Template Engine mit node.js

- **Ziel:** Saubere Trennung von Darstellung und Inhalt
  → Trennung von HTML und Node.js-Code
- **Lösung:** Template-Engine
- **Idee:**
  > Einlesen einer HTML-Vorlage (Template) mit vorgesehenen Platzhaltern
  > Ersetzung der Platzhalter durch Inhalte

- Muss ich die Template-Engine selber implementieren?

NEIN!

Template Engine
Mustache.js

Die elementare Nutzung (Quelle in der Abbildung)

```
const person = {
    firstName: "Christophe",
    lastName: "Coenraets",
    blogURL: "http://coenraets.org"
};
var template = "<h1>{{firstName}}
{{lastName}}</h1>Blog:{{blogURL}}";
var html = Mustache.to_html(template,
person);
```

Inhalt

Aus dem Template durch Ersetzung erstelltes HTML

Template Engine
Mustache.js

```
<div>
    <img
src="img/avatars/{{avatar}}" />
    <p>{{name}}</p>
</div>
```

Darstellung

```
const mustache =
require('mustache')

//template laden ...

inhalt={
    'name': 'Klaus',
    'avatar': 'BudSpencer.jpg'
}
var html =
mustache.render(template,
inhalt)
```

Inhalt

Template Engine
Mustache.js

Template laden?

```
const mustache = require('mustache')

//template laden ...

inhalt={
    'name': 'Klaus',
    'avatar': 'BudSpencer.jpg'
}
mustache.render(template, inhalt)
```

Template Engine
Mustache.js

Template laden?

Express hilft!

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

let route = express.Router()


let path = require('path');
const mustacheExpress = require('mustache-express')
app.set('view engine', 'mustache')
app.engine('html', mustacheExpress());
app.set('views', path.join(__dirname , 'views'))

router.get('/', function(req, res){
        inhalt={
          'name': 'Klaus',
          'avatar': 'BudSpencer.jpg'
        }
        res.render('tpl_datei_views.html', inhalt)
}
```

Gibt das generierte
html direkt zurück

Template Engines Consolidate

Template Engines sind nicht immer passend für Express

- Es gibt eine Reihe von Template Engines
- Nicht alle implementieren die für Express notwendige Funktion `__express(filePath, options, callback)`
- Wir haben deshalb auch besser mustache-express verwendet
- Das Package consolidate bietet eine passende Zwischenschicht für verschiedene Template-Engines an

- Beispiel:

```
const consolidate = require('consolidate')
const express = require('express‘)
let app = express();
app.engine('html', consolidate.mustache)
app.set('view engine', 'html')
```

Zustände
Cookies & Sessions

Zustand des Node.js-Skriptes geht nach Ausführung verloren

- Variablen sind nur für einen Aufruf gültig

Wie können mehrstufige Operationen/Transaktionen vorgenommen werden?

- Der Server muss erkennen, dass der nächste Aufruf der Seite in einem Kontext geschieht
- ISO/OSI, Schicht 5? Session?
  - > Nein, im HTTP-Protokoll ist das Aufgabe der Applikation!

Persistente JS Skripte

- Parameter, der den aktuellen „Schritt" speichert
- Cookies & Sessions bieten die Möglichkeit den Inhalt von Variablen länger zu speichern

Zustände
Cookies

Cookies bieten die Möglichkeit clientseitig Daten zu speichern

- Benutzer muss sich nicht mehrfach anmelden
- Aber auch Tracking des Nutzerverhaltens möglich

- Cookies werden im Header der Seite zurückgeliefert
- Das Setzen von Cookies muss daher erfolgen bevor Inhalt ausgegeben wird

- Beispiel:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=value
Set-Cookie: foo=bar; Expires=Wed, 09 Jun 2021 10:18:14 GMT

(Inhalt der Seite)
```

Zustände
Cookies

Speicherung des Cookies als „Datei“ auf dem Client

- Tatsächlich meist mit Hilfe von Datenbanken realisiert

Client sendet beim nächsten Aufruf der Webseite den Cookie mit

- Nur die Webseite, die den Cookie gesetzt hat, kann den Cookie lesen

- Beispiel:

```
GET / HTTP/1.1
Host: www.example.org
Cookie: name=value; foo=bar
…
```

Setzen eines Cookies in express

- **`res.cookie('name', 'value' [, options])`**
  - speichert einen Cookie mit dem Namen `name` und Wert `value`
  - `value` kann ein String oder ein Objekt (Serialisierung zu JSON) sein
  - `options` kann verschiedene Properties haben:

    einige Beispiel-Properties:
    - `expires`: Verfallsdatum des Cookies
    - `path`: gibt an von welchen Verzeichnissen der Cookie gelesen werden kann
    - `domain`: gibt an an welche Domains der Cookie ausgeliefert werden soll
    - **`httpOnly`**: kennzeichnet den Cookie als nur für den Webserver zugänglich
    - …
  - Beispiel:

```
res.cookie('rememberme', '1',
                    { expires: new Date(Date.now() + 900000),
                      path: '/' })
```

> Cookie „rememberme“ mit dem Wert „1“ läuft in 15 Minuten ab und kann von allen Verzeichnissen in der Domain aus gelesen werden.

