlewares möglich sind

Aber man kann in den Middlewares beispielsweise HTTP-Header setzen

```
res.set('Content-Type', 'text/plain');
```

```
res.set({ 'Content-Type': 'text/plain',  
          'Content-Length': ,123  
});
```


Hinweis:

Wenn `res.send()` oder `res.json()` den Antwort-Stream schließen, warum gibt es aber die `res.end()`-Methode überhaupt?

Wenn Sie keine Antwort verschicken wollen, dann ist `res.end()` die Lösung

```
res.status(404).end();
```

express - das Response Objekt

Responses können auf unterschiedliche Art und Weise gebaut werden:

Methode	Beschreibung
<code>res.download()</code>	Gibt eine Eingabeaufforderung zum Herunterladen einer Datei aus.
<code>res.end()</code>	Beendet den Prozess "Antwort".
<code>res.json()</code>	Sendet eine JSON-Antwort.
<code>res.jsonp()</code>	Sendet eine JSON-Antwort mit JSONP-Unterstützung.
<code>res.redirect()</code>	Leitet eine Anforderung um.
<code>res.render()</code>	Gibt eine Anzeigevorlage aus.
<code>res.send()</code>	Sendet eine Antwort mit unterschiedlichen Typen.
<code>res.sendFile</code>	Sendet eine Datei als Oktett-Stream.
<code>res.sendStatus()</code>	Legt den Antwortstatuscode fest und sendet dessen Zeichenfolgedarstellung als Antworthauptteil.

<https://expressjs.com/de/guide/routing.html>

Hinweis: Die Antwort wird bei einigen automatisch beendet. Außerhalb der express-API gibt es auch noch `res.write()`, bei dem dies nicht der Fall ist.

express - das Response Objekt

redirect

`res.redirect([status,] path)`

Beispiel 1: Logout

```
router.get('/logout', (req, res) => {  
  req.session.destroy((err) => {  
    if(err) return console.log(err)  
  });  
  res.redirect('/')  
});
```

Beispiel 2: Weiterleitung von einer alten URL auf die neue URL:

```
app.get('/old-url/', (req, res, next) => {  
  res.redirect(301, 'https://domain.com/new-url')  
})
```