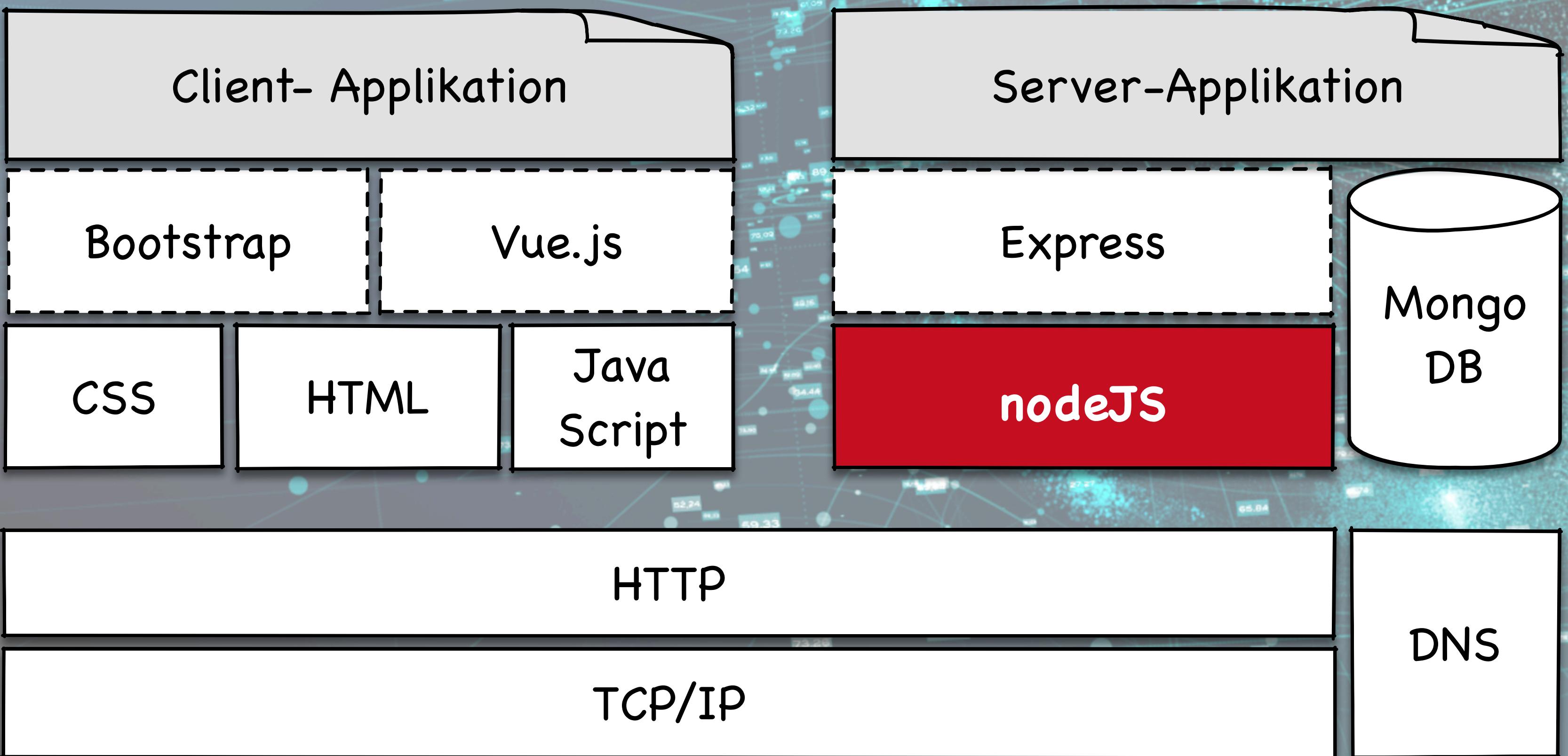


WEB TECHNOLOGIEN 2022

KAPITEL 7: SERVER-SIDE PROGRAMMING

PROF. DR. AXEL KÜPPER
FACHGEBIET SERVICE-CENTRIC NETWORKING | TU BERLIN & T-LABS

WEBTECHNOLOGIEN ÜBERBLICK



7.1 SERVER-SIDE SCRIPTING IM ÜBERBLICK

EINFÜHRUNG UND KLASSEFIKATIONEN

SPRACHEN FÜR SERVER-SIDE SCRIPTING

- ASP (*.asp)
- ActiveVFP (*.avfp)
- ASP.NET (*.aspx)
- ASP.NET MVC (*.cshtml)
- C (*.c, *.csp) via CGI
- ColdFusion Markup Language (*.cfm)
- Go (*.go)
- Groovy Server Pages (*.gsp)
- Hack (*.php)
- Haskell (*.hs) (example: Yesod)
- Java (*.jsp) via JavaServer Pages
- JavaScript using Server-side JavaScript (*.ssjs, *.js) (example: Node.js)
- Lasso (*.lasso)
- Lua (*.lp *.op *.lua)
- Progress WebSpeed (*.r, *.w)
- Pascal (*.p, *.pas, *.inc, *.px) (example: ModernPascal)
- Perl via the CGI.pm module (*.cgi, *.ipl, *.pl)
- PHP (*.php, *.php3, *.php4, *.phtml)
- Python (*.py) (examples: Pyramid, Flask, Django)
- R (*.rhtml) - (example: rApache)
- Ruby (*.rb, *.rbw) (example: Ruby on Rails)
- SMX (*.smx)
- Tcl (*.tcl)
- WebDNA (*.dna, *.tpl)
- Parser (*.p)
- Bigwig (*.wig)

https://en.wikipedia.org/wiki/Server-side_scripting

SERVER-SIDE SCRIPTING

- Technologie für Web-Backends bei der HTTP-Anfragen bearbeitet werden, indem ein Script oder ein binäres Programm im Backend ausgeführt wird, um eine dynamische Webseite zu erstellen
- Alternative zum Client-side Scripting, bei der Inhalte dynamisch per JavaScript im Frontend erzeugt werden
- Client-side und Server-side Scripting werden heute in Kombination verwendet
- Server-side Scripting implementiert Geschäftslogik einer Webanwendung, meist unter Zuhilfenahme von Datenbanken, Dateisystemen und Anwendungsservern im Backend

KLASSEFIKATIONEN

- Anwendungsprogramme versus Server Pages
- Kompilierte versus interpretierte Programme
- Alleinstehende (standalone) versus Integrierte Programme
- LAMP versus MEANS

7.1 SERVER-SIDE SCRIPTING IM ÜBERBLICK

ANWENDUNGSPROGRAMME: EIGENSTÄNDIG VERSUS INTEGRIERT

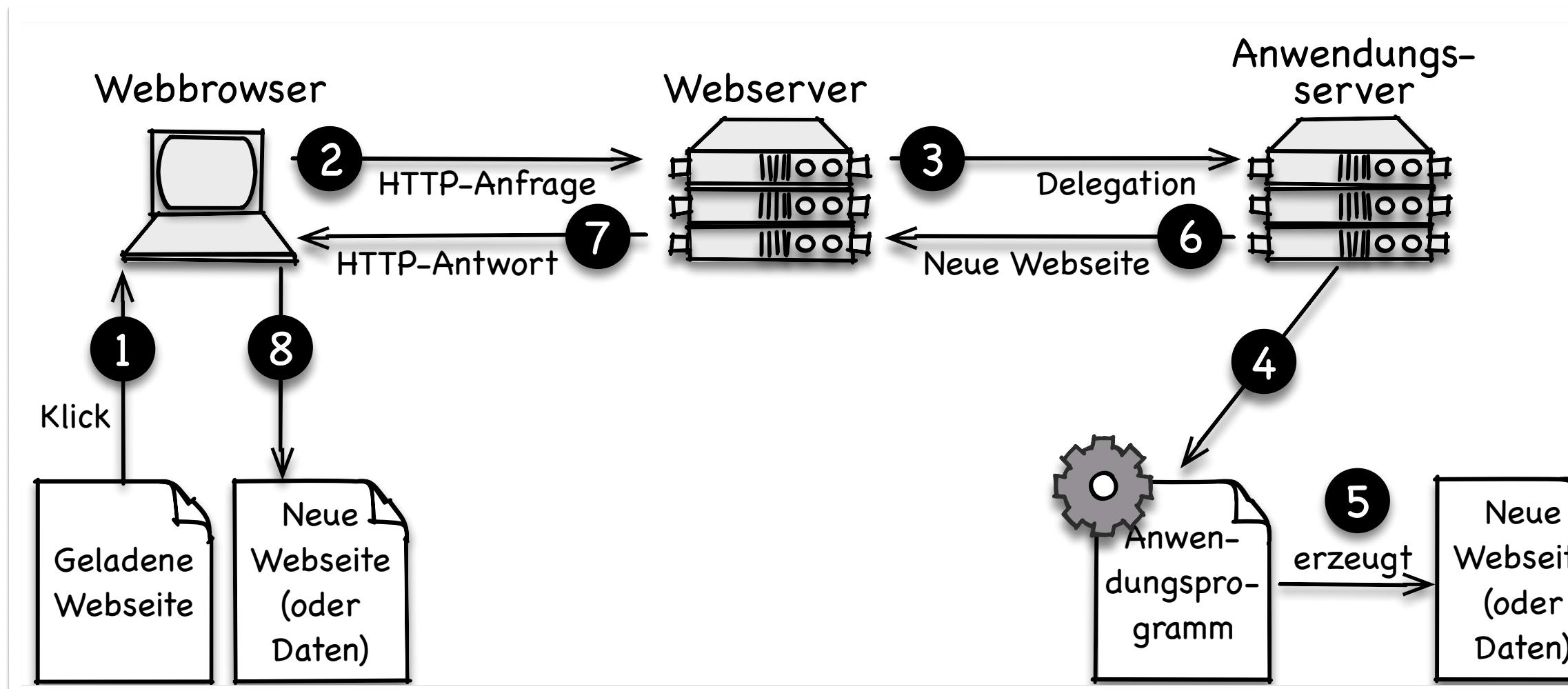
- Prinzip: Anwendungsprogramm erzeugt HTML auf der Ausgabe

EIGENSTÄNDIGES PROGRAMM

- Programm wird für jede Anfrage in einem eigenen Prozess gestartet
- Webserver delegiert Bearbeitung der Anfrage über eine Schnittstelle (*Common Gateway Interface, CGI*) an das Anwendungsprogramm
- Beispiel: Perl oder C-Programme, angebunden über das Common Gateway Interface

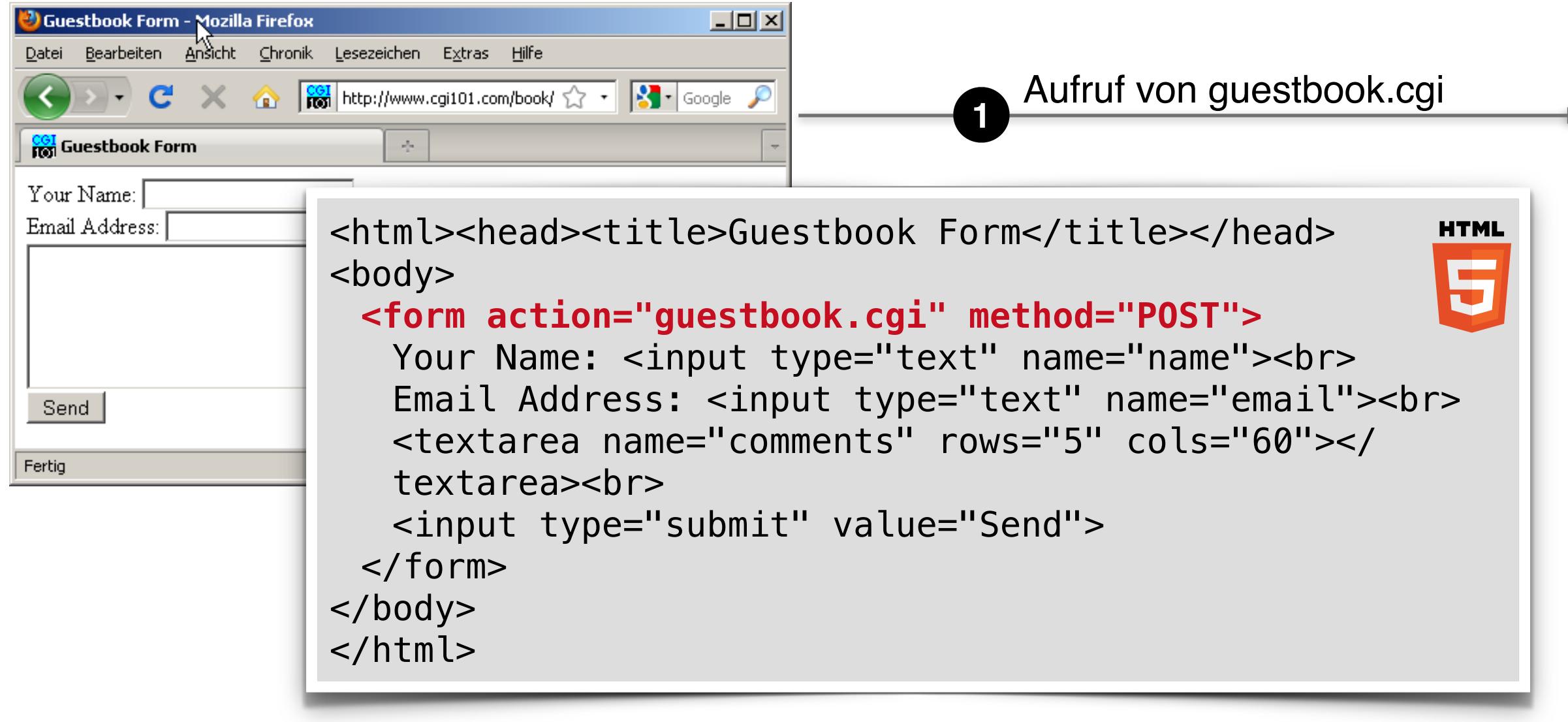
INTEGRIERTES PROGRAMM

- Anbindung an den Webserver über Erweiterungsmodule
- Anwendungsserver mit integriertem Webserver
- Beispiel: Tomcat zur Ausführung von Java-Programmen