Weitere Informationen: http://expressjs.com/de/4x/api.html#res.cookie

Lesen eines Cookies, den der Client gesendet hat

- Middleware: cookie-parser notwendig

```
var express = require('express')
var cookieParser = require('cookie-parser')

var app = express()
app.use(cookieParser())
```

- Liefert ein cookies-Objekt mit den Keys des Cookies
- Beispiel:

```
req.cookies.rememberme
```

- kann nicht im selben Skriptdurchlauf gespeichert und wieder gelesen werden
  - > Cookie wird erst beim Senden des Clients im Req-Objekt eingetragen
  - > Erst der nächste Skriptdurchlauf kann den Cookie lesen

Weitere Informationen: http://expressjs.com/de/4x/api.html#req.cookies

Zustände
Sessions

Cookies

- speichern alle Informationen clientseitig ab
- Daten können vom Benutzer manipuliert werden

Sessions

- speichern die Sitzungsdaten serverseitig ab
- Client erhält nur eine Session-ID (als Cookie)
  > Um das Setzen des Cookies kümmert sich Node
- Daten, die per Session gespeichert werden, werden auch nur auf dem Server gespeichert
  > keine Datenmanipulation möglich
  > komplexe Variablen (Arrays, Objekte) sind möglich

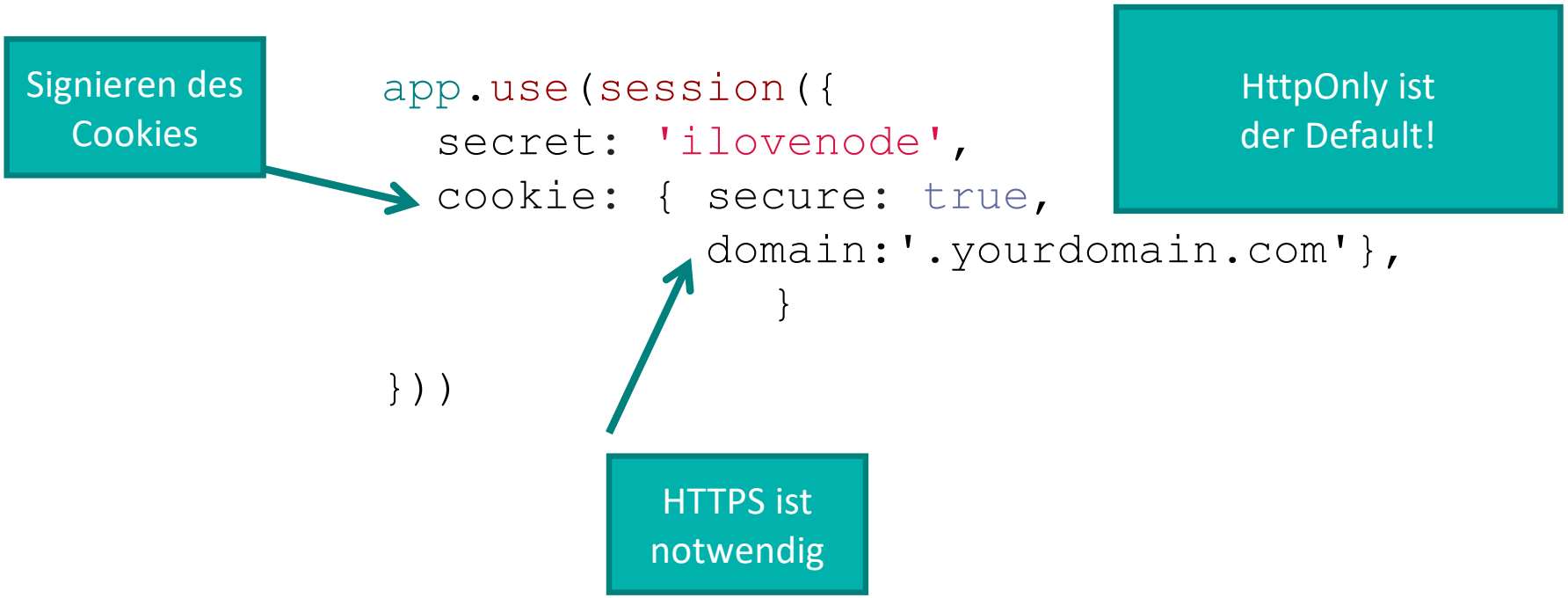
Zustände
Sessions

Benutzung in Node.js

- Middleware express-session notwendig

```
var session = require('express-session')
```

