

PRINZIPIEN DER SOFTWARETECHNIK

Die Softwaretechnik bietet bewährte Prinzipien, mit deren Hilfe wir die Komplexität eines Softwaresystems beherrschen können, z.B.:

1. Modularisierung
2. Trennung von Zuständigkeiten

MODULARISIERUNG

Modularisierung: Die Aufteilung eines Softwaresystems in seine Komponenten

*Eine **Komponente** ist eine in sich geschlossene Einheit, die “Dritten Funktionalität über Schnittstellen zur Verfügung stellt und unabhängig ausgeliefert werden kann.”*

Nach: Balzert H.; Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb; Springer Spektrum, 2011

TRENNUNG VON ZUSTÄNDIGKEITEN

- Englisch *Separation of Concerns* (SoC)
- Jede Komponente eines Softwaresystems hat **eine** klar festgelegte Zuständigkeit
- Vorteile u.A.:
 - ➕ Klarere Organisation und Struktur des Softwaresystems
 - ➕ Verbesserte Änderbarkeit und Austauschbarkeit einzelner Aspekte des Softwaresystems
 - ➕ Bessere Lokalisierung von Fehlern

TRENNUNG VON ZUSTÄNDIGKEITEN (2)

Typische zu trennende Aspekte:

- Technik ↔ Fachlichkeit
(z.B. Aufbau einer Datenbankverbindung → Aufsummierung von Rechnungspositionen)
- Oberflächenlogik ↔ Fachlogik
(z.B. Aktivieren/Deaktivieren des "Bestellen"-Buttons → Ausführung einer Bestellung)

Welche Beispiele für SoC haben wir bereits kennengelernt?

- HTML ↔ CSS ↔ JavaScript
- TCP/IP-Protokollstack



Zentrale Eigenschaften von Node.js:

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

MODULARISIERUNG MIT JAVASCRIPT

- ! Der ECMAScript-Standard sah lange (< ES6) keinerlei Möglichkeiten zur Modularisierung vor
- Es bestand jedoch ein großer Bedarf nach einer solchen Möglichkeit:
 - Durch die steigende Popularität von JavaScript entstanden immer komplexere Projekte (= große Code-Basis)
 - Da es nur globale Sichtbarkeit oder Funktionssichtbarkeit von Variablen gab (< ES6), war eine saubere Datenkapselung schwierig zu erreichen

MODULARISIERUNG MIT JAVASCRIPT (2)

- Mit reinem JavaScript war es höchstens möglich, Module eingeschränkt "nachzuahmen" (z.B. über die Ausnutzung der Funktionssichtbarkeit)
- Daher etablierten sich hier alternative Ansätze, die die fehlende Modularisierung über Bibliotheken ergänzen

ANSÄTZE ZUR MODULARISIERUNG

Beispiele:

Asynchronous Module Definition [↗](#) (AMD)

- Hauptsächlich clientseitiger Einsatz
- Bekannteste Bibliothek: [RequireJS](#) [↗](#)

CommonJS [↗](#)

- Serverseitiger Einsatz
- Insbesondere auch von Node.js eingesetzt

MODULE IN NODE.JS

- CommonJS definiert ein API zur Erstellung von Modulen
- Das Modulsystem von Node.js basiert auf CommonJS:
 - Ein Modul wird in einer separaten Datei definiert
 - *Datenkapselung*: Alle in einem Modul definierten Variablen sind auch nur innerhalb des Moduls sichtbar
 - Über das **Objekt `module.exports`** können Eigenschaften (z.B. Funktionen) gezielt sichtbar gemacht werden → Damit wird die *Schnittstelle* des Moduls definiert
 - Über die `require`-Funktion (diese haben wir bereits kennengelernt!) kann das Modul dann an anderer Stelle eingebunden und verwendet werden

MODULE IN NODE.JS: BEISPIEL


Datei wuerfel.js:

```
const getRandomNumber = function(min, max) {  
  return Math.floor(Math.random() * (max - min + 1) + min);  
};  
  
// Gibt eine zufällige Zahl zwischen 1 und 6 (inklusive) zurück  
const rollTheDice = function() {  
  return getRandomNumber(1, 6);  
};  
  
// Die Funktion "rollTheDice" wird unter dem Namen "wuerfle" Teil der öffentlichen  
// Schnittstelle des Moduls  
module.exports.wuerfle = rollTheDice;
```

Datei app.js:

```
// Einbinden des "wuerfel"-Moduls: "../wuerfel.js" ist der relative Dateipfad, unter  
// welchem das Modul gesucht wird  
const wuerfel = require("../wuerfel.js");  
  
// Ausgabe: Eine Zahl zwischen 1 und 6  
console.log(wuerfel.wuerfle());  
  
// Ausgabe: "TypeError: wuerfel.getRandomNumber is not a function".  
// Grund: getRandomNumber ist nicht Teil der öffentlichen Schnittstelle des Moduls  
console.log(wuerfel.getRandomNumber(1, 1000));
```

MODULARISIERUNG MIT ES6+

- Mittlerweile (\geq ES6) enthält der ECMAScript-Standard auch ein eigenes Modulsystem
- ! Unterstützung in Node.js aktuell (Stand 01/2019) jedoch lediglich [experimentell](#) 

Aufgabe:

Betrachten Sie die bisherigen Code-Beispiele. Welche Nachteile bzw. Probleme sehen Sie, wenn auf diese Weise komplexere Web-Anwendungen mit dem "http"-Modul entwickelt werden sollen?