7.1 SERVER-SIDE SCRIPTING IM ÜBERBLICK

ANWENDUNGSPROGRAMME: BEISPIEL CGI

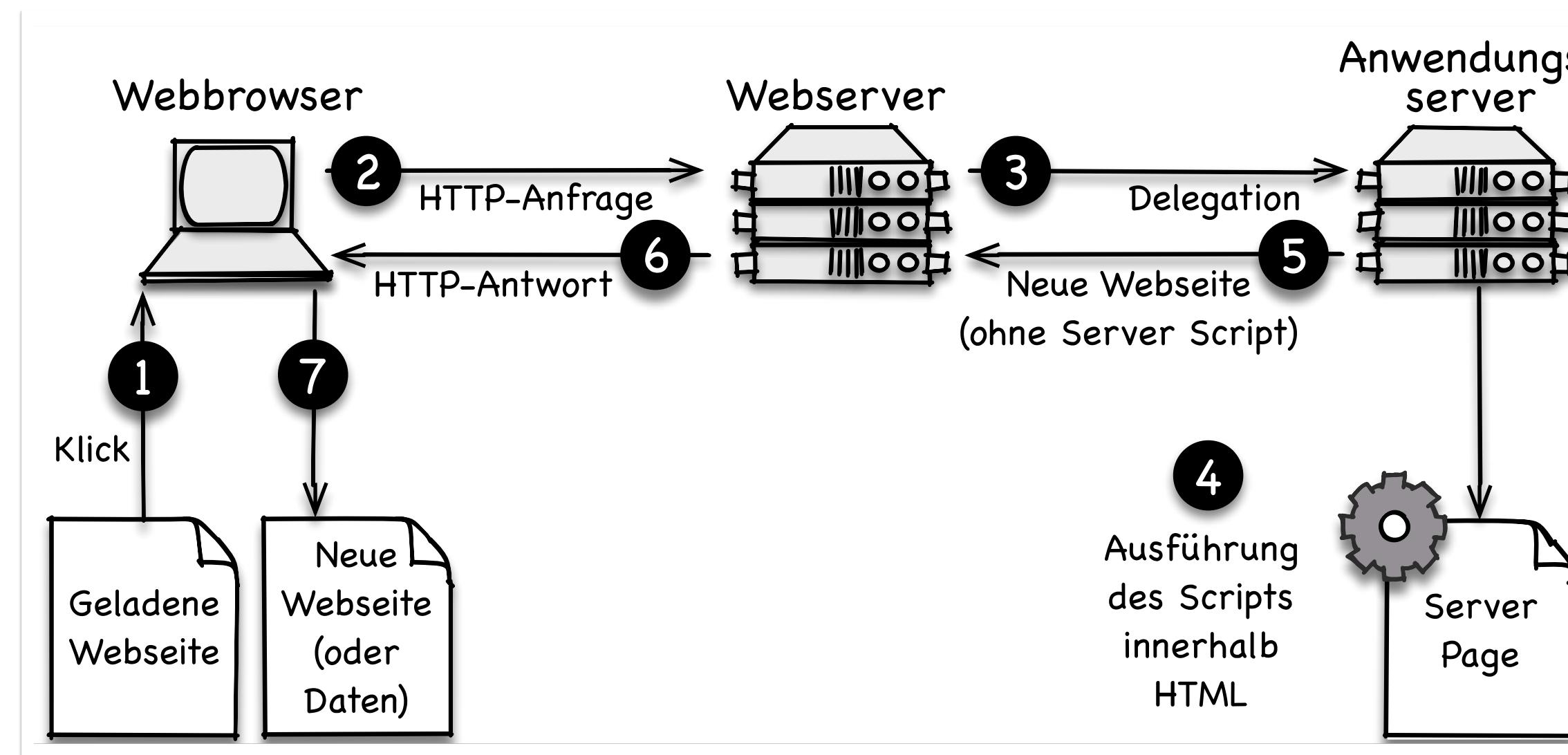


```
#!/usr/bin/perl -wT
use CGI qw(:standard);
use CGI::Carp qw(warningsToBrowser fatsalsToBrowser);
use strict;
print header;
print start_html("Results");
# first print the mail message...
$ENV{PATH} = "/usr/sbin";
open (MAIL, "|/usr/sbin/sendmail -oi -t -odq") or
&dienice("Can't fork for sendmail: $!\n");
print MAIL "To: recipient@cgi101.com\n";
print MAIL "From: nobody@cgi101.com\n";
print MAIL "Subject: Form Data\n\n";
foreach my $p (param()) {
print MAIL "$p = ", param($p), "\n";
}
close(MAIL);
# now write (append) to the file
open(OUT, ">>guestbook.txt") or &dienice("Couldn't open
output file: $!");
foreach my $p (param()) {
print OUT param($p), "|";
}
print OUT "\n";
close(OUT);
print <<EndHTML;
<h2>Thank You</h2>
<p>Thank you for writing!</p>
<p>Return to our <a href="index.html">home page</a>.</p>
EndHTML
print end_html;
sub dienice {
my($errmsg) = @_;
print "<h2>Error</h2>\n";
print "<p>$errmsg</p>\n";
print end_html;
exit;
}
```



7.1 SERVER-SIDE SCRIPTING IM ÜBERBLICK

SERVER PAGES



- Prinzip: Vorgefertigte HTML-Seiten enthalten ausführbare Anweisungen (Code Snippets) in einer bestimmten Programmiersprache
- Code Snippets werden vor Auslieferung der HTML-Seite durch den Anwendungsserver interpretiert
- Bei Ausführung der Code Snippets werden dynamische Inhalte erzeugt
- Code Snippets werden vor der Auslieferung an den Client entfernt (und durch Inhalte ersetzt)
- Beispiel: PHP, Active Server Pages, Java Server Pages

7.1 SERVER-SIDE SCRIPTING IM ÜBERBLICK

SERVER PAGES: BEISPIEL PHP

```
<html><head></head>
<body>
<?php
    /* if the "submit" variable does not exist, the form has
       not been submitted - display initial page */
    if (!isset($_POST['submit'])) {
?
<form action=<?php echo $_SERVER['PHP_SELF']; ?>" method="post">
Enter your age: <input name="age" size="2">
<input type="submit" name="submit" value="Go">
</form>
<?php
}
else {
    /* if the "submit" variable exists, the form has been
       submitted - look for and process form data */
    // display result
    $age = $_POST['age'];
    if ($age >= 21) {
        echo 'Come on in, we have alcohol and music
              awaiting you!';
    }
    else {
        echo 'You\'re too young for this club, come back
              when you\'re a little older';
    }
?
</body></html>
```

PHP: HYPERTEXT PREPROCESSOR

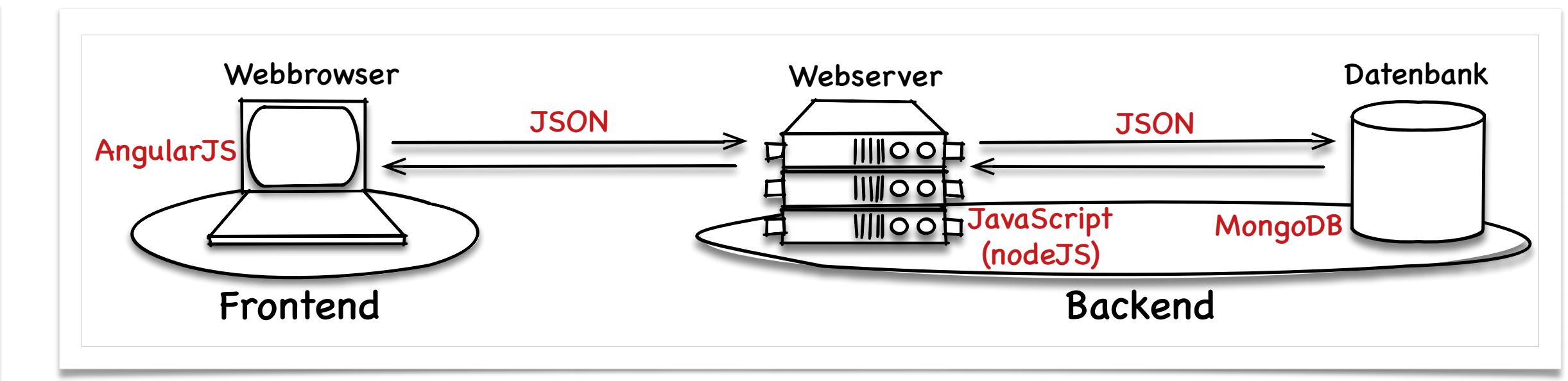
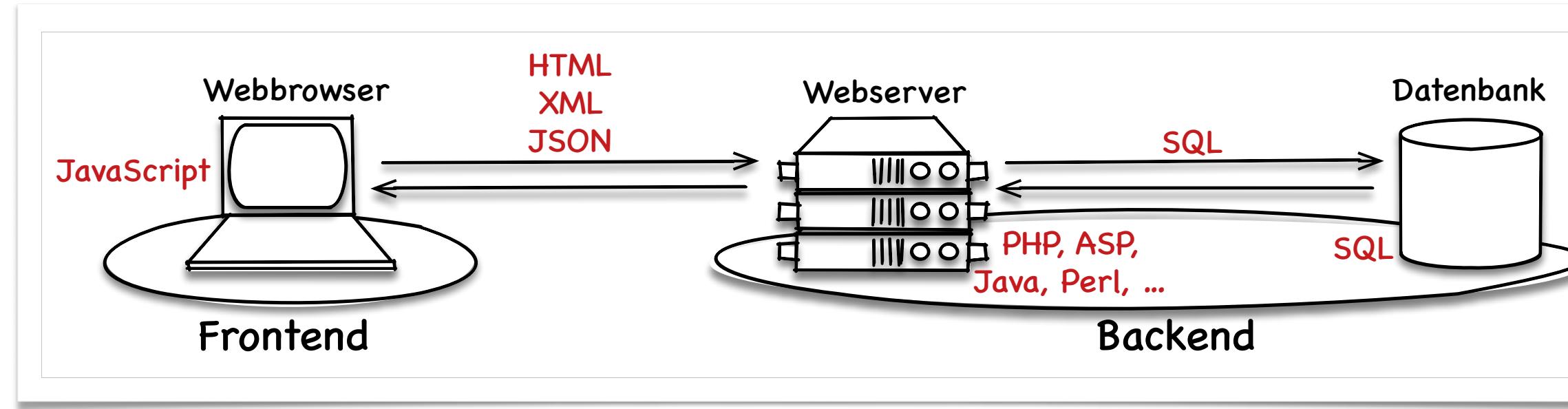
- Skriptsprache mit einer C-ähnlichen Syntax
- Zahlreiche Funktionsbibliotheken
- Ausführung durch Interpreter
- Interpretation der mittels <?php>-Elementen eingebunden Anweisungen und Ignorieren der HTML-Elemente

```
<!DOCTYPE HTML>
<html>
    <head>
        <title>Hallo-Welt-Beispiel</title>
    </head>
    <body>
        <?php
            echo 'Hallo Welt!';
        ?>
    </body>
</html>
```

Einfaches Beispiel: Ausgabe von
"Hallo Welt!"

7.1 SERVER-SIDE SCRIPTING IM ÜBERBLICK

LAMP VERSUS MEAN (WDHLG. AUS KAPITEL 1)



LAMP

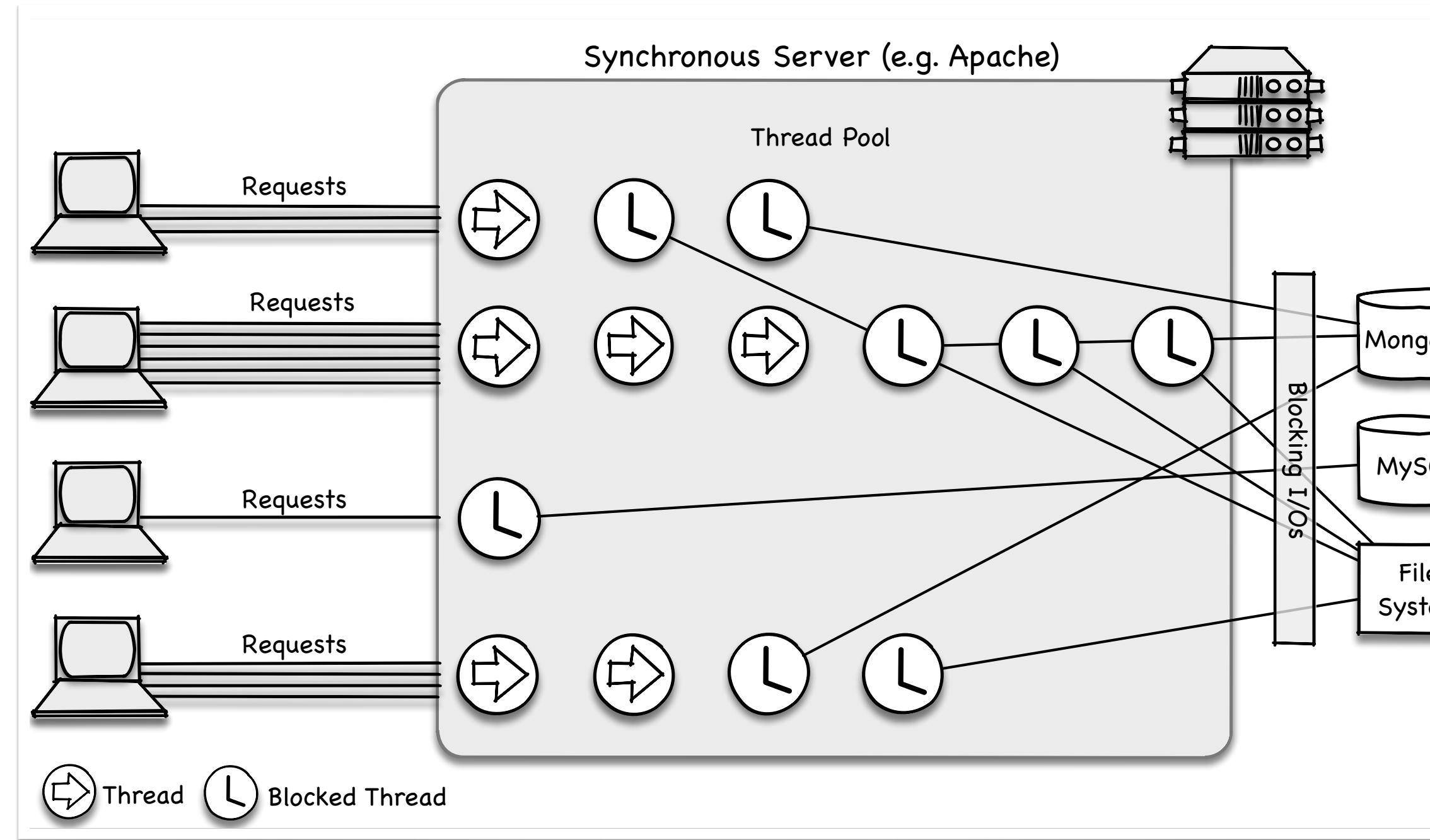
- LAMP: Linux, Apache, MySQL, PHP
- Seit 10 Jahren Standardkonfiguration für Webapplikationen
- Zunehmend veraltet und Anforderungen moderner Webapplikationen nicht mehr gewachsen
- PHP gilt als umständlich und erfordert Integration in Apache
- Apache genügt häufig nicht mehr Leistungsanforderungen (zu langsam)
- Viele Übersetzungen: XML to HTML to PHP to SQL
- Frontend arbeitet mit anderen Technologien als Backend
- SQL skaliert nicht gut (und gehört nun zu Oracle ;-)