- **Aktivierung des Session-Features per `app.use`**



Weitere Informationen: https://www.npmjs.com/package/express-session

Zustände
Sessions

- **Beenden einer Session**

```
req.session.destroy()
```

  - löscht `req.session`

Zustände
Sessions

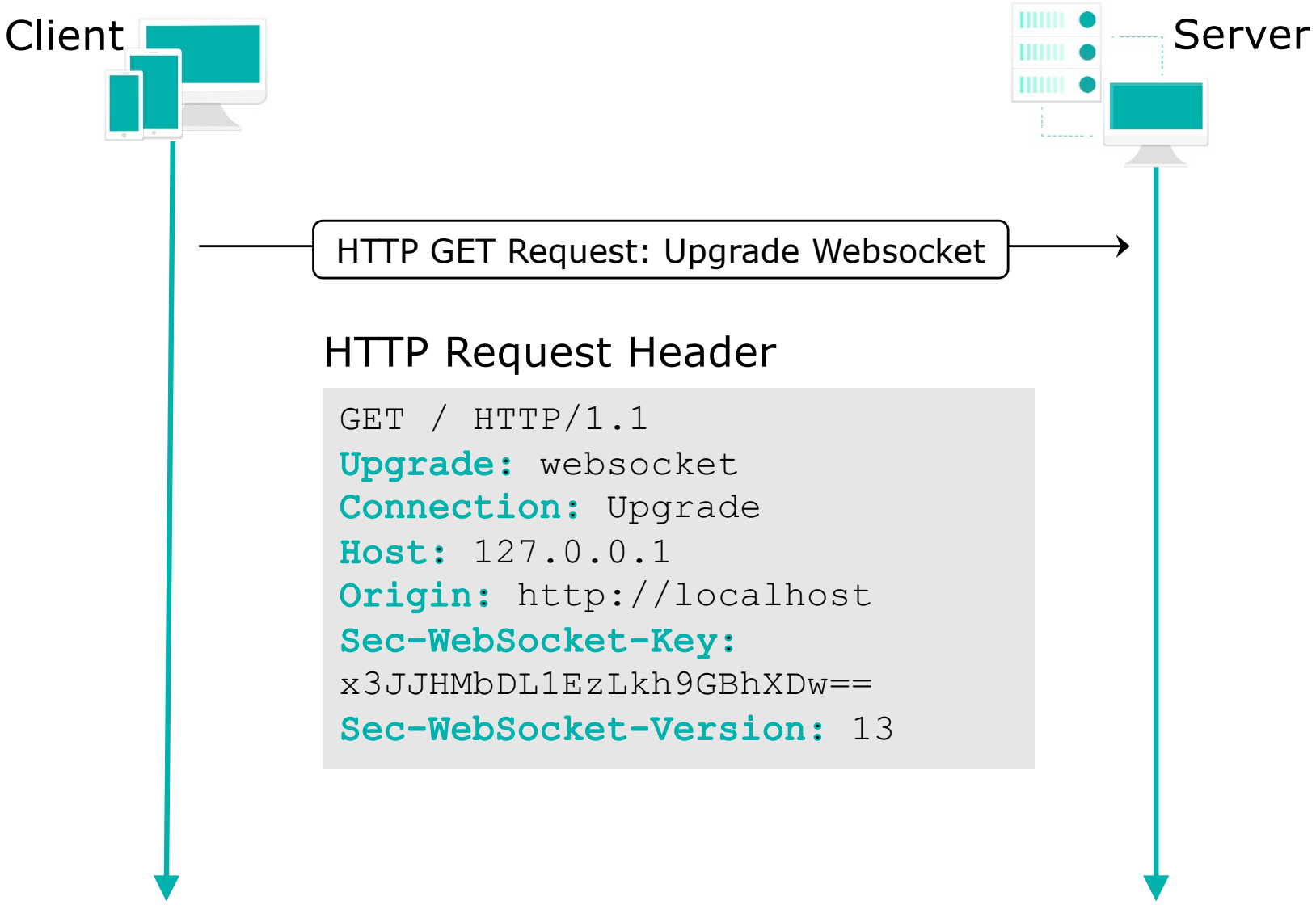
```
app = express();

// using express-session
app.use(require('express-session')({

    name: '_es_demo', // The name of the cookie
    secret: '1234', // The secret is required, and is used for signing cookies
    resave: false, // Force save of session for each request.
    saveUninitialized: false // Save a session that is new, but has not been modified

}));

// single path for root
app.get('/', function (req, res) {

    // simple count for the session
    if (!req.session.count) {
        req.session.count = 0;
    }
    req.session.count += 1;

    // respond with the session object
    res.json(req.session);

});
```

Wirkt nur, wenn das Session-Objekt nicht verändert wird. Dann wird es eben auch nicht gespeichert

Was sind Websockets?

- TCP basiertes Protokoll, welches eine bidirektionale Verbindung zwischen Webbrowser und Server ermöglicht
- nur **ein** Verbindungsaufbau notwendig
- TCP-Verbindung wird offengehalten
  - Im Gegensatz zu reinem HTTP: Serverantwort setzt Anfrage durch den Client voraus (Anfrage-Antwort-Prinzip)
- Anwendung: Echtzeitanwendungen
  - Chats
  - Spiele
  - Server-Monitoring
- wird von allen gängigen Browsern unterstützt