- Gesamter Code in lediglich einer Datei
- Keine Trennung von Verantwortlichkeiten (Vermischung von Fachlogik, Routing, HTML-Templates, etc)
- Programmierung auf einem geringen Abstraktionsniveau (sehr "low-level", viel muss manuell programmiert werden)
- ! Für komplexere Web-Anwendungen sollte ein *Web-Framework* verwendet werden!



Über Module lösbar

(SERVERSEITIGE) WEB-FRAMEWORKS


- Liefern einen Rahmen mit Basisfunktionen für wiederkehrende Aufgaben, z.B.:
 - Daten aus der Anfrage lesen
 - Die Anfrage auswerten und entsprechende Aktionen ausführen (*Routing*)
 - Datei-Uploads
 - Authentifizierung
- ➕ EntwicklerInnen müssen nicht "auf der grünen Wiese" beginnen → höhere Geschwindigkeit
- ➕ Standardfehler werden vermieden (Sicherheitslücken etc.)



EXPRESS

- <https://expressjs.com/> 
- Einfaches Web-Framework für Node.js
- Baut auf dem "http"-Modul auf
- Verfügbar als *Paket* per *npm*:
<https://www.npmjs.com/package/express> 

NPM

- npm ist ein Paketmanager, welcher in der Installation von Node.js enthalten ist
- Ursprünglich zur Verteilung von Node.js-Modulen entwickelt* - mittlerweile ein genereller Paketmanager für JavaScript-Anwendungen
- Verfügbare Pakete werden in der *npm-Registry* unter <https://npmjs.com>  verwaltet

* Ursprüngliche Bedeutung des Akronym: **N**ode **P**ackage **M**anager

PAKETE

- Dienen der Verteilung und Wiederverwendung von Code
- Können eine oder mehrere Module, aber auch ganze Bibliotheken, Frameworks oder Anwendungen enthalten
- Technisch wird ein Paket durch eine `package.json`-Datei beschrieben

PACKAGE.JSON

Beispiel:

```
{  
  "name": "exampleapp",  
  "version": "1.0.0",  
  "description": "Beispiel fuer  
                  ein Paket",  
  "main": "app.js",  
  "author": "Jörg Svensson",  
  "license": "MIT",  
  "dependencies": {  
    "ejs": "^2.6.1",  
    "express": "^4.16.4"  
  }  
}
```

- Die `package.json`-Datei enthält Metadaten des Pakets wie z.B. Name, Version, Autor und Lizenz
- Unter dem Schlüssel `dependencies` werden die Abhängigkeiten des Pakets aufgelistet
- Abhängigkeiten = andere Pakete, die das Paket benötigt (im Beispiel: `ejs` und `express`)

NPM-WERKZEUG

npm bietet ein Kommandozeilenwerkzeug, das bei verschiedenen Aufgaben unterstützt, z.B.:

Befehl	Beschreibung
<code>npm init</code>	Interaktives Erstellen einer <code>package.json</code> -Datei
<code>npm install <Paketname></code>	<ul style="list-style-type: none">• Lädt das Paket <code><Paketname></code> sowie dessen benötigte Pakete herunter und speichert diese im Verzeichnis <code>node_modules</code>• Mit der Option <code>--save</code> wird das Paket in den <code>dependencies</code>-Abschnitt einer vorhandenen <code>package.json</code>-Datei eingetragen
<code>npm list</code>	Zeigt die installierten Pakete an
<code>npm update</code>	Aktualisiert die in der <code>package.json</code> gelisteten Pakete (und deren Abhängigkeiten)

BEISPIEL: EXPRESS INSTALLIEREN

In folgendem Beispiel verwenden wir npm, um ein Projekt für eine Express-Anwendung einzurichten:

1. Ein Projektverzeichnis `myApp` erstellen und dieses auf der Konsole öffnen
2. `package.json`-Datei erzeugen (interaktiv - entsprechende Fragen beantworten):

```
$ npm init
```

3. Express installieren und als Abhängigkeit hinzufügen:

```
$ npm install --save express
```

EXPRESS EINBINDEN UND INITIALISIEREN

- Nach der Installation kann Express mittels `require("express")` eingebunden werden
- Das Modul "express" exportiert eine Funktion, durch deren Aufruf das Framework initialisiert wird (mit einer Standardkonfiguration)
- Der Server kann dann analog zum "http"-Modul mit der `listen`-Funktion gebunden werden

Datei `app.js`:

```
// Einbinden des "express"-Moduls
const express = require("express");
// Initialisieren von Express
const app = express();

// Server an Port 8023 binden --> Die "listen"-Funktion von Express delegiert
// einfach an die bekannte "listen"-Methode aus dem "http"-Modul
app.listen(8023, function() {
  console.log("Server lauscht auf http://localhost:8023");
});
```