MEAN

- MEAN: MongoDB, Express, AngularJS, NodeJS
- 100% free und Open Source
- Durchgängig JavaScript (Basis für AngularJS und NodeJS/ Express)
- Durchgängig JSON als Austauschformat
- MongoDB als dokumentenorientierte Datenbank mit einfacher Übersetzung zu JSON
- Event-basierte, asynchrone Umsetzung von JSON -> sehr schnell und sehr wenig Overhead

7.1 SERVER-SIDE SCRIPTING IM ÜBERBLICK

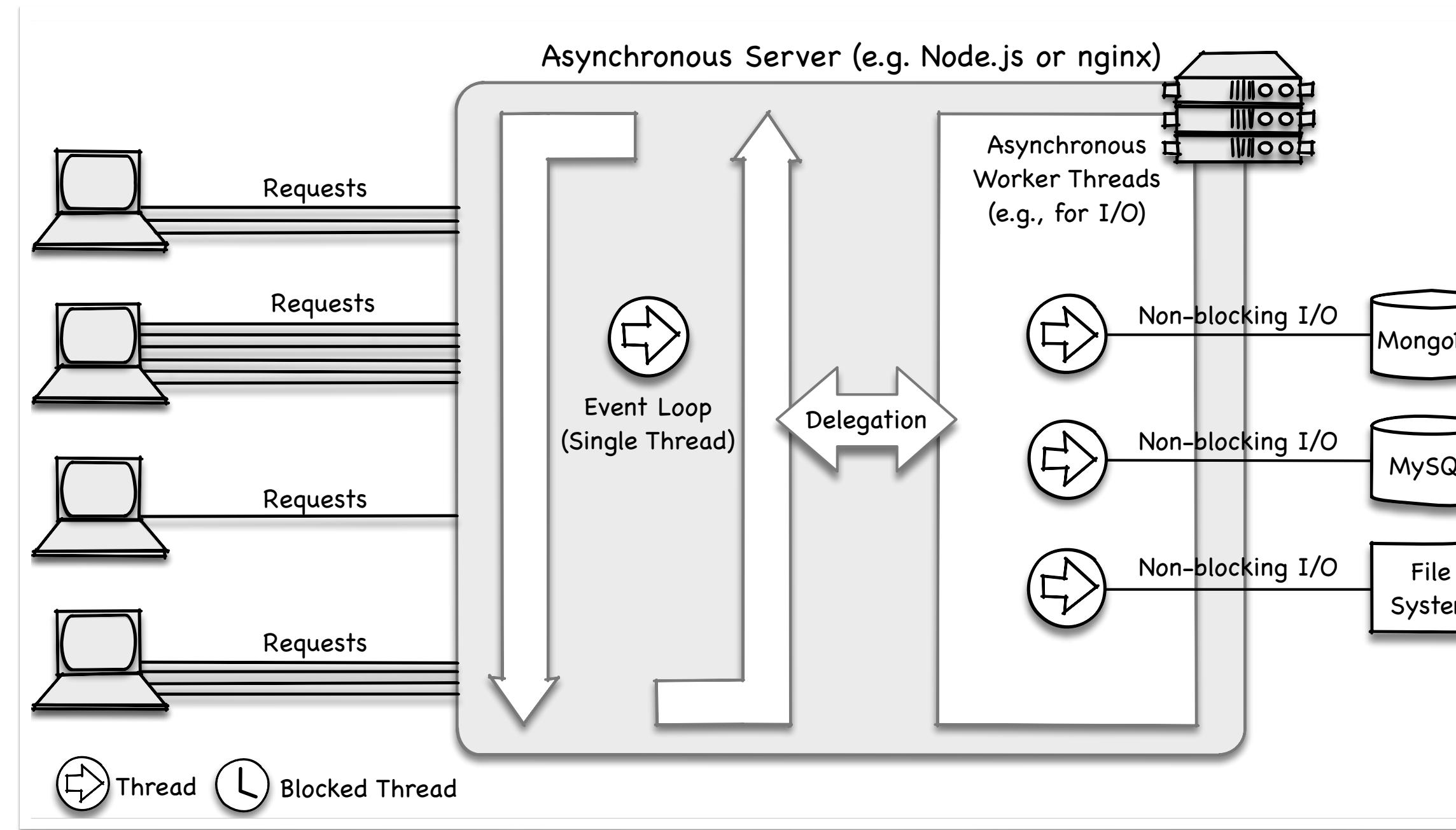
MULTI-THREAD-SERVER



- Erzeugung eines dedizierten Threads pro TCP-Verbindung
- Threads haben einen exklusiven Speicherbereich
- Hoher CPU-Aufwand für das Thread-Management, d.h. für die ständig wechselnde Zuweisung von CPU-Ressourcen zu Threads
- Jeder Thread bearbeitet exklusiv einen Anfrage/Antwort-Zyklus der Verbindung
- Lebenszeit eines Threads entspricht der Verbindungsduer, d.h. mehrere HTTP-Anfragen/Antworten (siehe **keep-alive** von HTTP)
- Maximal sechs gleichzeitige Verbindungen pro Client
- Synchrone Arbeitsweise: Operationen auf externen Ressourcen (Datenbanken, Dateisysteme, Kommunikation zu anderen Anwendungsservern) verursachen Blockierung des jeweiligen Threads
- Hoher Specheraufwand und CPU-Verbrauch führen zu mäßiger Skalierbarkeit

7.1 SERVER-SIDE SCRIPTING IM ÜBERBLICK

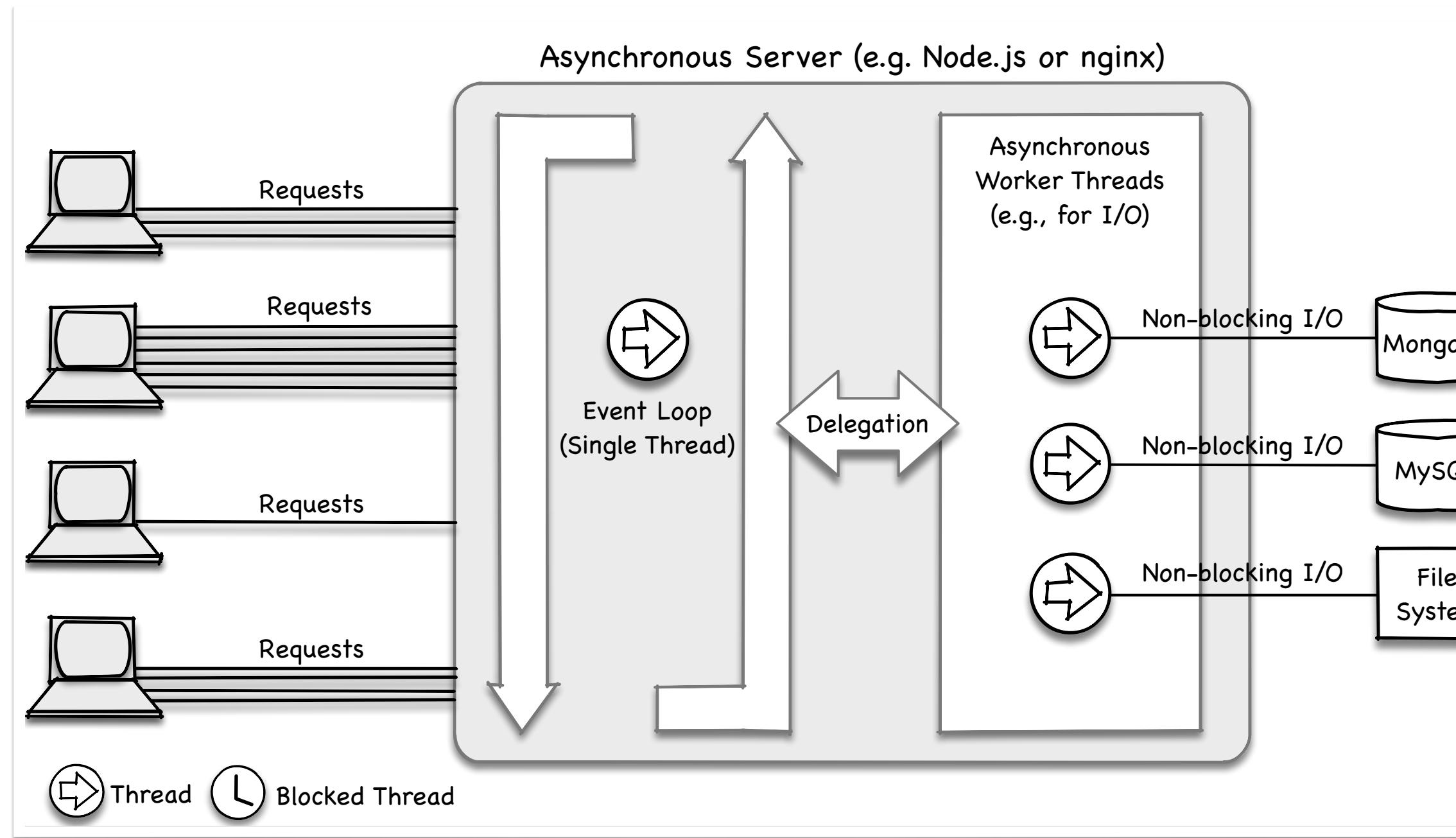
SINGLE-THREAD-SERVER (I)



- Bearbeitung sämtlicher TCP-Verbindungen durch einen einzelnen Thread (Single Thread)
- Single Thread nimmt HTTP-Anfragen entgegen, bearbeitet sie und erstellt und versendet Antworten
- Asynchrone Arbeitsweise: Operationen auf externen Ressourcen (Datenbanken, Dateisysteme, andere Anwendungsserver) werden als "Aufträge" an einen Pool mit asynchronen Threads (Worker Threads) delegiert
- Single Threads und Worker Threads blockieren nicht
- Event Loop: Warteschlange von Ereignissen die durch den Single-Thread nacheinander abgearbeitet werden

7.1 SERVER-SIDE SCRIPTING IM ÜBERBLICK

SINGLE-THREAD-SERVER (II)

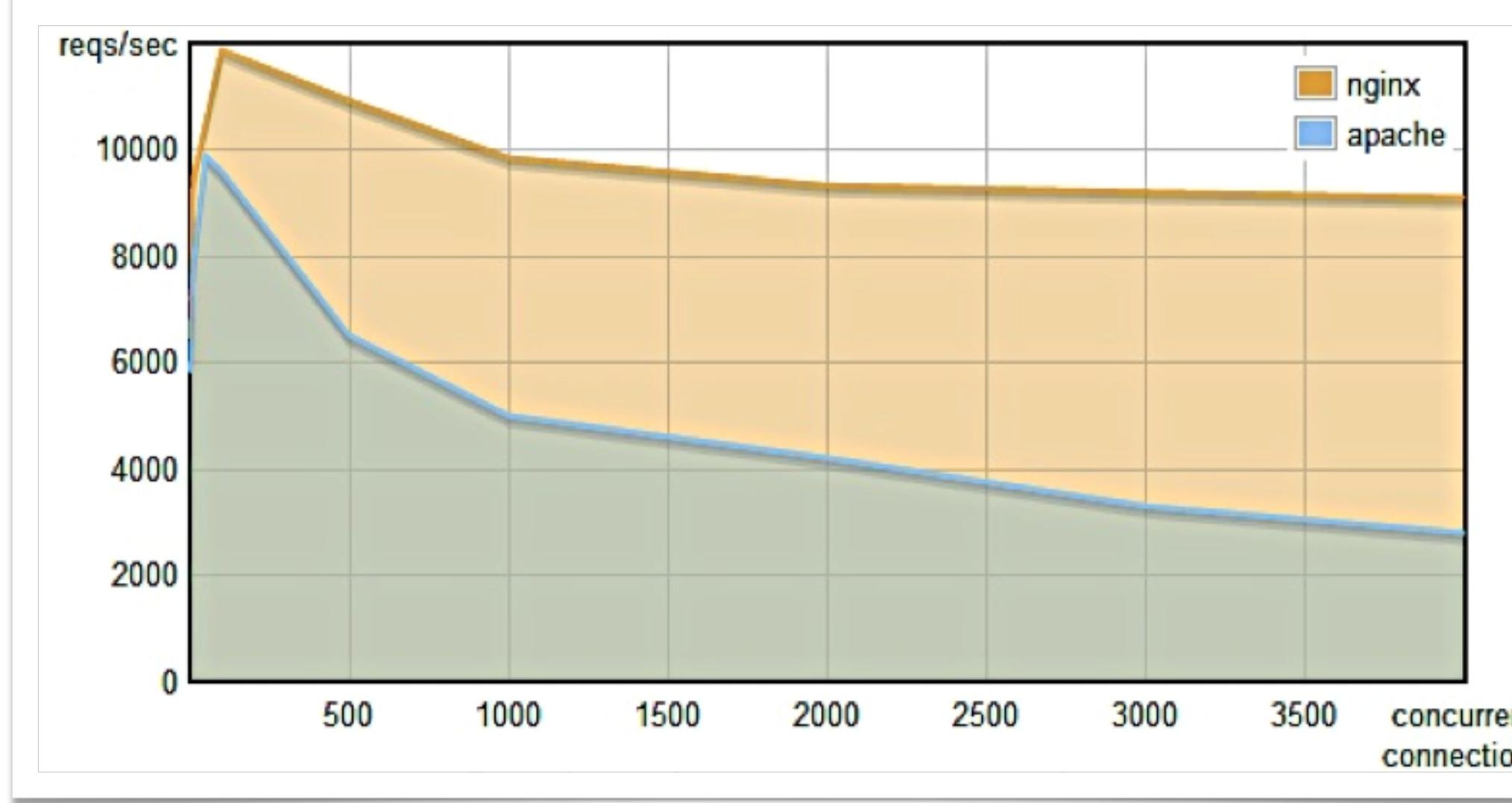


- Ereignisse
 - Anfragen für neue TCP-Verbindungen
 - HTTP-Anfragen
 - Fertig bearbeitete I/O-Aufträge externer Ressourcen (Datenbanken, Anwendungsserver, Dateisysteme)
- Einstellen von Ereignissen in die Event Loop mittels Callback-Funktionen
- Single-Thread-Ansatz verursacht wesentlich weniger Overhead (Speicher, CPU,...) als Multi-Thread-Ansatz und skaliert besser
- Voraussetzung: rechenarme Aufgaben mit vielen I/O-Operationen (z.B. Webapplikationen)