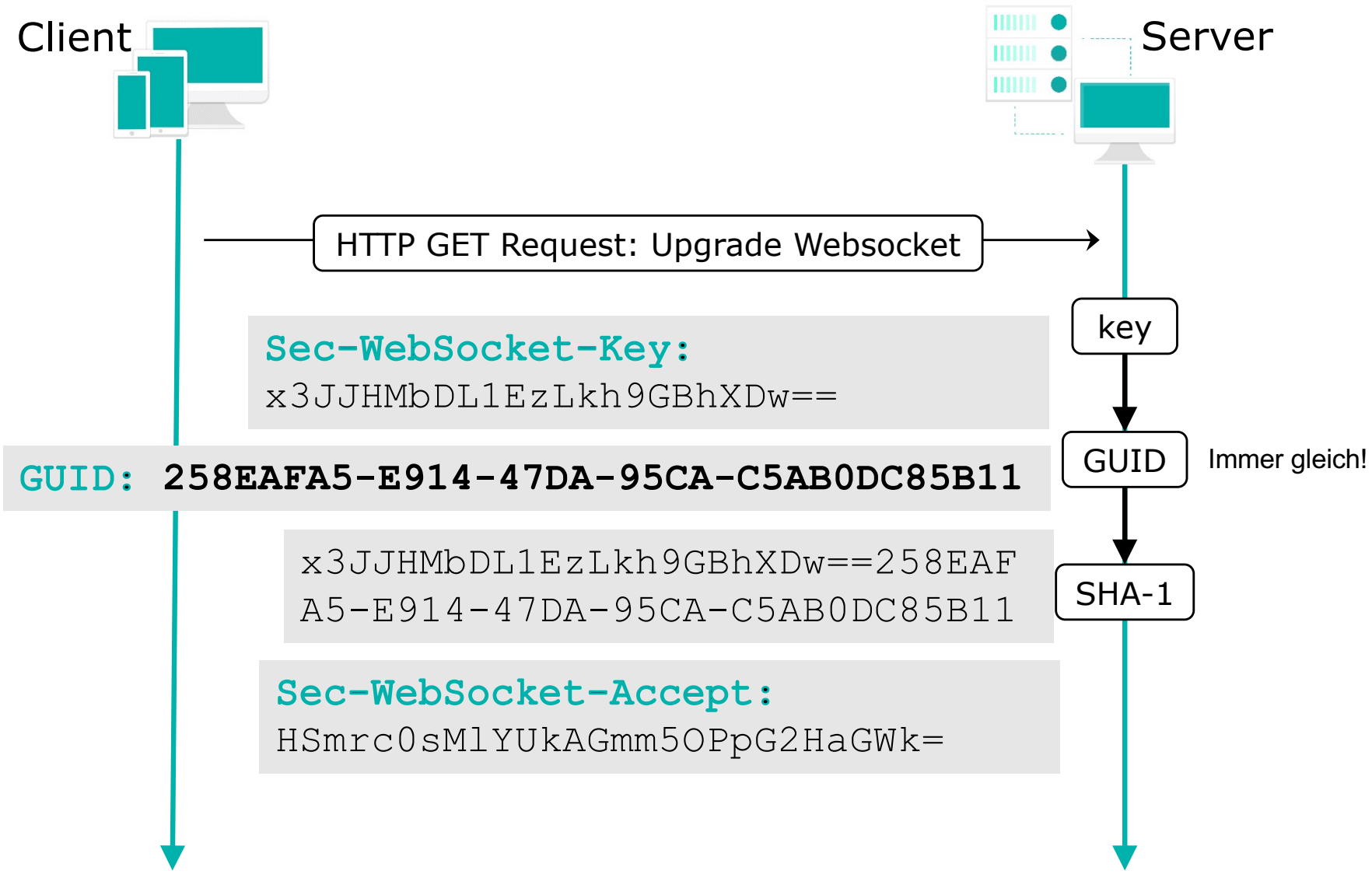
Node.js
Websockets Handshake



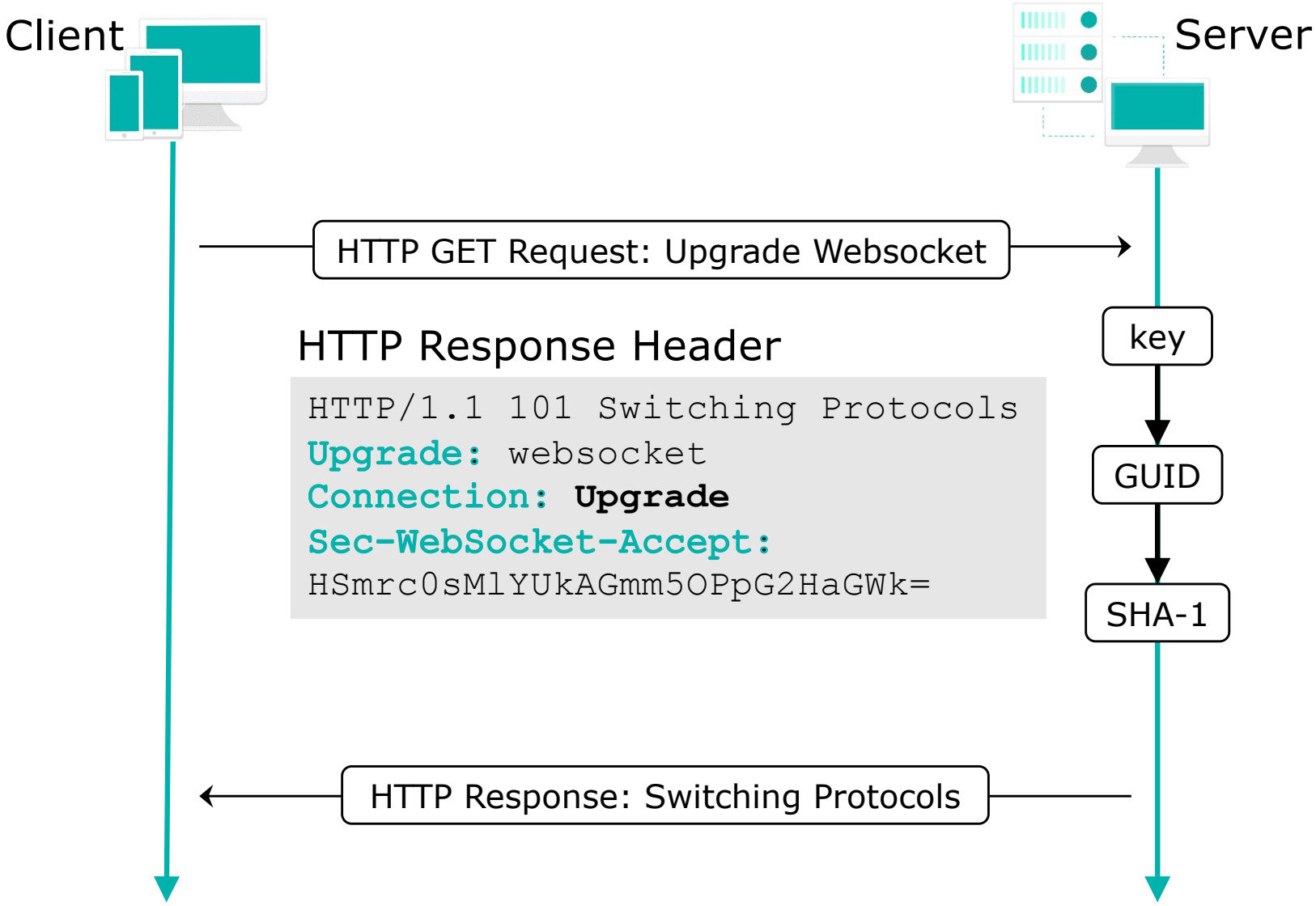
HTTP Request Header

```
GET / HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: 127.0.0.1
Origin: http://localhost
Sec-WebSocket-Key:
x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Version: 13
```