Start der Anwendung wie bisher über das Kommando: `node app.js`

EXPRESS: GRUNDKONZEPT

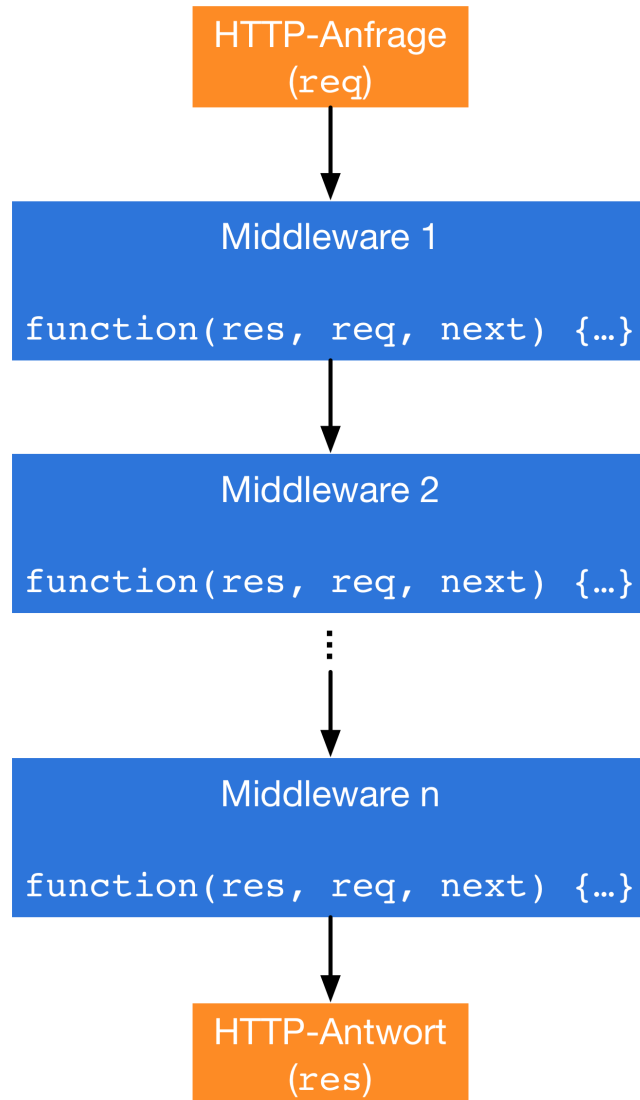
HTTP-Anfrage
(req)

Wie beim Request-Listener des "http"-Moduls hat Express bei einer eintreffenden Anfrage Zugriff auf:

- Ein Request-Objekt (`req`), das die eingetroffene Anfrage repräsentiert
- Ein Response-Objekt (`res`), das die Antwort des Servers repräsentiert

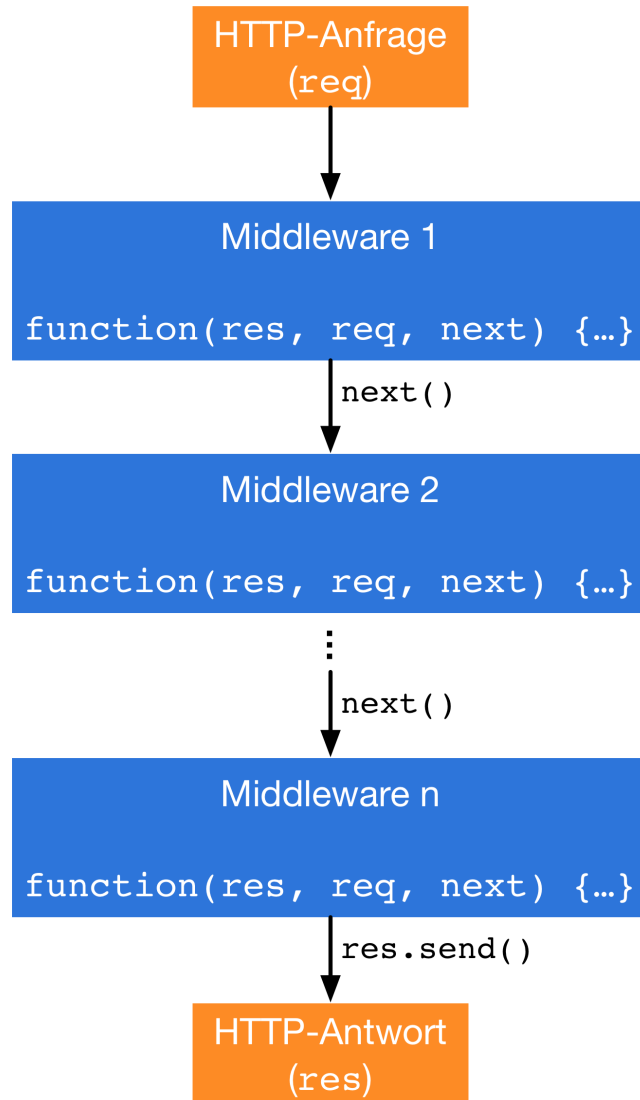
HTTP-Antwort
(res)

EXPRESS: GRUNDKONZEPT



- Die beiden Objekte werden nun durch eine Kette von *Middlewares* geleitet
- Eine Middleware ist eine Funktion mit den Parametern `req`, `res` und `next`
- `next` ist jeweils die nächste Middleware-Funktion in der Kette
- Middlewares werden über die `app.use`-Funktion registriert - die Abarbeitung erfolgt in der Reihenfolge der Registrierung

EXPRESS: GRUNDKONZEPT



Jede Middleware in der Kette kann nun:

- Anfrage und Antwort verarbeiten/modifizieren
- Durch Aufruf von `next()` an die nächste Middleware weiterleiten
- Durch Senden der Antwort den Durchlauf durch die Kette beenden (nachfolgende Middlewares kommen dann nicht mehr zum Zuge)

MIDDLEWARE: BEISPIEL

Datei app.js:

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

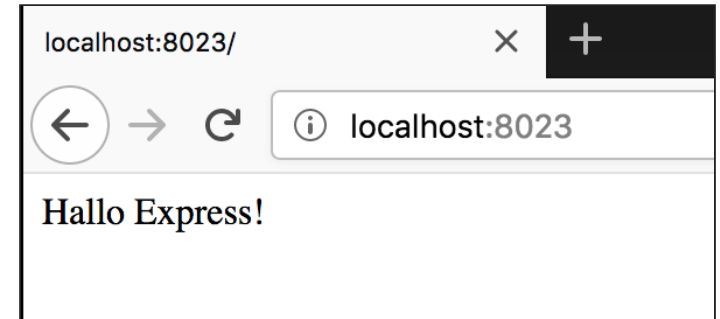
// Middleware-Funktionen werden über die
// "use"-Funktion eingebunden
app.use(function(req, res, next) {
  console.log("Erste Middleware!");
  next(); // Aufruf der nächsten Middleware
});

app.use(function(req, res, next) {
  console.log("Zweite Middleware!");
  // Senden der Antwort - Kette beenden
  res.send("Hallo Express!");
});