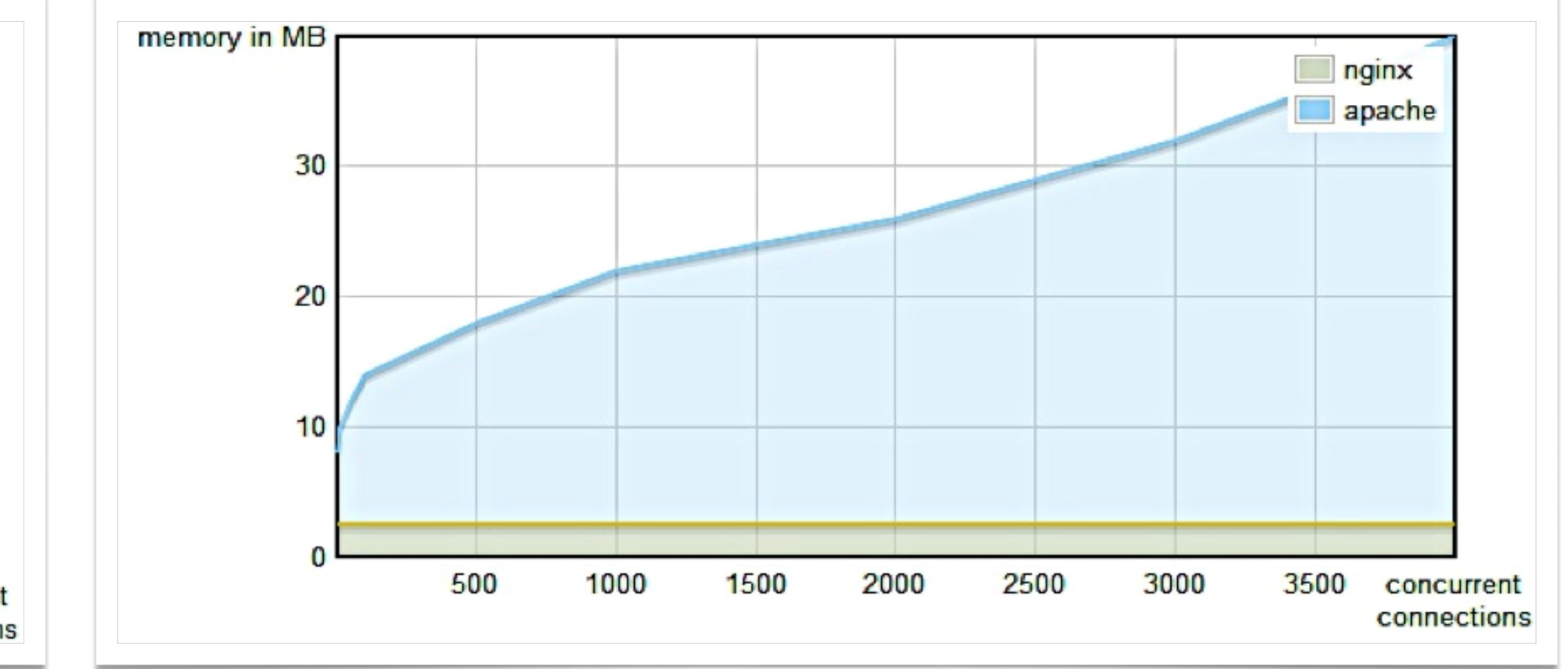
7.1 SERVER-SIDE SCRIPTING IM ÜBERBLICK

MULTI- VERSUS SINGLE-THREAD-SERVER

MAX. # BEARBEITETER ANFRAGEN PRO SEKUNDE
IN ABHÄNGIGKEIT OFFENER VERBINDUNGEN



SPEICHERVERBRAUCH VERSUS
OFFENER VERBINDUNGEN



- Apache: Multi-Thread-Server mit synchroner Verarbeitung und blockierenden Threads
- nginx: Single-Thread-Server mit asynchroner Verarbeitung und nicht-blockierende Threads (vergleichbar zu Node.js)

7.2 NODE.JS ÜBERBLICK

ANFORDERUNGEN MODERNER WEBANWENDUNGEN

- Bessere Skalierbarkeit zur Lösung C10k-Problems: 10.000 Clients müssen quasi gleichzeitig bedient werden können
- Webanwendungen dürfen weder in Komfort noch Reaktivität nativen Anwendungen nachstehen
- Echtzeitfähigkeit zur schnellstmöglichen Versorgung des Benutzers mit neuen Informationen

LÖSUNG: NODE.JS

- Seit 2005 entwickelt von Ryan Dahl, 2009 erstmals veröffentlicht auf der jsconf.eu-Konferenz in Berlin
- Basiert auf JavaScript und der JavaScript-Laufzeitumgebung V8, ursprünglich entwickelt für Google Chrome
- Ereignisgesteuerte, ressourcensparende Architektur, die eine Großzahl gleichzeitig bestehender Netzverbindungen erlaubt
- Integrierte Module für HTTP-Client- und Serverfunktionalitäten, Zugriff auf das Dateisystem etc.
- Beliebig erweiterbar durch zahlreiche externe Module