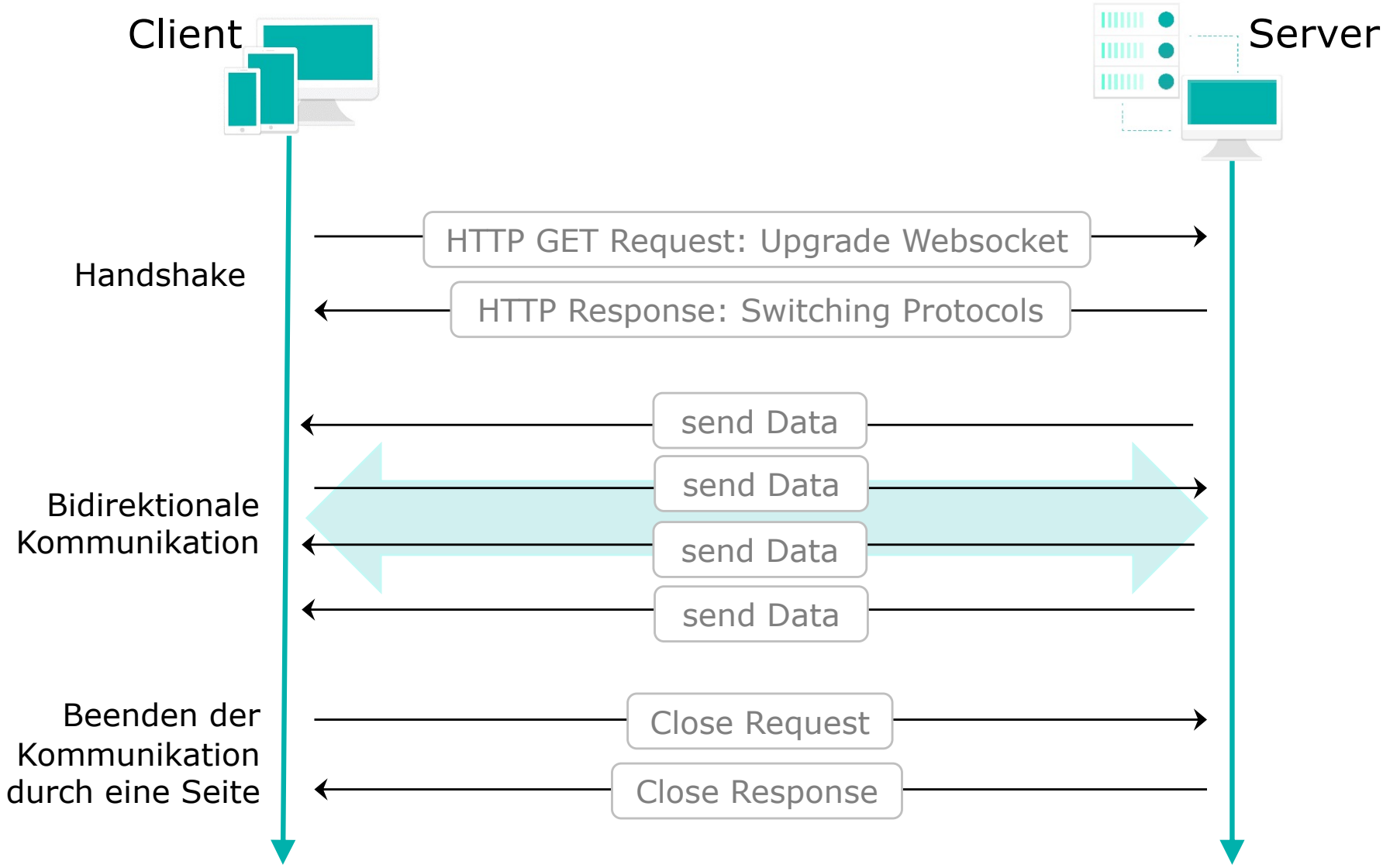
Node.js
Websockets Handshake



Node.js
Websockets Handshake



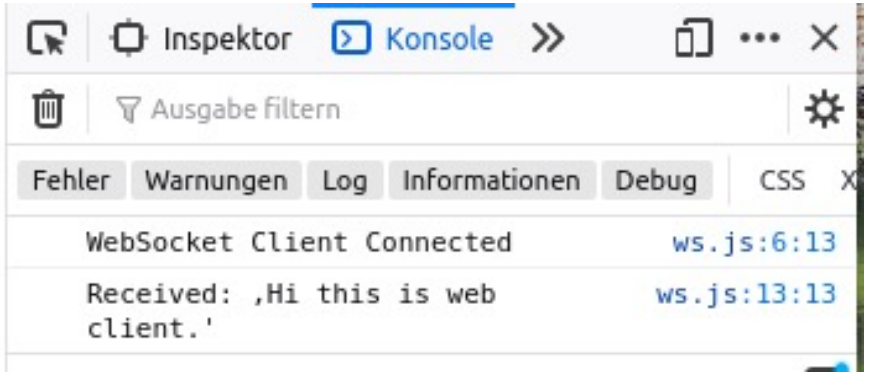
Node.js
Websockets Kommunikation



Index.html

```
<!DOCTYPE html>
<html>
<head>
  <title>WebSocket Playground</title>
</head>
<body>
  <p>Websocket Playground (Press F12)</p>
  <script src="ws.js"></script>
</body>
</html>
```

Websocket Playground (Press F12)



Node.js
Websockets - Client

ws.js (Client/Browser)

```
const socket = new WebSocket("ws://localhost:8080/");
//bei https muss wss verwendet werden

// Auf Verbindungsereigniss reagieren
socket.onopen = function () {
    console.log("WebSocket Client Connected");
    //senden von daten
    socket.send("Hi this is web client.");
}

// Auf Nachrichten reagieren
socket.onmessage = function (e) {
    console.log("Received: ," + e.data + "'");
};
```

Node.js
Websockets - Server

```
const express = require('express');
const ws = require('ws’);
// connectionHandler für Websocket in externer Datei
const connectionHandler = require('./websockets/connection.js');

const app = express();

// Einbinden der URL websockets
app.use(express.static(path.join(__dirname, 'websockets’));

app.get('/', (req, res) => {
    res.sendFile('index.html', { root: __dirname });
})

app.get('/ws.js', (req, res) => {
    res.sendFile('ws.js', { root: __dirname });
})
```