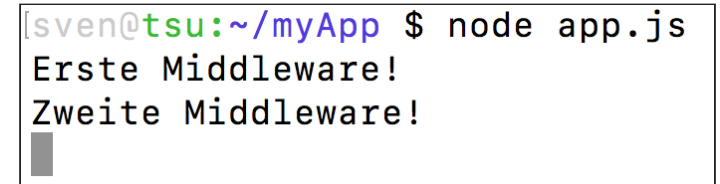
// Diese Middleware wird nicht mehr aufgerufen
app.use(function(req, res, next) {
  console.log("Dritte Middleware!");
});

app.listen(8023);
```

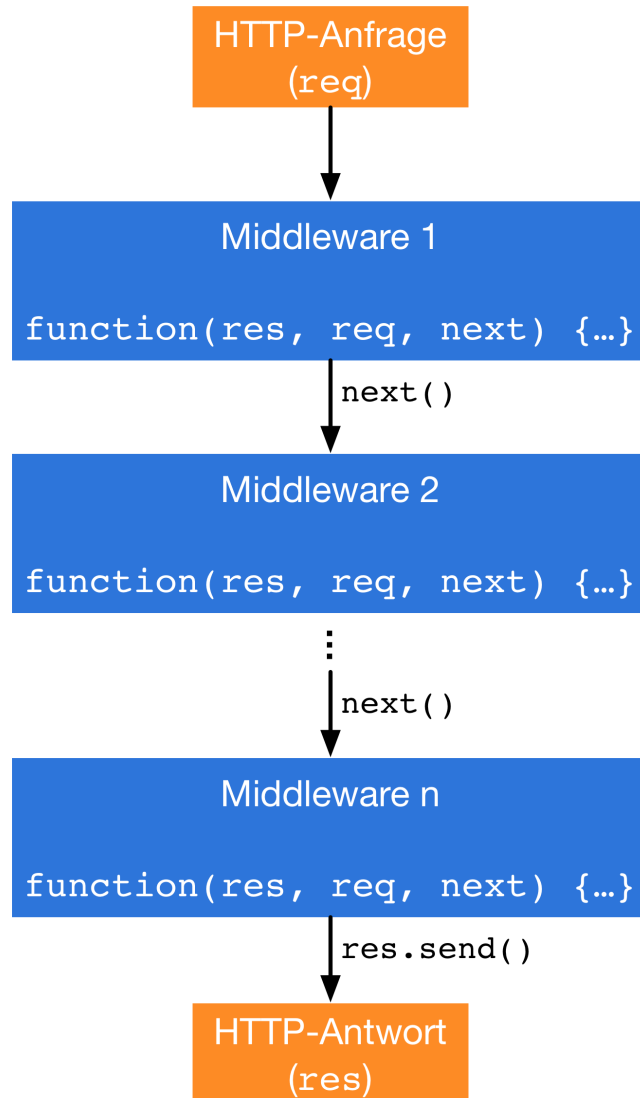
Ausgabe im Browser:



Ausgabe auf der Konsole:



EXPRESS: GRUNDKONZEPT



- Diese Struktur folgt dem allgemeinen Entwurfsmuster "Zuständigkeitskette" (*Chain of Responsibility*), welches häufig in Web-Frameworks zu finden ist
- Ziel: Flexibler Aufbau einer Web-Anwendung durch Kombination von Middlewares
- Es gibt viele **vorgefertigte Middlewares** [↗](#) für verschiedene Zwecke

BEISPIEL: AUSLIEFERN STATISCHER DATEIEN

- Zum Ausliefern statischer Dateien (z.B. CSS, Bilder, statische HTML-Dateien) liefert Express eine Middleware direkt mit
- Diese Middleware wird durch die Funktion `express.static` [↗](#) realisiert, die über `app.use` registriert wird
- Als Parameter erwartet `express.static` ein Verzeichnis
- Wird über die URL der Anfrage eine Datei angefordert, so sucht `express.static` diese Datei im gegebenen Verzeichnis
- Wird eine entsprechende Datei gefunden, so wird diese ausgeliefert, ansonsten wird per `next()` die nächste Middleware aufgerufen

BEISPIEL: AUSLIEFERN STATISCHER DATEIEN (2)

Datei `app.js`:

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

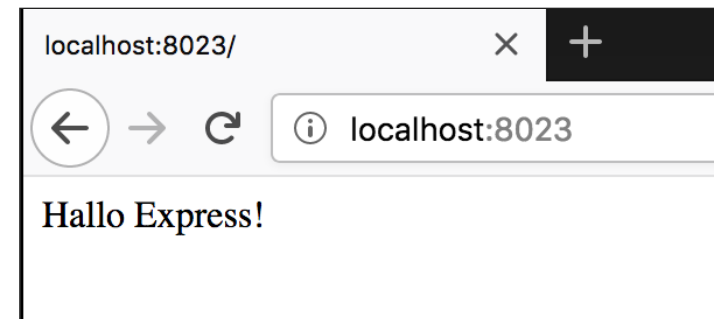
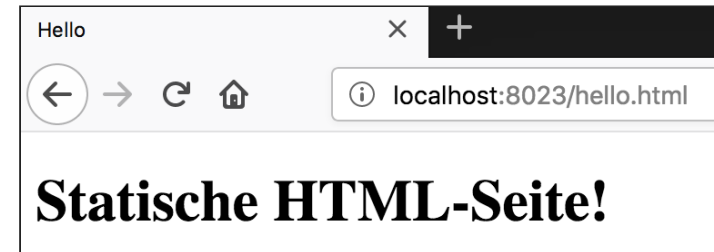
// "express.static" als erste Middleware einbinden
// --> Statische Dateien sollen im Verzeichnis
//      "public" gesucht werden
app.use(express.static("public"));

// Nur falls keine statische Datei ausgeliefert
// wurde, wird die nächste Middleware in der
// Kette aufgerufen
app.use(function(req, res, next) {
  // Senden der Antwort - Kette beenden
  res.send("Hallo Express!");
});

app.listen(8023);
```