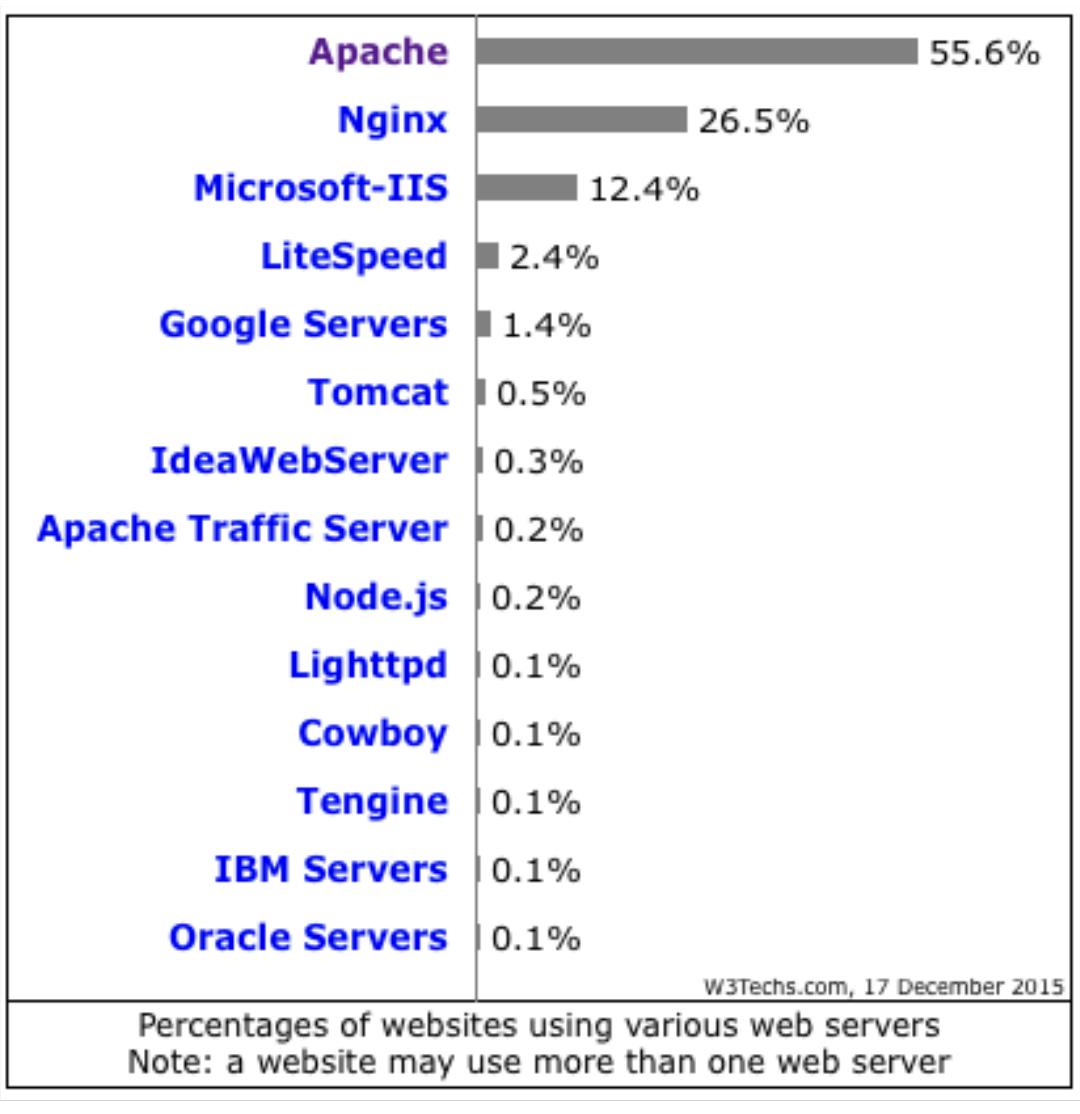


Ryan Dahl

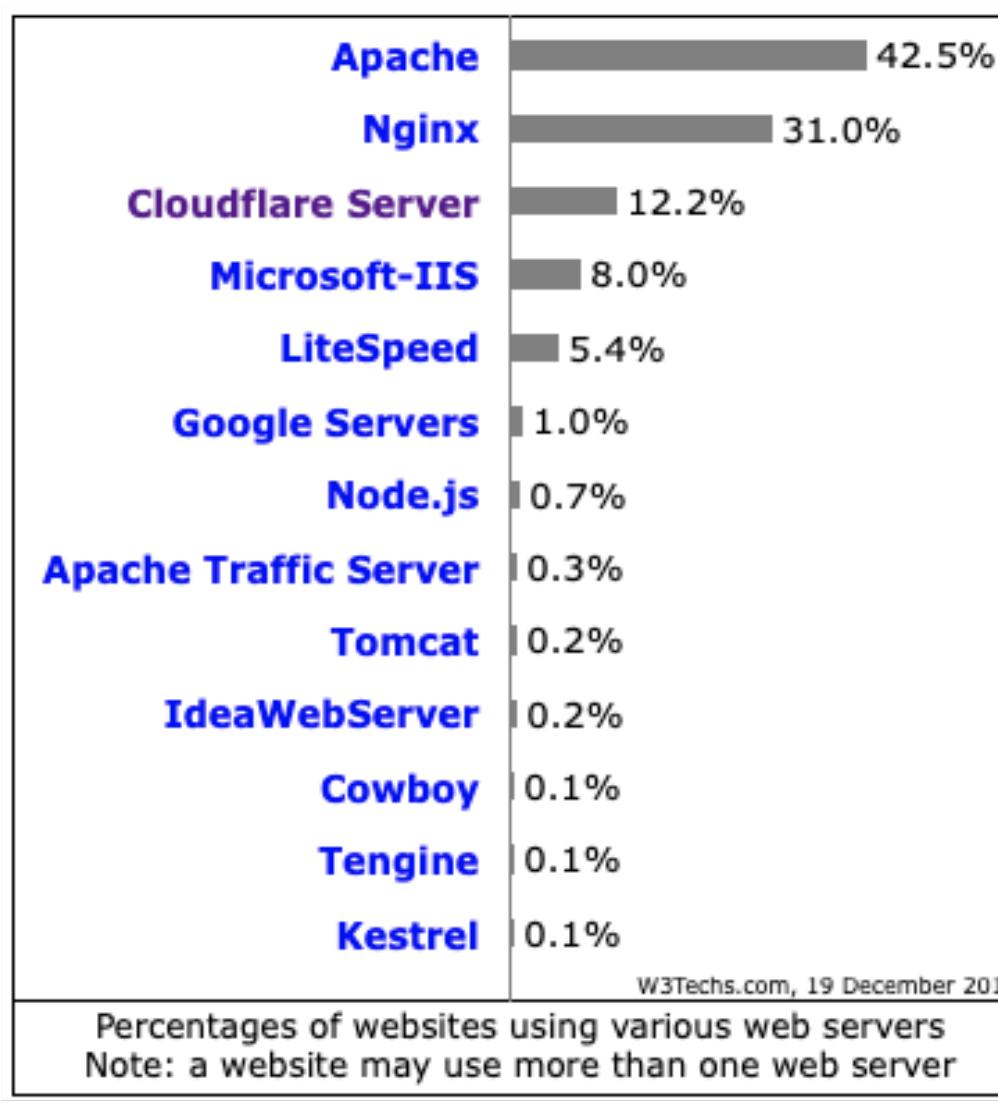
7.2 NODE.JS NGINX



Dezember 2015



Dezember 2019



ÜBERBLICK UND EIGENSCHAFTEN

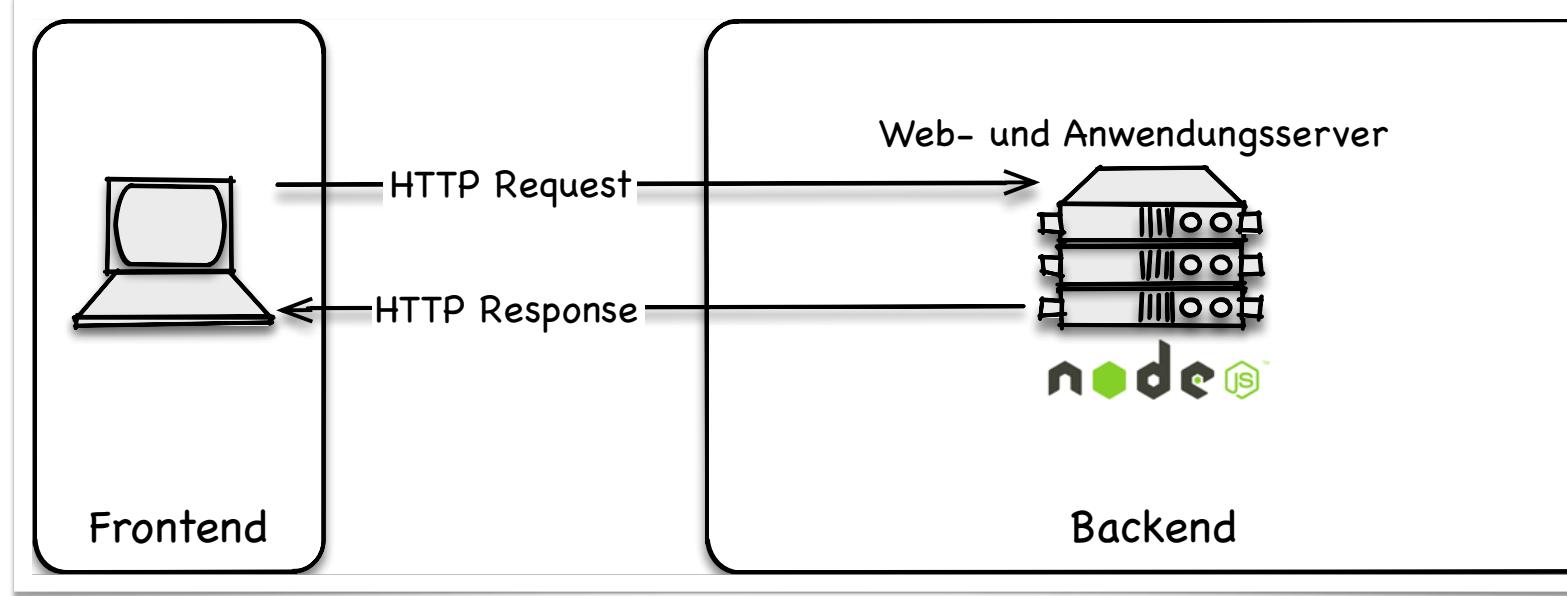
- Modular aufgebauter Webserver mit umfangreichen Managementfunktionen
- Prinzipien: Ereignisbasiert und Single-Threaded
- Hohe Leistung und gute Konfigurierbarkeit
- Häufiger Einsatz als Reverse Proxy

REVERSE PROXY

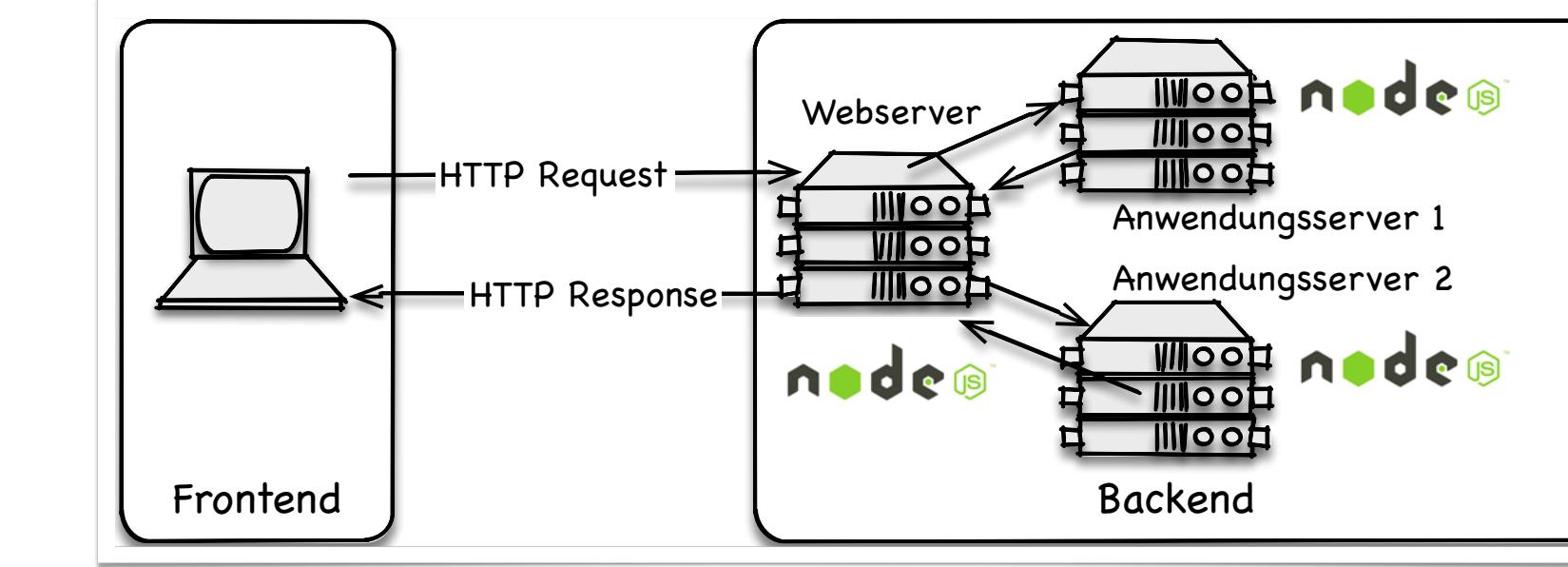
- Intermediärer Webserver zwischen Client und Backend
- Abdeckung des gesamten TCP/IP-Protokollstapels
- Load Balancing: Gleichmäßige Verteilung von Anfragen auf die Web- oder Anwendungsserver eines Clusters
- Komprimierung von Webinhalten
- SSL Endpoint: Durchführung von Ver- und Entschlüsselung zur Entlastung von Web- und Anwendungsservern
- Sicherheit und Anonymität: Erkennung von Angriffen durch Kombination mit Intrusion Detection Systemen (IDS)

7.2 NODE.JS

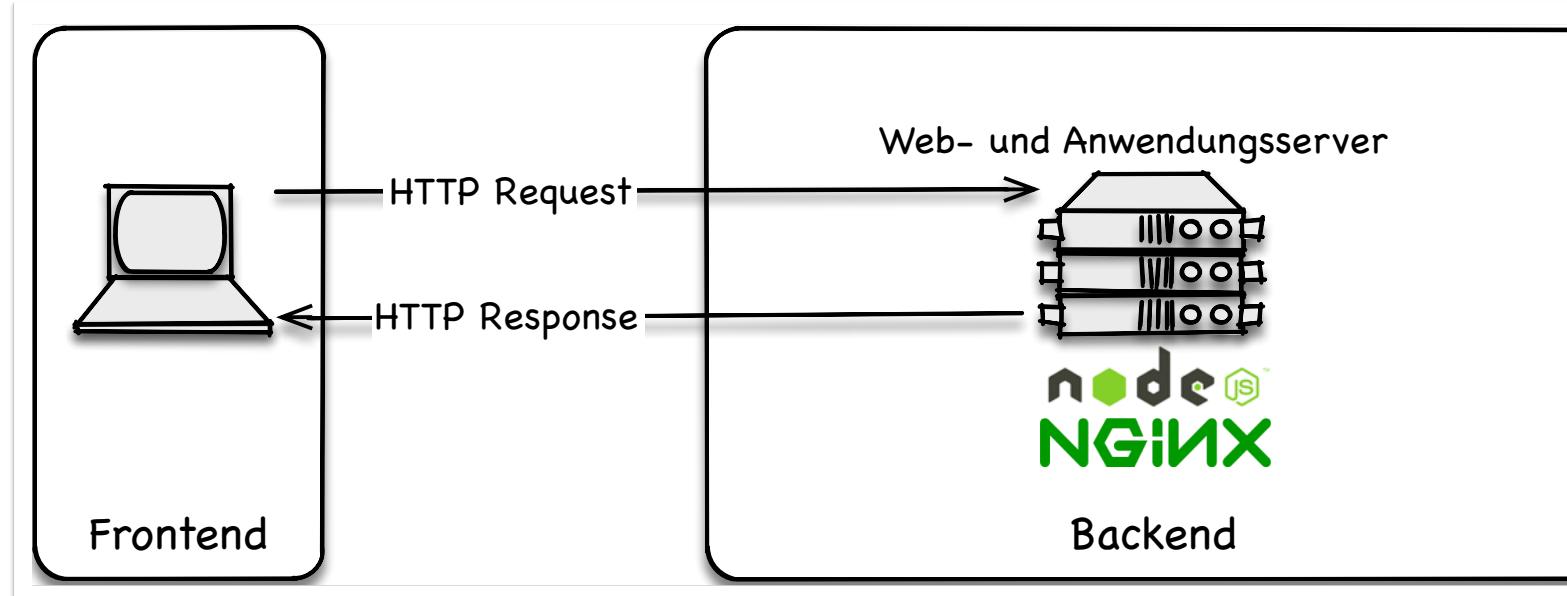
NGINX UND NODE.JS - VERSCHIEDENE KONFIGURATIONEN



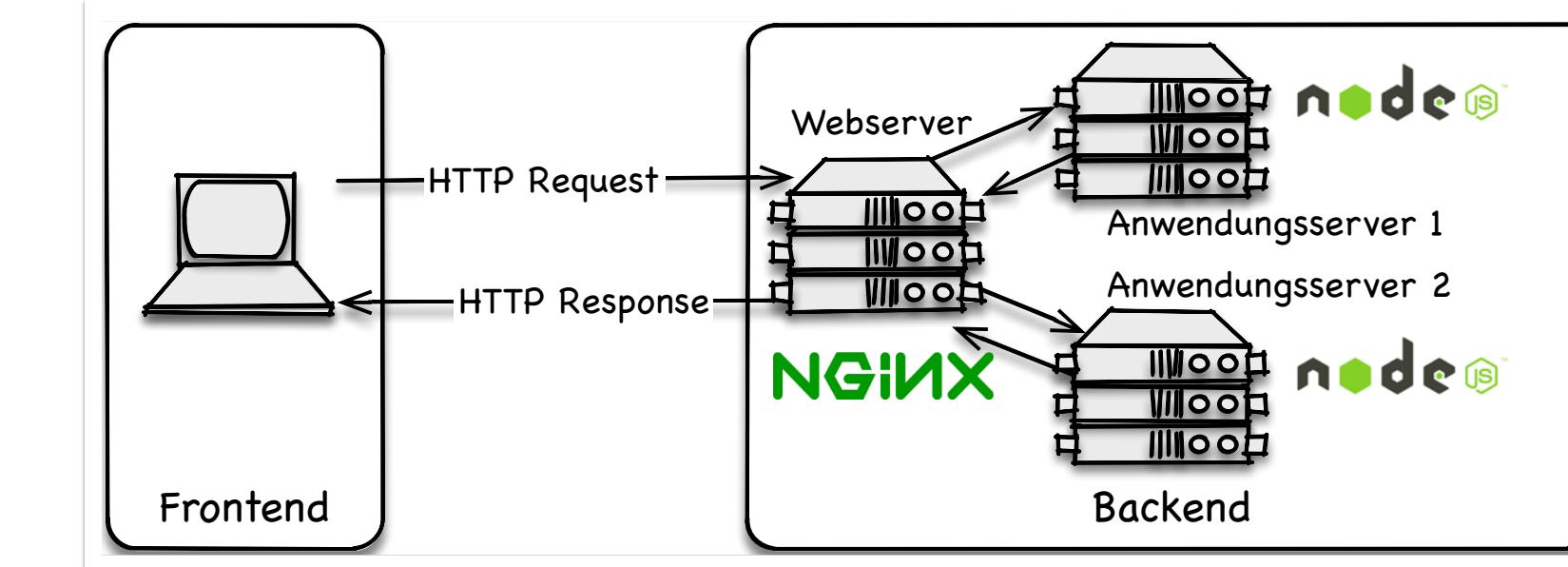
- Kleine Webanwendungen ohne Skalierbarkeitsanforderungen
- Umsetzung von Sicherheits-, Performance- und Sitzungsmanagement durch Node.js



- Komplexe skalierbare Webanwendungen
- Umsetzung von Sicherheits-, Performance- und Sitzungsmanagement durch Node.js



- Kleine Webanwendungen
- Umsetzung von Sicherheits-, Performance- und Sitzungsmanagement durch nginx



- Komplexe skalierbare Webanwendungen
- Umsetzung von Sicherheits-, Performance- und Sitzungsmanagement durch nginx

7.2 NODE.JS MODULE

Jedes Modul enthält eine Instanz von `module`, welche `module.exports` und `exports` wie folgt definiert:

greetings.js:
Beispiel für ein Modul mit zwei Funktionen, welches exportiert werden soll

```
var exports = module.exports = {};
```

1

```
1 sayHelloInEnglish = function() {
2   return "Hello";
3 }
4 sayHelloInSpanish = function() {
5   return "Hola";
6 }
7
8 exports.sayHelloInEnglish = function() {
9   return "HELLO";
10}
11
12 exports.sayHelloInSpanish = function() {
13   return "Hola";
14}
```

2

3

greetings.js:
Funktionen werden mit `exports` exportiert

```
8 module.exports.sayHelloInEnglish = function() {
9   return "HELLO";
10}
11
12 module.exports.sayHelloInSpanish = function() {
13   return "Hola";
14}
```

4

greetings.js
(Alternative):
Alternative mit `module.exports`

NODE-MODULE

- Node.js verwendet eigenes Modulsystem (unabhängig von ES6)
- Integrierte Module: Vorinstallierte Standardmodule
- 3rd-Party-Module: Module von Drittanbietern

module-MODUL

- Jedes Modul enthält automatisch eine Instanz von `module`
- `require()`: Methode innerhalb von `module` um Module in den eigenen Code zu importieren
- `module.exports`: Objekt innerhalb von `module` zum Export von Code
- `exports`: Referenz auf `module.exports`



<https://nodejs.org/api/modules.html>

7.2 NODE.JS MODULE

Das Objekt **module.exports** hat nach der Zuweisung den folgenden Wert:

Jedes Modul enthält eine Instanz von **module**, welche **require** wie folgt definiert:

Import des Moduls **greetings.js** und Nutzung der importierten Funktionen

```
5 module.exports = {  
  ...  
  sayHelloInEnglish: function() {  
    return "HELLO";  
  },  
  
  sayHelloInSpanish: function() {  
    return "Hola";  
  }  
};
```

```
6 var require = function(path) {  
  // ...  
  return module.exports;  
};
```

```
7 1 var greetings = require("./greetings.js");  
2 greetings.sayHelloInEnglish();  
3 greetings.sayHelloInSpanish();
```

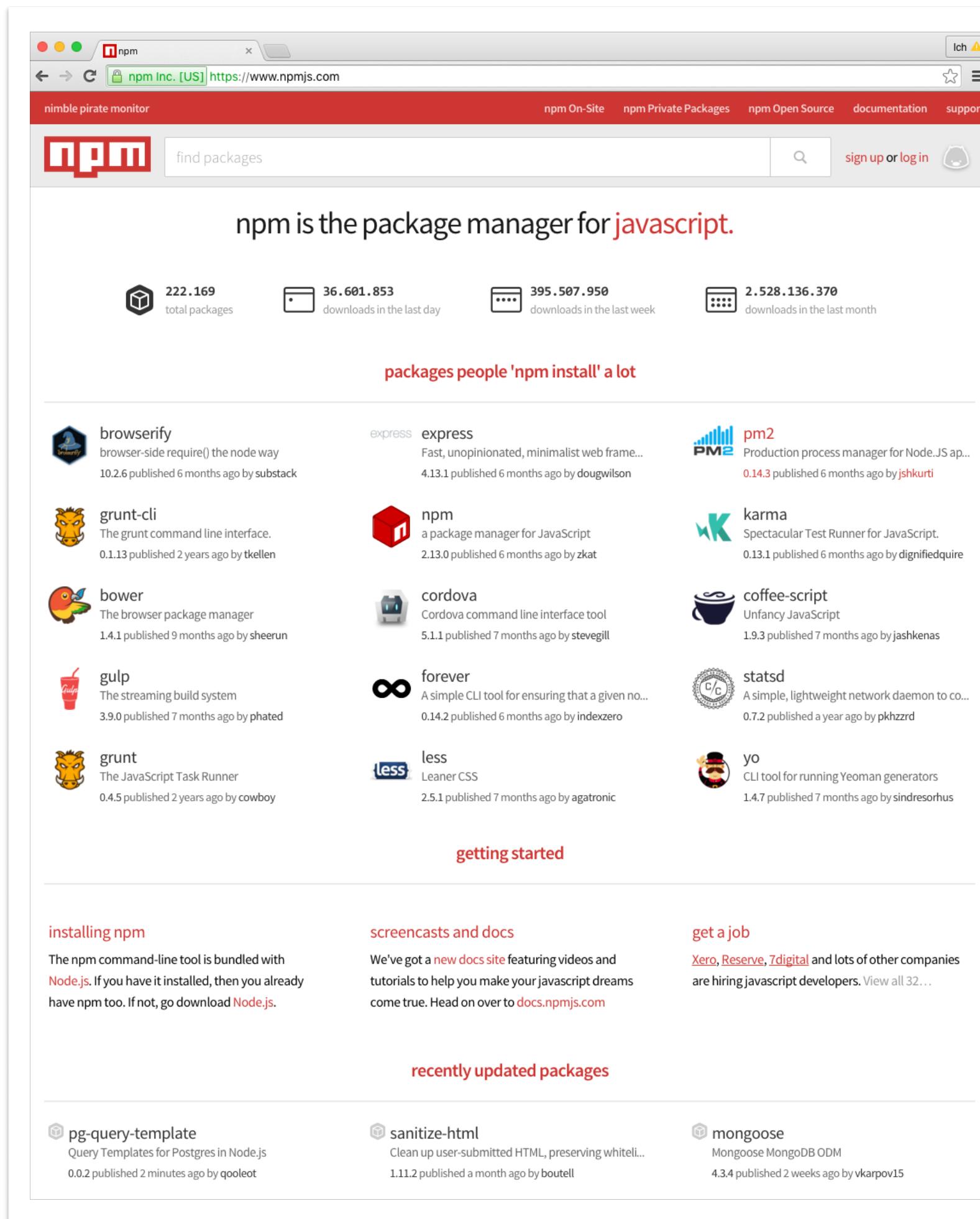
BEACHE

- Mit **require** importierte Funktionen und Objekte verhalten sich wie Funktionen und Objekte, die in der gleichen Datei des importierenden Quellcodes stehen
- **module.exports** darf nicht überschrieben werden, da sonst die Referenz auf das zu exportierende Objekt überschrieben wird



<https://nodejs.org/api/modules.html>

7.2 NODE.JS NODE PACKET MANAGER (I)



- Package: Zusammenfassung mehrerer Module
- NPM: Tool zum Installieren, Aktualisieren und Entfernen von Third-Party-Packages
- Zwei Erscheinungsformen
 - Web-basiertes Verzeichnis als Marktplatz für öffentlich verfügbare Third-Party-Packages
 - Command Line Interface zum Management von Packages auf dem jeweiligen Node.js-System
- Lokal Installation von Packages:

```
$ npm install <Package Unique Name>
```
- Systemweite Installation (unter /usr/local/lib/node_modules):

```
$ npm install -g <Package Unique Name>
```
- Entfernen von Packages:

```
$ npm uninstall <Package Name>
```

```
$ npm uninstall -g <Package Name>
```
- Aktualisierung von Packages:

```
$ npm update <Package Name>
```

```
$ npm update -g <Package Name>
```

7.2 NODE.JS

NODE PACKET MANAGER (II)

The screenshot shows a terminal window titled "demo — bash — 87x38". The window displays the following text:

```
Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
[name: (demo) filmapp
[version: (1.0.0) 0.0.1
[description: Demo fuer Vorlesung Webtechnologien
[entry point: (index.js) server.js
[test command:
[git repository:
[keywords: Node.js, MongoDB, Angular.js, Express
[author: Axel Küpper
[license: (ISC) TUB
Sorry, license should be a valid SPDX license expression.
[license: (ISC) MIT
About to write to [REDACTED] webtech-ws15/demo/package.json:

{
  "name": "filmapp",
  "version": "0.0.1",
  "description": "Demo fuer Vorlesung Webtechnologien",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "Node.js",
    "MongoDB",
    "Angular.js",
    "Express"
  ],
  "author": "Axel Küpper",
  "license": "MIT"
}

Is this ok? (yes) yes
```

PACKAGE.JSON

- Konfigurationsdatei innerhalb eines Anwendungsverzeichnisses zum Management von Abhängigkeiten zwischen Paketen
- Format: json
- Rein manuelle Erstellung oder mit Hilfe von npm:

```
$ npm init
```

7.2 NODE.JS

NODE PACKET MANAGER (III)

RESULTIERENDES PACKAGE.JSON

```
1  {
2      "name" : "webtechnologien",
3      "version" : "0.0.1",
4      "description" : "Demo für Vorlesung Webtechnologien",
5      "main" : "server.js",
6      "scripts" : {
7          "test" : "echo \"Error: no test specified\" && exit 1"
8      },
9      "keywords" : ["Node.js", "MongoDB", "Angular.js", "Express"],
10     "author" : "Axel Küpper",
11     "license" : "MIT",
12     "dependencies" : {
13         "express" : "latest",
14         "mongoose" : "latest"
15     }
16 }
```

- **npm init** startet einen Nutzerdialog (siehe vorherige Folie) und erstellt anschließend **package.json**

- Hier: manuelles Hinzufügen von Abhängigkeiten
- Alternativ: Hinzufügen von Abhängigkeiten durch **npm**:

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

- Beispiel installiert das Packet **express** und fügt Abhängigkeit in **package.json** ein
- Automatische Installation von Packages basierend auf **package.json** im Anwendungsverzeichnis

```
$ npm install
```

- Aktualisierung

```
$ npm update
```

7.2 NODE.JS INTEGRIERTE MODULE

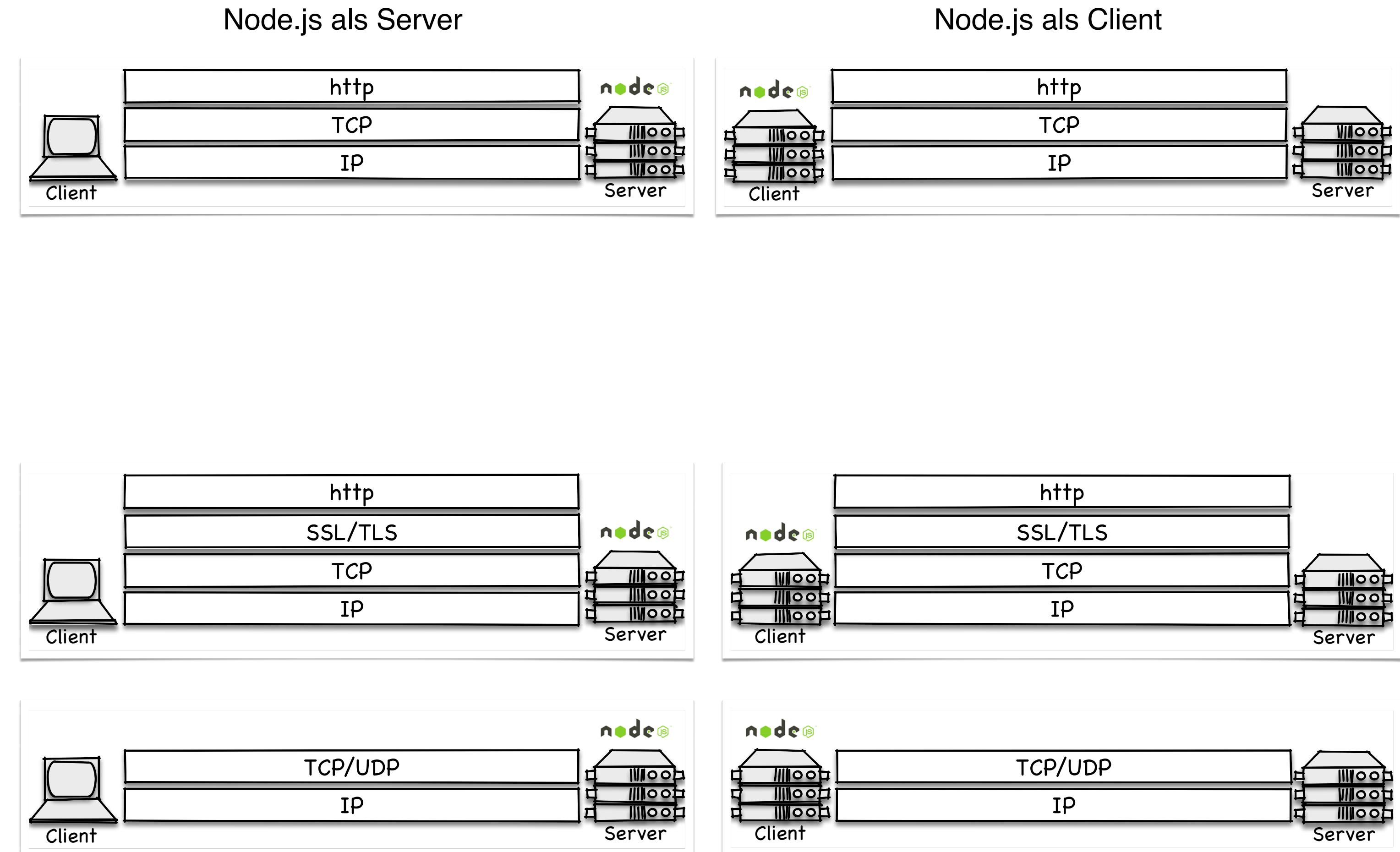
Modulname	Beschreibung
assert	Integriertes Testframework
buffer	Modul für den Umgang mit binären Rohdaten
child_process	Verwaltung von Kindprozessen
cluster	Prozessverwaltung für Mehrkernprozessoren
console	console-Funktionalität in JavaScript
crypto	Sichere Verbindung und Verschlüsselung
debugger	Mittel zur Laufzeitanalyse von Applikationen
dgram	Kommunikation über UDP
dns	Namensauflösung unter Node.js
domain	Sammlung von Exceptions und Fehlern einer Gruppe von Operationen
events	Basismodul für Objekte, die Events erzeugen
fs	Dateisystemoperationen
http	HTTP-Client und -Server
https	HTTPS-Client und -Server
module	Modulloader von Node.js
net	Client- und Serverkomponenten für Netzwerkstreams
os	Betriebssystemoperationen auslesen
path	Umgang mit Pfadinformationen

Modulname	Beschreibung
punycode	Codierung und Decodierung von Punycode-Zeichenketten
querystring	Erstellung und Parsing von URL Query Strings
readline	Einlesen von Informationen über einen Stream wie die Standardeingabe
repl	Interaktive Node.js-Shell
stream	Schreib- und lesbare Datenstreams
string_decoder	Umwandlung von Buffer-Objekten in Zeichenketten
timers	Zeitabhängige Funktionen
tls	Verschlüsselte Kommunikation über Datenstreams
tty	Zwischenschicht zum Ansprechen von Standardein- und -ausgabe
url	Funktionalität zum Umgang mit URLs
util	Hilfsfunktionen
vm	Ausführungsumgebung für JavaScript-Code
zlib	Möglichkeit zum Komprimieren und Dekomprimieren von Daten