Node.js
Websockets - Server

```
// wir haben keinen eigenen Server für Websockets und nutzen express
const wsServer = new ws.Server({ noServer: true });

// Aktivierung des connectionHandler
wsServer.on('connection', connectionHandler);

// Starte unseren Server - > Liefert http-Server-Objekt!
const server = app.listen(8080);

// Der folgende Code ist immer gleich
// Wenn wir ein Upgrade erhalten, dann verarbeiten wir 3 Parameter
server.on('upgrade', (request, socket, head) =>
  // Führe Upgrade aus: Hierzu wird der registrierte Event
  // ‘connection’ verwendet
    { wsServer.handleUpgrade(request, socket, head,
        socket => { wsServer.emit('connection', socket, request); }
        )
  });
```

Node.js
Websockets - Server

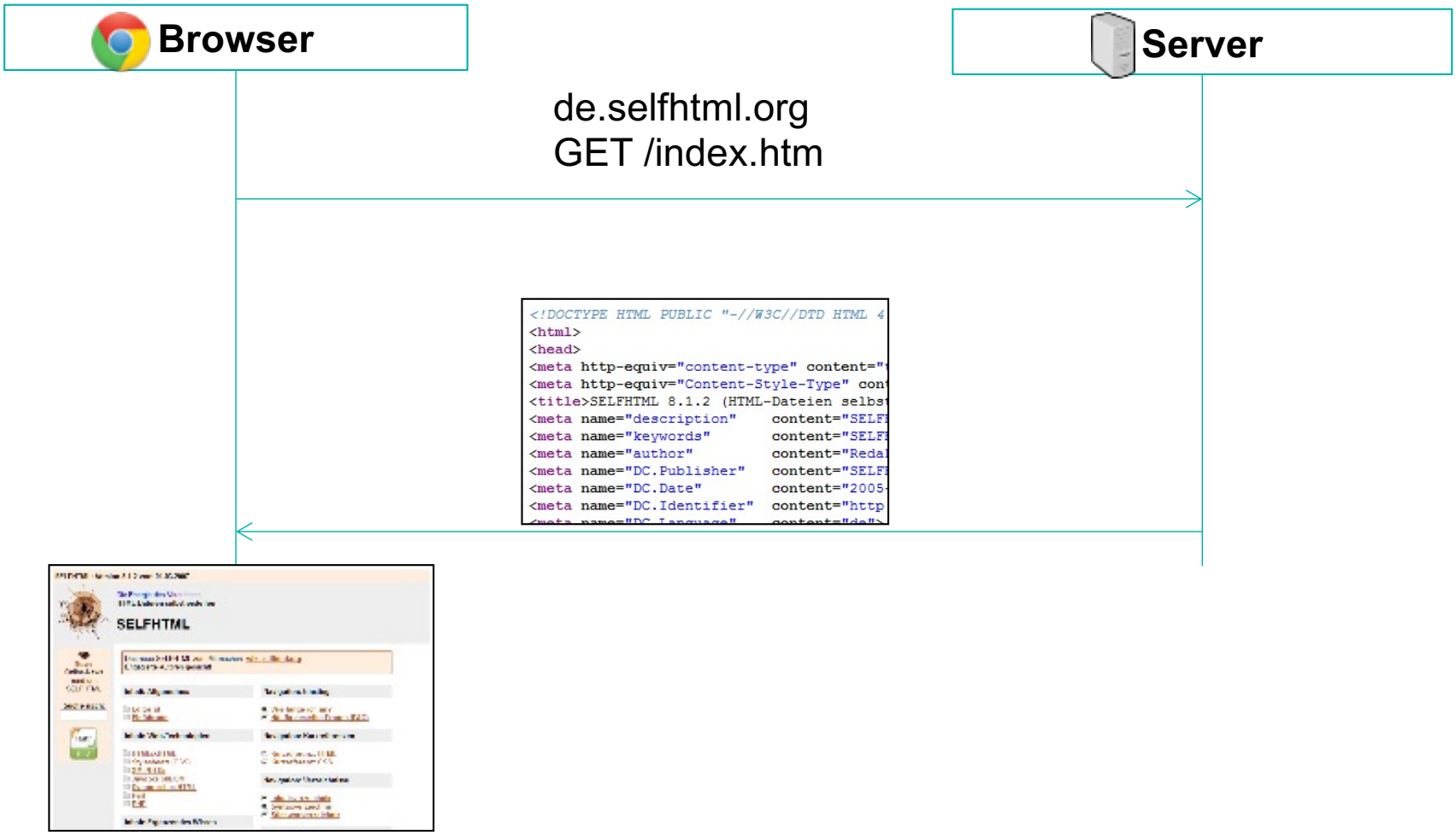
```
// './websockets/connection.js'
// Connection Handler (wird durch connection aufgerufen)

module.exports = function (socket, req) {

  console.log("Someone connected");

  socket.on('message', function (message) {
    console.log("Nachricht:" + message);
    //sendet Nachricht zurück
    socket.send(message);
  });

}
```