! Annahme: Im Verzeichnis `public` liegt eine statische HTML-Datei `hello.html`

Ausgaben im Browser:



ROUTING VON ANFRAGEN

Erinnerung: Beim "http"-Modul erfolgte das Routing z.B. durch Auswertung der URL und der HTTP-Methode im Request-Objekt:

```
const url = request.url;
const method = request.method;

if (url === "/" ) {
    // Antwort für Zugriff auf URL "/" erzeugen
    [...]
} else if (url.startsWith("/new") && method === "GET") {
    // Antwort für Zugriff auf URL "/new" mit Methode GET erzeugen
    [...]
} else if (url.startsWith("/new") && method === "POST") {
    // Antwort für Zugriff auf URL "/new" mit Methode POST erzeugen
    [...]
}
```

ROUTING VON ANFRAGEN MIT EXPRESS

Über die URL der Anfrage:

- Eine Middleware kann beim Registrieren über `app.use` zusätzlich einer URL zugewiesen werden:

```
app.use(URL, Middleware-Funktion);
```

- Die Middleware wird dann nur ausgeführt, wenn die URL in der HTTP-Anfrage der angegebenen *URL* entspricht
- Wird der Parameter *URL* nicht angegeben (wie in unseren bisherigen Beispielen), ist der Standardwert `" / "` ("alle URLs")

ROUTING VON ANFRAGEN MIT EXPRESS

(2)

Über die HTTP-Methode der Anfrage:

- Zu `app.use` existieren Varianten, die jeweils auf eine HTTP-Methode einschränken, z.B.:

```
// Einschränkung auf GET-Anfragen  
app.get(URL, Middleware-Funktion);  
// Einschränkung auf POST-Anfragen  
app.post(URL, Middleware-Funktion);  
// Einschränkung auf PUT-Anfragen  
app.put(URL, Middleware-Funktion);
```

- Die Varianten unterstützen die gleichen Parameter wie `app.use`

BEISPIEL: EINFACHES ROUTING MIT EXPRESS

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

app.get("/new", function(req, res, next) {
  // Antwort für Zugriff auf URL "/new" mit Methode GET erzeugen
  [...]
});

app.post("/new", function(req, res, next) {
  // Antwort für Zugriff auf URL "/new" mit Methode POST erzeugen
  [...]
});





app.use(function(req, res, next) {
  // Antwort für Zugriff alle anderen URLs erzeugen
  [...]
});