7.3 KOMMUNIKATION MIT NODE.JS

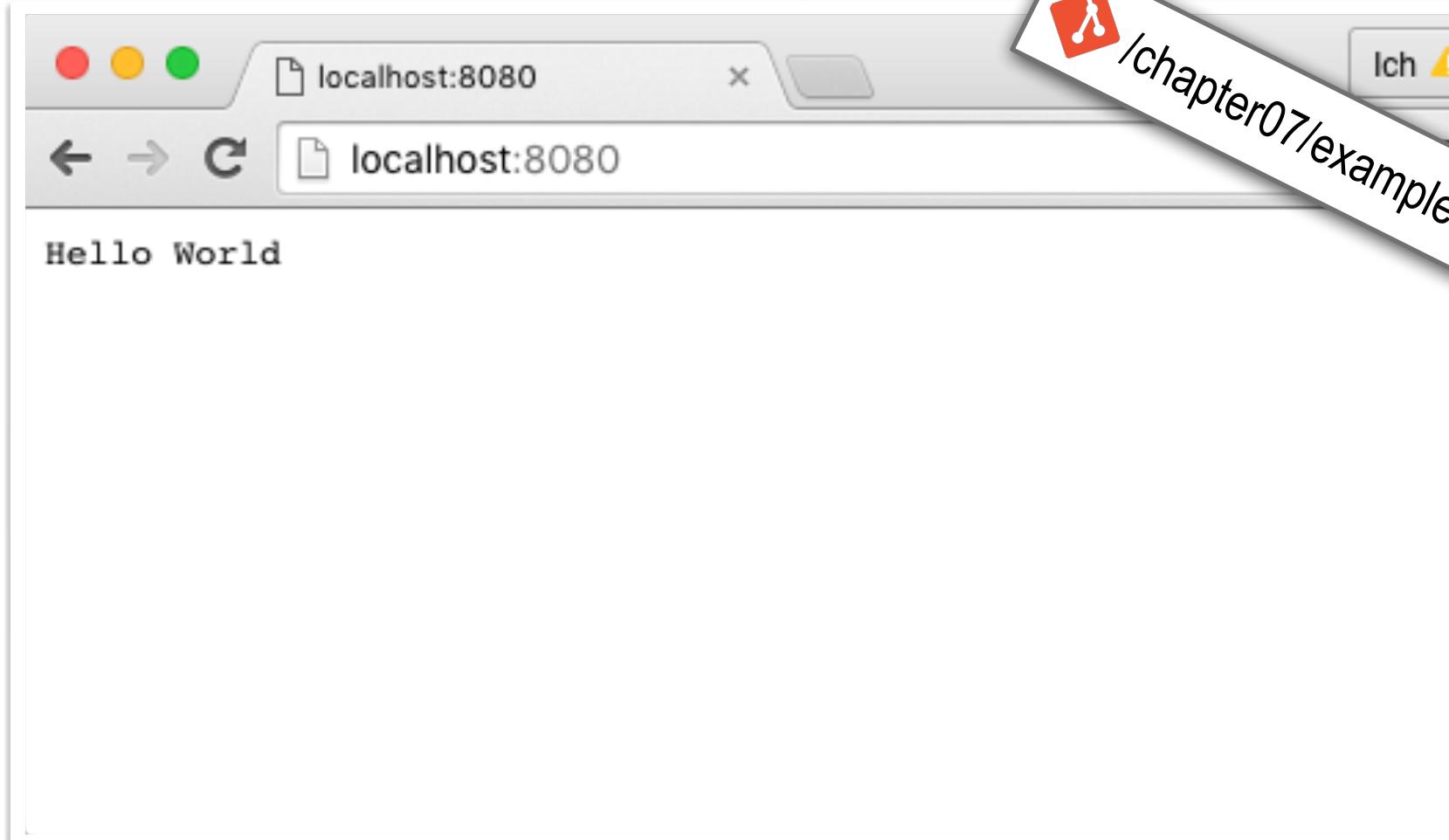
MÖGLICHKEITEN DER KOMMUNIKATION

- Traditionelle Kommunikation über das HTTP-Protokoll
- Kommunikation über eine sichere HTTP-Verbindung
- Umsetzung der sicheren Verbindung über Transport Layer Security (TLS) zwischen TCP und HTTP (oder dem Vorgänger Secure Socket Layer)
- Authentifizierung (des besuchten Servers oder gegenseitig), Vertraulichkeit, Integrität
- Websockets: direkte Kommunikation über das Transportprotokoll (TCP oder UDP)



7.3 KOMMUNIKATION MIT NODE.JS

HTTP-MODUL: BEISPIELE (I)



```
1 var http = require('http');
2 http.createServer(function (request, response) {
3   response.writeHead(200, {
4     'content-type': 'text/plain; charset=utf-8'});
5   response.write('Hello ');
6   response.end('World\n');
7 }).listen(8080, '127.0.0.1');
8 console.log('Webserver wird ausgeführt.');
```

- Client- und Serverfunktionalitäten für einen Node.js-Knoten
- Einbindung über **require('http')**
- Erzeugung eines Webservers durch **createServer**-Methode des **http**-Objekts
- Übergabe einer Callback-Funktion bei Erzeugung des Servers, die bei jeder HTTP-Anfrage durch die Laufzeitumgebung aufgerufen wird
- **request**: Zugriff auf Header-Felder und Inhalte der HTTP-Anfrage
- **response**: Zugriff auf Header-Felder und Inhalte der HTTP-Antwort
- Zusammenstellen einer HTML-Seite mittels **writeHead** (für Header-Parameter) bzw. **write** auf dem **response**-Objekt

7.3 KOMMUNIKATION MIT NODE.JS

HTTP-MODUL: BEISPIELE (II)



```
1 var http = require('http');
2 http.createServer(function (request, response) {
3   response.writeHead(200, {
4     'content-type': 'text/html; charset=utf-8'});
5   var body = '<!DOCTYPE html>' +
6     '<html>' +
7       '<head>' +
8         '<meta charset="utf-8">' + '<title>Node.js Demo</title>' +
9       '</head>' +
10      '<body>' +
11        '<h1 style="color:green">Hello World</h1>' +
12      '</body>' +
13      '</html>';
14   response.end(body);
15 }).listen(8080, '127.0.0.1');
16 console.log('Webserver wird ausgeführt.');
```

```
1 var http=require('http');
2 var url=require('url');
3 http.createServer(function(request, response) {
4   response.writeHead(200, {
5     'content-type': 'text/html; charset=utf-8'});
6   var urlString=url.parse(request.url, true);
7   var body='Hello '+urlString.query.name;
8   response.end(body);
9 }).listen(8080, '127.0.0.1');
10 console.log('Webserver wird ausgeführt.');
```

7.3 KOMMUNIKATION MIT NODE.JS

HTTP-MODUL: BEISPIELE (III)

ERZEUGUNG EINES WEBSERVERS MIT `createServer`

```
1 var http = require('http');
2
3 http.createServer(function (request, response) {
4   response.writeHead(200, {
5     'content-type': 'text/plain; charset=utf-8'});
6   response.write('Hello ');
7   response.end('World\n');
8 }).listen(8080, '127.0.0.1');
```

- `createServer`-Methode erzeugt im Hintergrund einen Webserver
- Übergabe einer Callback-Funktion, die bei einem eintreffenden Request aufgerufen wird
- Schreiben des Antwort-Headers mittels `writeHead`

ERZEUGUNG AUF KONVENTIONELLE ART

```
1 var Server = require('http').Server;
2
3 var server = new Server();
4 server.addListener('request', function (request, response) {
5   response.statusCode = 200,
6   response.setHeader('content-type', 'text/plain; charset=utf-8');
7   response.write('Hello World');
8   response.end();
9 });
10 server.listen(8080, 'localhost');
```

- Alternative Erzeugung des Webservers über eine Konstruktorfunktion
- Registrierung der Callback-Funktion mittels **addListener**-Methode auf dem **server**-Objekt
- **addListener**: gleiche Funktionsweise wie **addEventListener** bei JavaScript im Browser
- **addListener** ist eine Alias für **on**
- Weitere Methoden zum Schreiben des Antwort-Headers



7.3 KOMMUNIKATION MIT NODE.JS STATISCHE HTML-SEITEN

```
1 var http = require('http');
2 var fs = require('fs');
3 var url = require('url');
4
5 http.createServer( function (request, response) {
6     var pathname = url.parse(request.url).pathname;
7     console.log("Request for " + pathname + " received.");
8     fs.readFile(pathname.substr(1), function (err, data) {
9         if (err) {
10             console.log(err);
11             response.writeHead(404, {'Content-Type': 'text/html'});
12         } else {
13             response.writeHead(200, {'Content-Type': 'text/html'});
14             response.write(data.toString());
15         }
16         response.end();
17     });
18 }).listen(8080);
```



- Bisher: dynamische Erzeugung einer HTTP-Antwort
- Hier: Übertragung einer statischen Datei (text oder HTML) aus dem Dateiverzeichnis des Servers
- Benötigte Module: **http**, **url**, **fs**
- **url**: Operationen auf URLs, z.B. Extraktion des Pfadnamens
- **fs**: Operationen auf dem lokalen Dateisystem, zum Beispiel Lesen oder Schreiben von Dateien
- Beachte: Lesen einer Datei erfolgt asynchron, d.h. Verarbeitung der gelesenen Daten durch Callback-Funktion



<https://nodejs.org/api/modules.html>

7.3 KOMMUNIKATION MIT NODE.JS

AUSLESEN DES NACHRICHTENKÖRPERS EINER ANFRAGE

```
1 var http = require('http');
2
3 http.createServer(function(request, response) {
4   var body = '';
5   request.on('data', function(data) {
6     body += data.toString();
7   });
8   request.on('end', function() {
9     console.log(body);
10    response.writeHead(200, {'content-type': 'text/plain'});
11    response.end(body);
12  });
13 }).listen(8080, 'localhost');
```

- Problem: Body einer HTTP-Anfrage kann in mehreren Teilen übertragen werden, d.h. in mehreren TCP-Segmenten
- Betroffen sind **POST**- und **PUT**-Nachrichten
- Verschiedene Events informieren Node.js-Server über den Zustand einer HTTP-Anfrage
 - **data**: wird ausgelöst, sobald neue Daten geliefert werden
 - **end**: wird ausgelöst, sobald die Übermittlung sämtlicher Teile einer Anfrage abgeschlossen ist
 - **close**: signalisiert, dass die TCP-Verbindung beendet wurde
- Bearbeitung der Ereignisse durch Callback-Funktionen
- Beispiel: die in einem TCP-Segment übermittelten Daten werden in der Callback-Funktion übergeben und lassen sich in einem Buffer zwischenspeichern
- Beachte: **end** und **close** können pro Anfrage nur einmal ausgelöst werden
- Beachte: tritt **close** vor **end** auf, ist Übertragung unvollständig
- Beachte: **on**-Methode alternativ für **addListener**

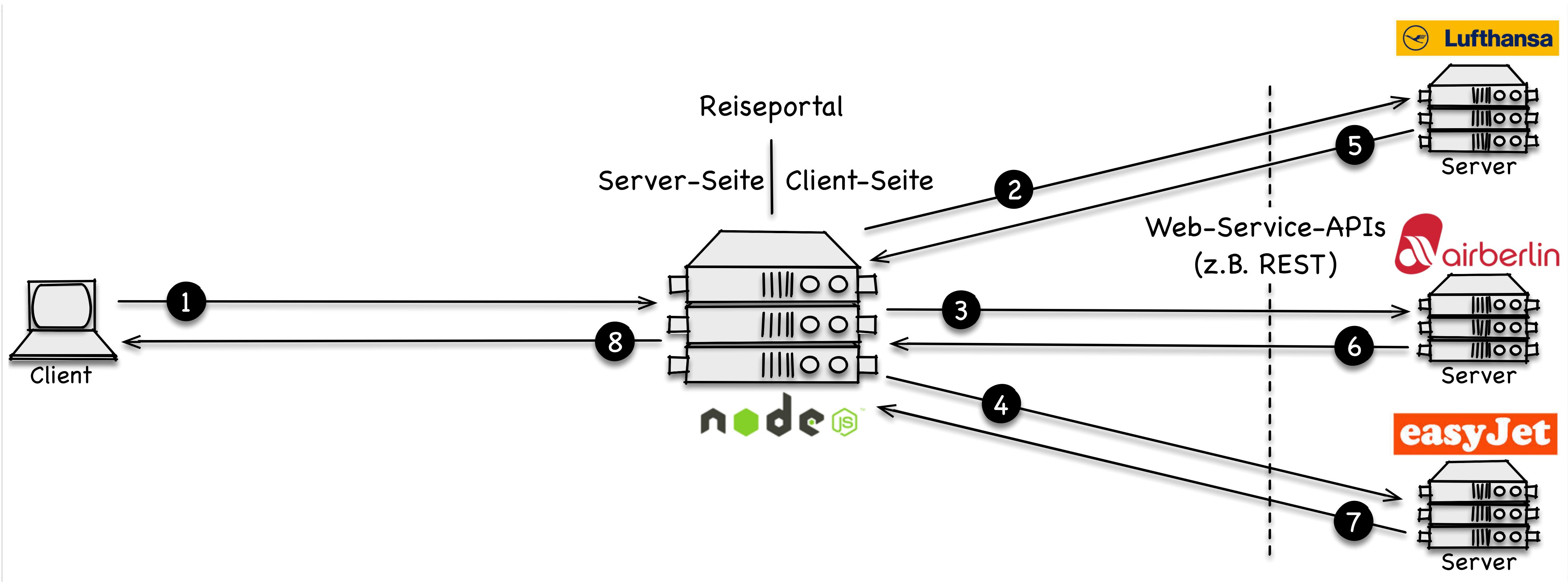


<https://nodejs.org/api/modules.html>

7.3 KOMMUNIKATION MIT NODE.JS

NODE.JS ALS CLIENT (I)

Szenario



7.3 KOMMUNIKATION MIT NODE.JS

NODE.JS ALS CLIENT (II)

Erstellung eines Clients mit `request()`

```
1 var http = require('http');
2 http.request('http://www.google.de', function (response) {
3   response.on('data', function (data) {
4     console.log(data.toString());
5   });
6 }).end();
```

Verwendung von `request()` mit mehreren Optionen

```
1 var http = require('http');
2 http.request({
3   host: 'www.google.de', port: 80, method: 'GET',
4   path: '/?q=node.js',
5   headers: {
6     'accept' : 'text/html',
7     'userAgent' : 'node.js'
8   },
9   agent: false}, function (response) {
10   response.on('data', function (data) {
11     console.log(data.toString());
12   });
13 }).end();
```

Erstellung eines Clients mit `get()`

```
1 var http = require('http');
2 http.get('http://www.google.de', function (response) {
3   response.on('data', function (data) {
4     console.log(data.toString());
5   });
6 });
```

- Erstellung eines Node.js-Clients der Anfragen an Server richtet
- Bisher: Methode **createClient()** (mittlerweile aber deprecated)

request() -METHODE

- **request()** initiiert Anfrage, die asynchron ausgeführt wird
- Ausführung von **request()** mit verschiedenen Optionen
- Abholen der Antwort über Callback-Funktion auf dem **response**-Objekt
- Auslesen der Antwort-Daten mittels Events: **data**, **end**, **close**
- Aufruf der **end()**-Methode zum Absenden der Abfrage

get() -METHODE

- Setzt automatisch **GET**-Methode und sendet Anfrage automatisch ab
- Keine Pedants für **POST**, **PUT**, **DELETE**