Dynamische Webseiten

https://nodejs.org/

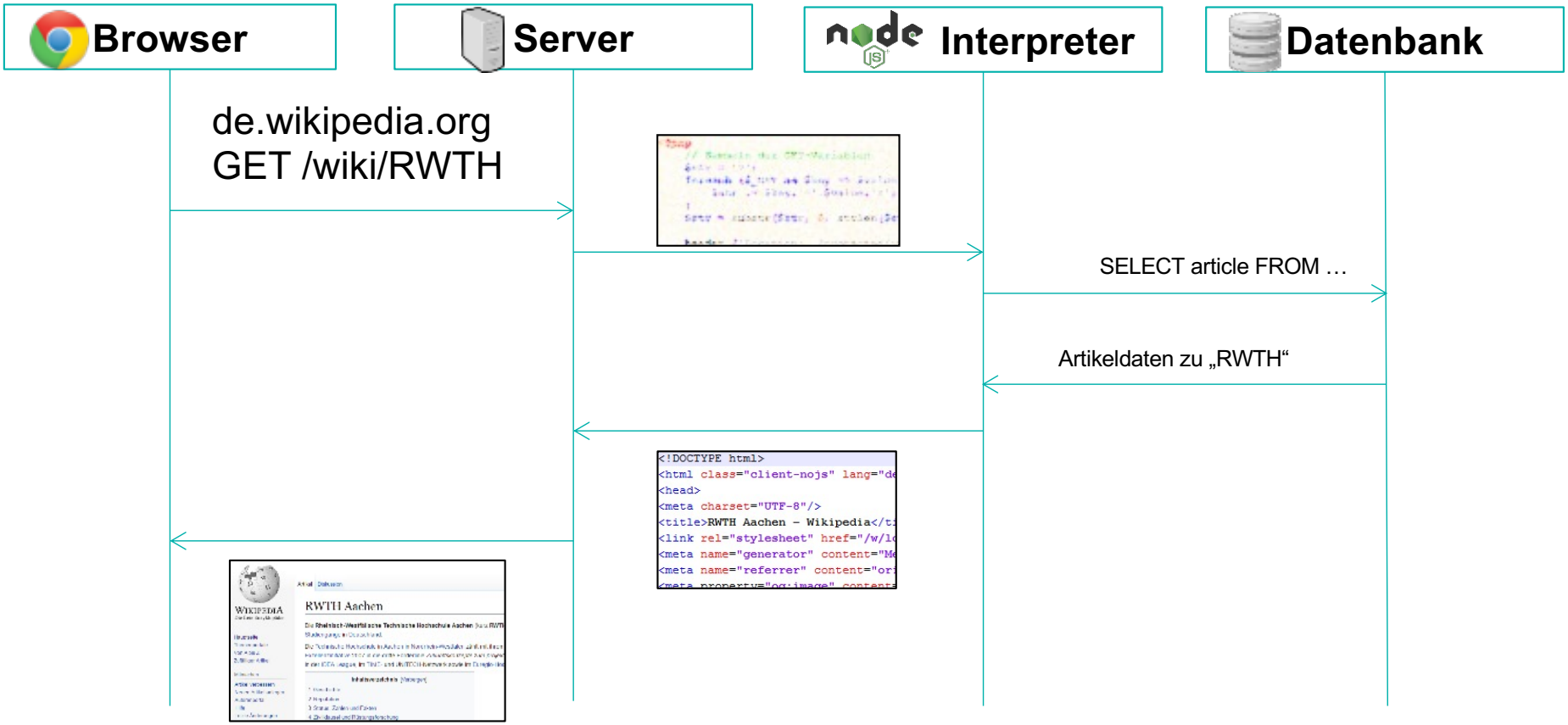
Dynamische Webseiten

Aufruf einer statischen Webseite (vereinfacht)



Dynamische Webseiten mit node.js

Aufruf einer dynamischen Webseite (vereinfacht)



Dynamische Webseiten
Verarbeitung von Formularen

Beispiel: Login-Formular

```
<form action="do?q=login" method="post">
    <input type="text" name="username" />
    <input type="password" name="pw" />
    <input type="submit" value="Login" />
</form>
```

- Übermittelt Parameter „username“ und „pw“ per POST-Request (hierbei werden die Daten urlencoded übergeben, z.B. durch Verwenden der **%xx-Notation** für Sonderzeichen, z.B. **%20 für +**

- Auswertung des GET-Parameters (q=login)
  - > Auf welcher Seite befinden wir uns?
- Auswertung des Formulars
  - > Ist „username“ und „pw“ korrekt?
  - > Evtl. per Datenbankabfrage herausfinden?

Dynamische Webseiten
Verarbeitung von Formularen

GET-Request

- Übertragung der Daten über die Adresszeile
- Parameter werden durch das Zeichen ? In der URL eingeleitet
- Parameter werden in express im Objekt `req.query` gespeichert
- Nicht geeignet zur Übertragung großer Datenmengen oder sensibler Daten

- Beispiel: `index.html?id=5&foo=bar`
  - > Zugriff im node.js-Skript:

```
req.query.id // 5
req.query.foo // bar
```

Dynamische Webseiten
Verarbeitung von Formularen

POST-Request

- Übertragung der Daten im HTTP-Body
  > Daten sind nicht in der URL sichtbar

- Zugriff auf POST-Daten in express:

```
req.body.text // liefert POST-Parameter text
```

Validierung der Nutzereingaben

- Beschränkungen des Clients können umgangen werden
- Beispiel:

```
<input type="text" name="plz" maxlength="5" />
```

- Parameter kann trotzdem mehr als 5 Zeichen haben
  - > User umgeht aktiv die Beschränkung
  - > Client unterstützt/versteht die Beschränkung nicht
- Serverseitige Prüfung von Nutzerdaten ist zwingend erforderlich

- **„Never trust the user“**

Check- und Filter-Funktionen

- Third Party Node.js Modul: express-validator
- Validierung der Eingaben

```
check([field, message])
```

> `field`: zu validierende (Array von) Strings
> `message`: eigens spezifizierte Fehlermeldung
> Rückgabe: Validation Chain
  - Hierbei handelt es sich um eine Middleware, die an die Route-Handler gegeben werden sollte

Dynamische Webseiten
Validierung von Eingaben

Ausgangssituation:

- Einfache Route die Formdaten verarbeiten soll
  - `name`: Der Name einer Person (String)
  - `email`: Eine gültige Mail Addresse nach RFC5322
  - `age`: Das Alter der Person als Zahl

Beispiel ohne Überprüfung:

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

const app = express()
app.use(express.urlencoded({extended: true}))
app.post('/form', (req, res) => {
    const name = req.body.name
    const email = req.body.email
    const age = req.body.age
})
```

Middleware zu Verarbeitung von POST (und PUT) Formulardaten, kodiert mit %xx für Sonderzeichen. Extended verwendet eine Bibliothek für komplexere Objekte (Arrays,...)
Ohne Path: Für alle URLs

Dynamische Webseiten
Validierung von Eingaben

Beispiel mit Überprüfung:

```
const express = require('express');
const {check, validationResult} = require('express-validator');
const app = express()

// Konvertiere Sonderzeichen
app.use(express.urlencoded({extended: true}))

function validationError(req, res, next) {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        // 422 Unprocessable Entity
        return res.status(422).json({errors: errors.array()})
    } else {
        next();
    }
}
// [...] nächste Folie
```

Dynamische Webseiten
Validierung von Eingaben

```
// [...] vorherige Folie

app.post('/form', [
    check('name').isLength({ min: 3 }),
    check('email').isEmail(),
    check('age').isNumeric(),
    validationError],
    (req, res) => {
        const name = req.body.name
        const email = req.body.email
        const age = req.body.age
     });
```

Das hier ist ein **Array von Middleware-Funktionen**, die ausgeführt werden, deshalb rufen wir in `validationError` `next` auch auf. Check liefert eine "validation-Chain" als Middleware, die damit faktisch filtert (und bei sanitize-Funktionen korrigiert)

Dynamische Webseiten
Bereinigung von Eingaben

- Bereinigung der Eingaben

```
sanitize*(fields).[one ore more sanitizer]
```

> `fields`: zu validierende (Array von) Strings
> *: **`Body`**, **`Cookie`**, **`Param`**, **`Query`**
> Middleware, die **`*`** manipuliert

```
buildSanitizeFunction(locations)
```

> `locations:` Request Locations (z.B. `['body', 'query']`)

Chaining wird verwendet, um weitere Sanitizer anzuwenden:
Verfügbare Sanitizer: https://github.com/validatorjs/validator.js#sanitizers

Beispiel

```
// escapes HTML entities and cuts whitespaces
app.post('/register',
    sanitizeBody('user').escape().trim(),
    (req, res, next) => {
       console.log(req.body.user);
       console.log(req.body.pw); // not sanitized
    }
);
```

Hinweis: Bei obigem escape() handelt es sich um das escape des Sanitizers und nicht etwa um das veraltete escape() als globales Objekt

(https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/escape)

bcryptjs
Passwortspeicherung

Sichere Speicherung von Passwörtern

- Wir wollen Passwörter nicht im Klartext speichern
  - > Gefahr, dass die Datenbank in falsche Hände gerät
  - > Nutzer verwenden oft das gleiche Passwort auf mehreren Seiten
- Besser: Hash-Werte speichern
- Third Party Node.js Modul: bcrypt.js
  - > `npm install bcryptjs`

- Beispiel:

```
console.log(bcrypt.hashSync('superPW123', salt))
// liefert z.B. $2y$10$dNvqygJOO4B/9HHS/aedre.BIdVclwmDK7d4tr0yxS4q5PRl0cRUG
// oder         $2y$10$haURtemZLULGRZatmmfeKuemcoUHunYpBUfE6EqZ6lnJW579gXa56
// oder         $2y$10$jNafet8iW5/NgmC85mJbde2ipWgpwvyuNA.YGYPxYxEUhA1rzxnzy
```

- Gleicher Funktionsaufruf liefert unterschiedliche Werte?
  - > Salt wird mit abgespeichert: Format

```
$[algorithm]$[cost]$[salt][hash]
```

bcryptjs
Passwortspeicherung

Passwort-API

- Einfache Möglichkeit Passwörter sicher zu speichern

```
bcrypt.hash(data, salt [, cb])
```

  > `data:` das Passwort
  > `salt:` entweder Salt oder Kostenfaktor: Anzahl (2^{salt}) von Runden mit dem der Hash erstellt wird, Salt wird in dem Fall automatisch generiert)
  > `cb:` optionaler Callback, der nach dem Hashen aufgerufen wird
  > Beispiel: `bcrypt.hash('superPassword', 10);`

- Überprüfung ob Passwort korrekt ist:

```
bcrypt.compare(data, encrypted [, cb])
```

  > `encrypted:` zu vergleichendes Passwort als hash
  > Beispiel: `bcrypt.compare('superPassword', 3ba21110b68aebffd636e0aa7776be7cf792e3cc21a8a82213220db5a490dd1f);`

bcryptjs
Passwortspeicherung

Achtung

- Bei `bcrypt.hash` und `bcrypt.compare` handelt es sich um **asynchrone** Funktionen
- D.h. diesen sollte in Kombination mit `await` in einer mit mit `async` notierten Funktion aufgerufen werden

Alternative

- Synchrone Variante:
  - > `bcrypt.hashSync`
  - > `bcrypt.compareSync`

Zu bevorzugen ist die asynchrone Variante, damit die Event Loop nicht blockiert wird!

Siehe dazu auch: https://jinoantony.com/blog/async-vs-sync-nodejs-a-simple-benchmark

Schädliche Module

- jeder kann Module per `npm` veröffentlichen und konsumieren
- dabei können schädliche Module in Umlauf geraten
- Gefahr besteht, dass man diese versehentlich per `npm` installiert
- Beispiel **Typosquatting**: `socketio` anstelle von `socket.io`
  - `socket.io` ist das verbreitetste Modul für Echtzeit-Webanwendungen
  - `socketio` (ähnlich benannt) → könnte Schadcode enthalten
- auch bekannte Module können verwundbar sein (z.B. durch ein Update)
  - Empfehlung: Version des Moduls in der `package.json` fixieren
  - `npm-audit` nutzen: Projekt nach Verwundbarkeiten durchsuchen

BE CAREFUL!