app.listen(8023);
```


ANFRAGE- UND ANTWORT-OBJEKT

Das Request- und das Response-Objekt von Express erweitern die entsprechenden Objekte aus dem "http"-Modul um zusätzliche Eigenschaften und Funktionen, z.B.:

Response

<code>redirect</code> 	Leitet an eine gegebene URL weiter. Parameter: <ul style="list-style-type: none">• <code>status</code> (optional): Statuscode für die Weiterleitung• <code>path</code>: URL, an die weitergeleitet werden soll
<code>send</code> 	Sendet die HTTP-Antwort. Einziger Parameter ist <code>body</code> - der zu sendende Inhalt der Antwort (der passende <code>Content-Type</code> -Header wird automatisch gesetzt).
<code>status</code> 	Setzt den Statuscode der HTTP-Antwort.
<code>render</code> 	Erzeugt eine Ansicht (<i>View</i>) und sendet diese zum Client.*

* Dazu später mehr!

ANFRAGE- UND ANTWORT-OBJEKT

Das Request- und das Response-Objekt von Express erweitern die entsprechenden Objekte aus dem "http"-Modul um zusätzliche Eigenschaften und Funktionen, z.B.:

Request

`body` 

Zugriff auf die Daten im Body der HTTP-Anfrage*

`query` 

- Zugriff auf die Daten im "Query"-Teil der URL
- Die Daten sind in Form eines JavaScript-Objektes zugreifbar
- Beispiel: Für die Anfrage `GET /todo.html?title=Lernen` liefert `req.query.title` den Wert `Lernen`

* Dazu später mehr!

ANFRAGE- UND ANTWORT-OBJEKT

Im Folgenden betrachten wir folgende Aspekte näher:

1. `Request.render`:

Erzeugt eine Ansicht (*View*) und sendet diese zum Client.

2. `Response.body`:

Zugriff auf die Daten im Body der HTTP-Anfrage

ERZEUGEN VON ANSICHTEN (VIEWS)

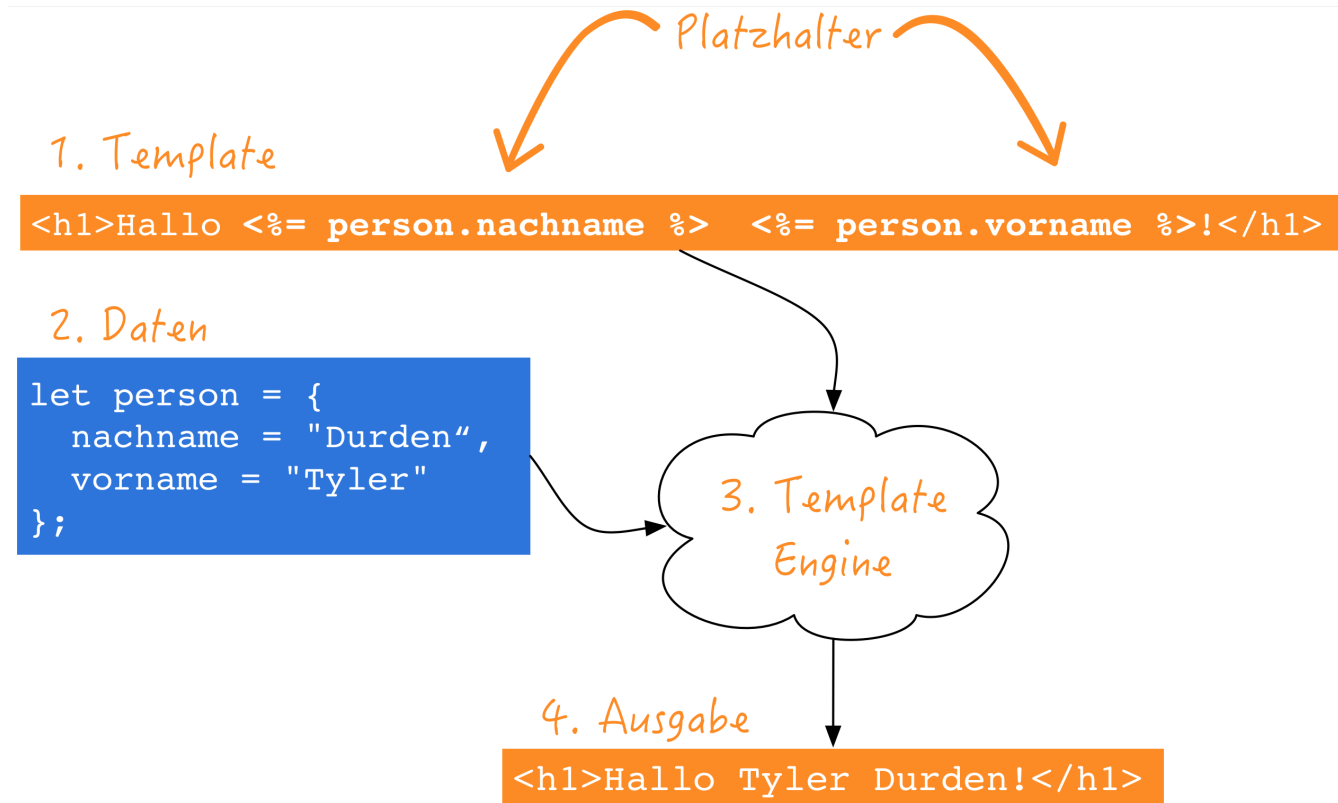
In unseren bisherigen Beispielen haben wir die Template-Literale von JavaScript verwendet, um HTML-Seiten (dynamisch) zu erzeugen

Welche Nachteile hat dieser Ansatz?

- Gefahr der Vermischung von Zuständigkeiten (z.B. Darstellung und Applikationslogik)
- Schlechte Unterstützung durch Entwicklungsumgebungen
- Ggf. schlechtere Lesbarkeit und Wartbarkeit des Codes

TEMPLATE-ENGINE

Um hier eine bessere Trennung von Zuständigkeiten zu erreichen, nutzen Web-Frameworks typischerweise *Template-Engines*



TEMPLATE-ENGINES: FUNKTIONSWEISE

- Analogie: Word-Serienbrief
- Die zu erzeugenden Textdateien liegen als *Template* (Vorlage) vor
- Ein Template enthält typischerweise statischen Text (z.B. HTML-Code) und Platzhalter für die dynamischen Anteile
- Zur Laufzeit ersetzt die Template-Engine die Platzhalter durch konkret vorliegende Daten

TEMPLATE-ENGINES IN EXPRESS

- Express unterstützt verschiedene Template-Engines, z.B.:
 - [EJS](#)
 - [Pug](#)
 - [Handlebars](#)
- Diese unterscheiden sich insbesondere in:
 - der Syntax der Templates, und
 - dem Sprachumfang der Template-Sprache (z.B. nur einfache Platzhalter, oder auch komplexere Ausdrücke mit Kontrollstrukturen, Verzweigungen, etc.).

BEISPIEL: EJS

- Simple Template-Engine zur Erzeugung von HTML
- Die Template-Sprache von EJS verwendet JavaScript für die dynamischen Anteile
- EJS installieren und als Abhängigkeit zu einer bestehenden `package.json` hinzufügen:

```
$ npm install --save ejs
```

- EJS in eine Express-Anwendung einbinden:

```
// EJS als Template-Engine konfigurieren
app.set("view engine", "ejs");
// Verzeichnis, in welchem EJS die Templates sucht, festlegen
// (hier: "views")
app.set("views", "views");
```


EJS-TEMPLATES

- EJS-Templates werden in Dateien mit der Endung `.ejs` abgelegt
- Die statischen Teile eines EJS-Templates sind normaler HTML-Code
- Die dynamischen Teile werden mit speziellen Tags markiert und enthalten JavaScript-Code, z.B.:

`<%= ausdruck %>`

Schreibt den Wert des JavaScript-Ausdrucks *ausdruck* in den resultierenden HTML-Code.

`<% ausdruck %>`

Erzeugt keine Ausgabe im HTML-Code, kann jedoch bedingte Anweisungen und Schleifen enthalten, die dann bei Auswertung des Templates ausgeführt werden.

EJS-TEMPLATE: BEISPIEL

Datei `views/hello.ejs`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello Express</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <h1>Hello-App</h1>
    <!-- Bedingte Anweisung: Enthaltener HTML-Code (ul) landet
         nur im Ergebnis wenn der Wert der Variable "name" truthy ist -->
    <% if (name) { %>
      <ul>
        <!-- Schleife: Für jedes Element im Array "greetings" wird ein li-Element erzeugt -->
        <% for (greeting of greetings) { %>
          <!-- Im li-Element wird der aktuelle Wert von "greeting" und "name" ausgegeben -->
          <li><%= greeting %>, <%= name %>!</li>
        <% } %>
      </ul>
    <% } else { %>
      <!-- Bedingte Anweisung: Ausgabe falls der Wert der Variable "name" falsy ist -->
      Kein Name angegeben!
    <% } %>
  </body>
</html>
```

HTML-SEITE ERZEUGEN

- Das Erzeugen einer konkreten HTML-Seite mit EJS kann über die `render`-Funktion des `Response`-Objekts ausgelöst werden
- Die Funktion benötigt dazu zwei Argumente:
 - `view`: Der Dateiname des EJS-Templates (ohne `.ejs`-Endung)
 - `locals`: Ein Objekt, welches sämtliche Daten enthält, die das Template zur Auswertung benötigt (z.B. Variablen, deren Werte ausgegeben werden sollen)
- `render` schreibt die HTML-Seite nach Erzeugung durch die Template-Engine in die HTTP-Antwort

HTML-SEITE ERZEUGEN: BEISPIEL

Datei app.js:

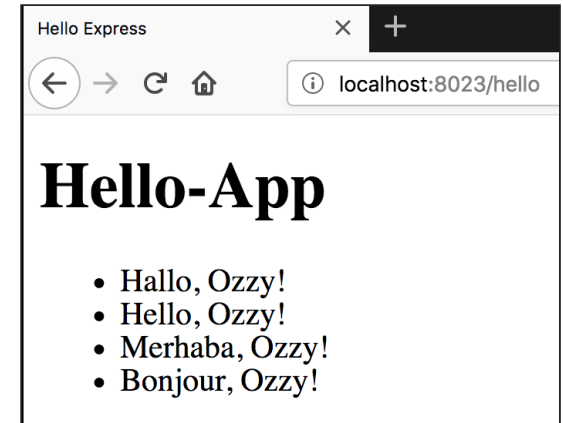
```
const express = require("express");
const app = express();

app.set("view engine", "ejs");
app.set("views", "views");

app.get("/hello", function(req, res, next) {
  // render setzt hier folgende Argumente:
  // 1. Den Namen des Templates (hier: "hello", es
  //    wird also eine Datei "hello.ejs" im
  //    Verzeichnis "views" gesucht)
  // 2. Ein Objekt mit den Daten für das Template
  //    (hier liefert das Objekt die Werte der
  //    benötigten Variablen "name" und "greetings")
  res.render("hello", {
    greetings: ["Hallo", "Hello",
               "Merhaba", "Bonjour"],
    name: "Ozzy"
  });
});

app.listen(8023);
```

Ausgabe im Browser:



TEMPLATES MODULARISIEREN

- Über `<%- include(externes_template) %>` ermöglicht EJS die Modularisierung von Templates
- Das externe Template wird relativ zum Template gesucht, welches das `include` enthält
- Beim Auswerten wird das `include` durch den Inhalt des externen Templates ersetzt
- Typischer Anwendungsbereich: Wiederverwendung durch Auslagern gemeinsamer/wiederkehrender Teile der Webseite wie z.B. Kopfbereich und Fußbereich

TEMPLATES MODULARISIEREN: BEISPIEL

Datei views/hello.ejs:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello Express</title>
    <meta charset="utf-8" />
  </head>
  <body>
    <%- include("includes/header.ejs") %>
    <% if (name) { %>
      <ul>
        <% for (greeting of greetings) { %>
          <li><%= greeting %>, <%= name %>!</li>
        <% } %>
      </ul>
    <% } else { %>
      Kein Name angegeben!
    <% } %>
    <%- include("includes/footer.ejs") %>
  </body>
</html>
```

Datei views/includes/header.ejs:

```
<header><h1>Hello-App</h1></header>
```

Datei views/includes/footer.ejs:

```
<footer><small>&copy;2022 ACME</small></footer>
```

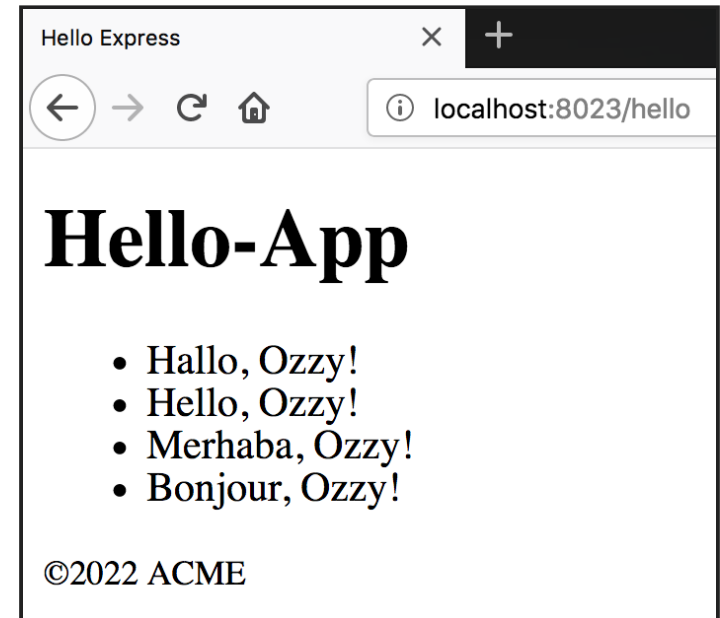
TEMPLATES MODULARISIEREN: BEISPIEL

(2)

Resultierender HTML-Code:

```
<!DOCTYPE html>
<html>
<head>
  <title>Hello Express</title>
  <meta charset="utf-8" />
</head>
<body>
  <header><h1>Hello-App</h1></header>
  <ul>
    <li>Hallo, Ozzy!</li>
    <li>Hello, Ozzy!</li>
    <li>Merhaba, Ozzy!</li>
    <li>Bonjour, Ozzy!</li>
  </ul>
  <footer><small>&copy;2022 ACME</small></footer>
</body>
</html>
```

Ausgabe im Browser:



FAZIT: ERZEUGEN VON ANSICHTEN MIT TEMPLATES

- ➕ Fördern eine bessere Trennung von Zuständigkeiten (insbesondere Darstellung und Applikationslogik)
- ➕ Ermöglichen gute Strukturierung und Wiederverwendbarkeit durch Modularisierung
- ➕ Werden durch Entwicklungsumgebungen oft besser unterstützt als Template-Literale
- ⚠️ Insbesondere bei mächtigen Template-Sprache (wie EJS) müssen EntwicklerInnen darauf achten, sich in den Templates auf reine Präsentationslogik zu beschränken

ANFRAGE- UND ANTWORT-OBJEKT

Im Folgenden betrachten wir folgende Aspekte näher:

1. `Request.render`:

Erzeugt eine Ansicht (*View*) und sendet diese zum Client.

2. `Response.body`:

Zugriff auf die Daten im Body der HTTP-Anfrage

ERINNERUNG: ANFRAGE-BODY MIT DEM "HTTP"-MODUL LESEN

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

const server = http.createServer(function(request, response) {
  const url = request.url;
  const method = request.method;

  if (url.startsWith("/new") && method === "POST") {
    let body = "";
    request.on("readable", function() {
      let data = request.read();
      body += data !== null ? data : "";
    });
    request.on("end", function() {
      [...]
    });
  }
}).listen(8042);
```

ANFRAGE-BODY MIT EXPRESS LESEN

- Express vereinfacht den Zugriff auf die Daten im Body der Anfrage durch die Eigenschaft `body` im `Request`-Objekt
 - ! Diese Eigenschaft hat jedoch zunächst standardmäßig den Wert `undefined`
 - Grund: Die Daten im Body müssen erst eingelesen und aufbereitet werden
- ➔ Dies kann z.B. durch die Middleware [body-parser](#) erfolgen

BODY-PARSER EINBINDEN

- body-parser installieren und als Abhängigkeit zu einer bestehenden `package.json` hinzufügen:

```
$ npm install --save body-parser
```

- body-parser in eine Express-Anwendung einbinden:

```
// Modul einbinden
const bodyParser = require("body-parser");

// Middleware registrieren (möglichst am Anfang der Kette)
// Beispiel hier: Mit HTML-Formular gesendete Daten einlesen
app.use(bodyParser.urlencoded({ extended: false }));
```

- Für nachfolgende Middlewares sind die Daten dann über `req.body` als JavaScript-Objekt zugreifbar (analog zu `req.query`)

BODY-PARSER: BEISPIEL

Datei public/helloForm.html:

```
<!DOCTYPE html>
<html>
<head>
  <title>ToDoApp (EJS)</title>
  <meta charset="utf-8" />
</head>
<body>
  <form action="helloForm" method="POST">
    <label for="textField">Name:</label>
    <input id="textField"
           name="helloInput">
  </form>
</body>
</html>
```

Datei app.js:

```
const express = require("express");
const bodyParser = require("body-parser");
const app = express();

app.use(bodyParser.urlencoded({ extended: false }));
app.use(express.static("public"));

app.post("/helloForm",
  function(req, res, next) {
    // Zugriff auf die Daten erfolgt
    // über "req.body"
    console.log(req.body.helloInput);
  });

app.listen(8023);
```

ROUTING MODULARISIEREN

- Bei größeren Web-Anwendungen kann das Routing sehr komplex werden
- Um auch hier modularisieren zu können, bietet Express den `express.Router`
- Auf einer Instanz von `express.Router` können analog zu `app` über Middleware-Funktionen Routen definiert werden

ROUTING MODULARISIEREN (2)

- Die Router-Instanz selbst kann jedoch in ein Modul ausgelagert werden
- Dann kann man die Router-Instanz bei Bedarf importieren und z.B. einer übergeordneten Route zuweisen
- Die Router-Instanz kann sich dann z.B. gezielt um einen bestimmten Routenbereich kümmern

express.Router: BEISPIEL

Datei app.js:

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

// Modul mit den Routen einbinden
const router
  = require('./routes/router.js');

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

// Router registrieren -> Der Router
// verwaltet jetzt Routen, die mit
// "/app" beginnen
app.use("/app", router);

app.listen(8023);
```

Datei routes/router.js:

```
const express = require("express");
const router = express.Router();

// Routen auf der Router-Instanz
// definieren
router.get("/new",
  function(req, res, next) {
    res.send("new-Route");
  });

router.get("/list",
  function(req, res, next) {
    res.send("list-Route");
  });

// Router zugreifbar machen
module.exports = router;
```