7.3 KOMMUNIKATION MIT NODE.JS

NODE.JS ALS CLIENT (III)

DATEN ZUM SERVER SENDEN

```
1 var http = require('http');
2
3 var request = http.request({
4   host: 'localhost',
5   port: 8080,
6   method: 'POST',
7   path: '/users',
8   headers: {
9     'Transfer-Encoding': 'chunked'
10  }
11 }, function (response) {});
12
13 request.write('[{"firstname": "James", "surname": "Bond"},]');
14 request.write('{"firstname": "Sherlock", "surname": "Holmes"}]');
15
16 request.end();
```

UMGANG MIT DER ANTWORT EINES SERVERS

```
1 var http = require('http');
2
3 var request = http.request({
4   host: 'localhost',
5   port: 8080,
6   method: 'GET',
7   path: '/'
8 });
9
10 response.on('response', function (response) {
11   if (response.statusCode === 200 &&
12     response.headers['content-type'] === 'text/html') {
13     var body = '';
14     response.on('data', function (data) {
15       body += data;
16     });
17     response.on('end', function () {
18       console.log(body);
19     });
20   } else {
21     console.log('an error occurred');
22   }
23 });
24
25 request.end();
```



<https://nodejs.org/api/modules.html>

7.4 REST WAS IST DAS DENN?



Roy Fielding

Begründer des REST-Paradigmas

- Representational States Transfer (REST) ist ein Paradigma für die Gestaltung von Web-basierten Programmierschnittstellen (Application Programming Interfaces, APIs), vornehmlich für die Maschine-zu-Maschine-Kommunikation
- Alternative zu anderen Web-Service-Technologien wie SOAP und XML-Remote-Procedure-Call
- Webdienste, die dem REST-Paradigma folgen, werden als RESTful bezeichnet

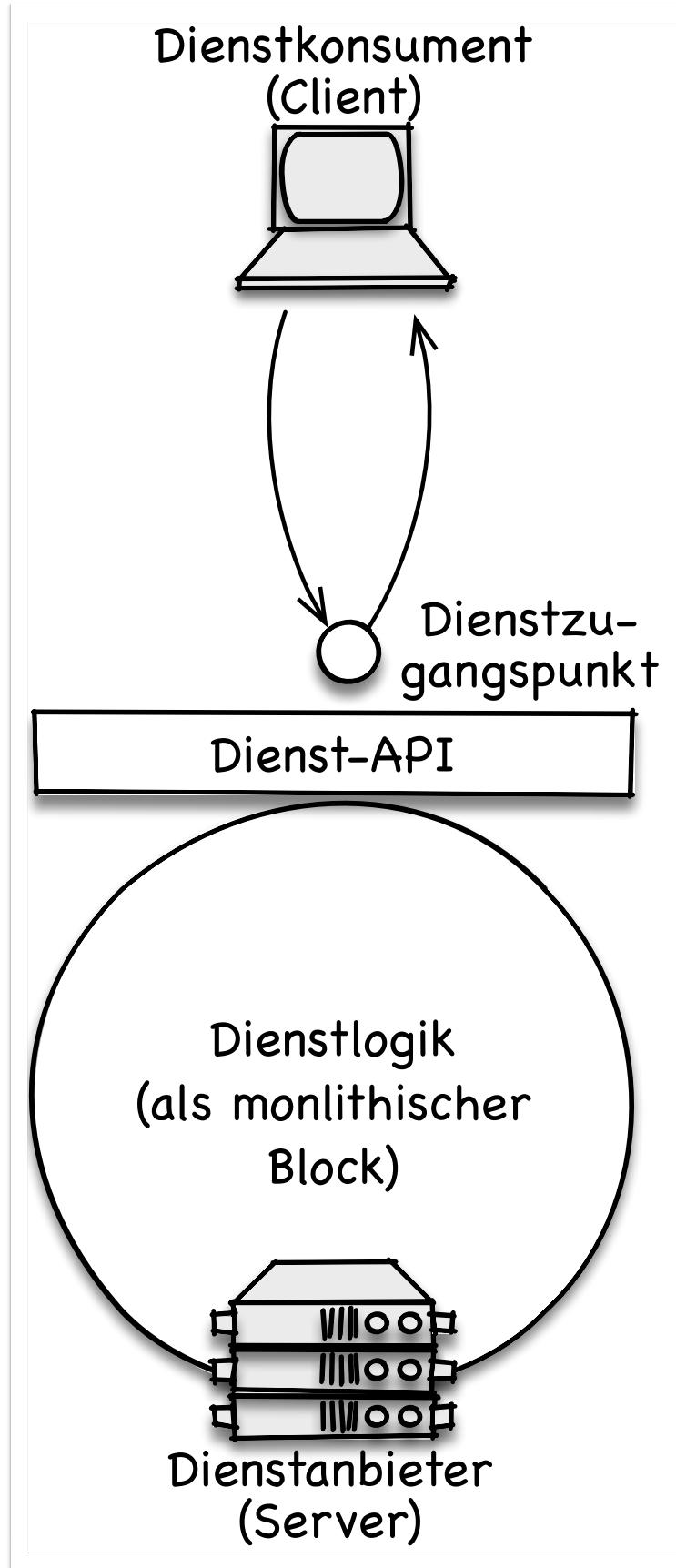
ANSATZ

- Dienstanfragen und -antworten basieren auf der Repräsentation von Ressourcen
- Eine Ressource ist jede Art von kohärenter und sinnvoller Information (z.B. eine Liste oder Elemente einer Liste), die über eine URI zur Verfügung gestellt werden kann
- Eine Repräsentation ist die Darstellungsform einer Ressource (z.B. HTML, XML oder JSON)

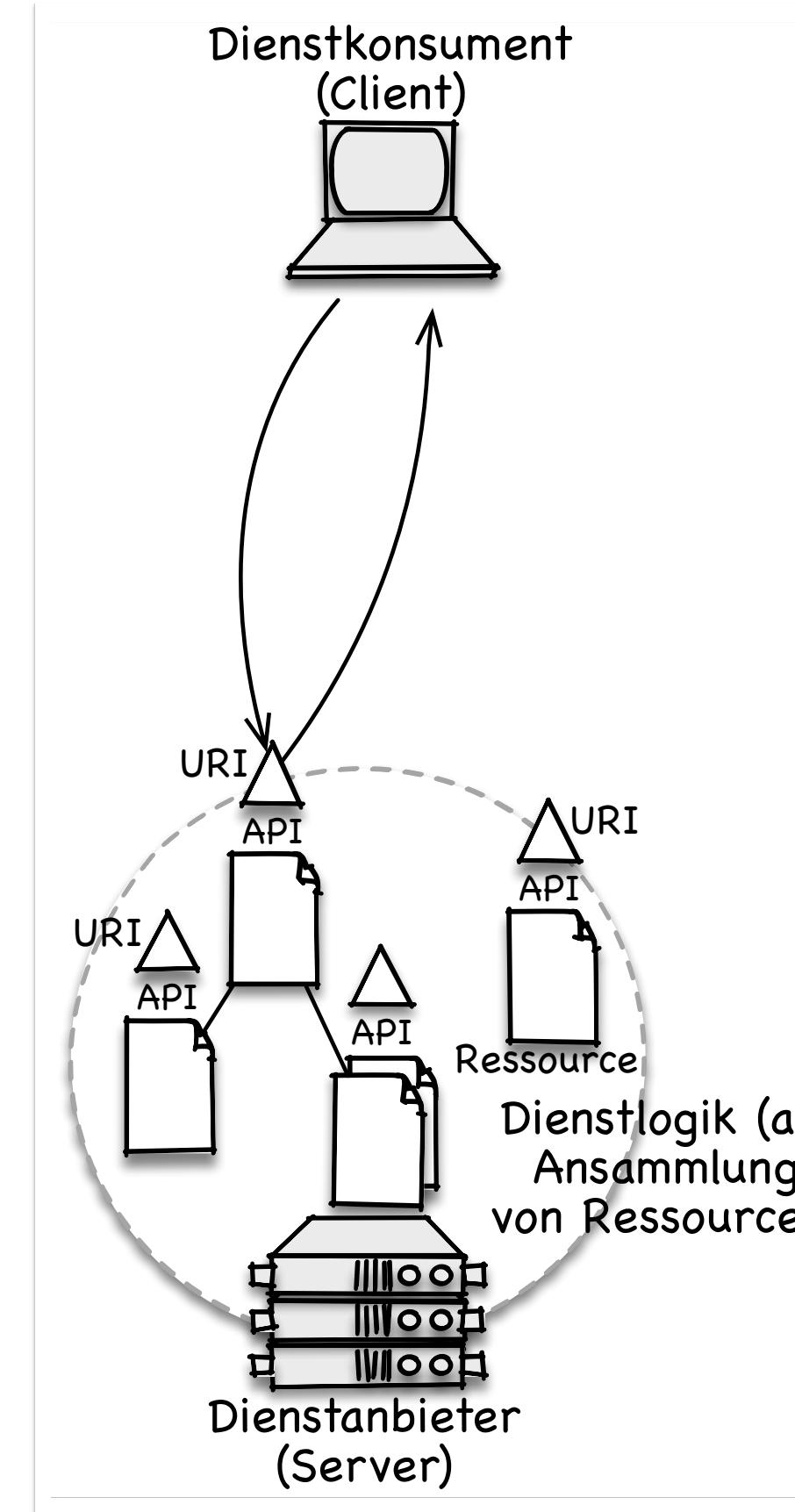
7.4 REST

WAS IST EIN DIENST?

LOGISCHER AUFBAU EINES RESTLESS-DIENSTES



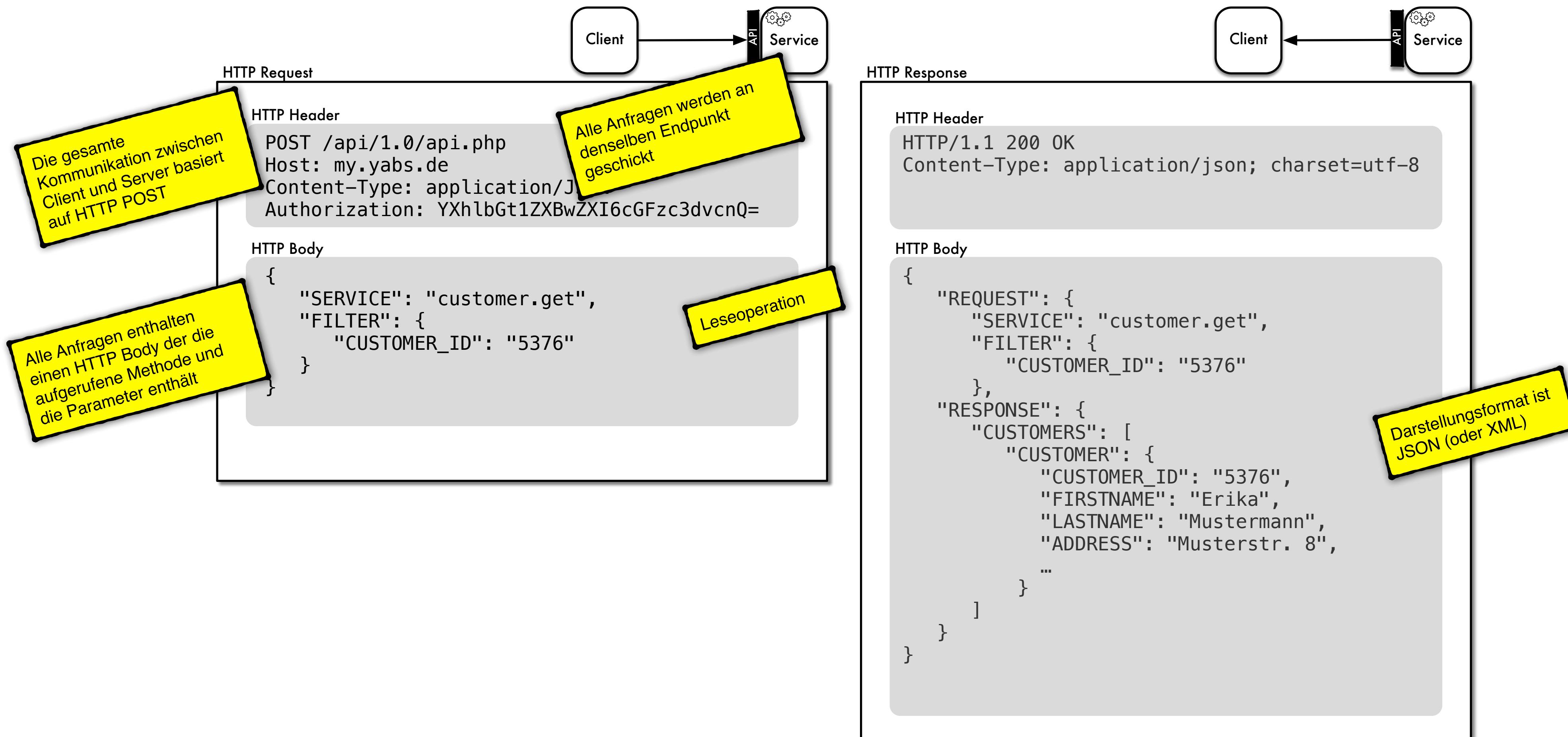
LOGISCHER AUFBAU EINES RESTFUL-DIENSTES



- Wiederholung von Kapitel 1: Ein Webdienst ist ein Programm, das über das Internet oder ein Intranet anderen Programmen Funktionalitäten zur Verfügung stellt
- Ein Dienstkonsument ist die Software, die einen Dienst aufruft
- Eine API (Application Programming Interface, Programmierschnittstelle) ist die syntaktische Spezifikation der Operationen und ihrer Parameter, die ein Dienst anbietet
- Ein Dienstzugangspunkt (Endpunkt) definiert das Protokoll für den Nachrichtenaustausch und die Adresse, unter welcher der Dienst mit seiner API erreichbar ist
- Beispiele für Dienstzugangspunkte
 - HTTP und URI
 - TCP und IP-Adresse
 - SMS und Mobiltelefonnummer

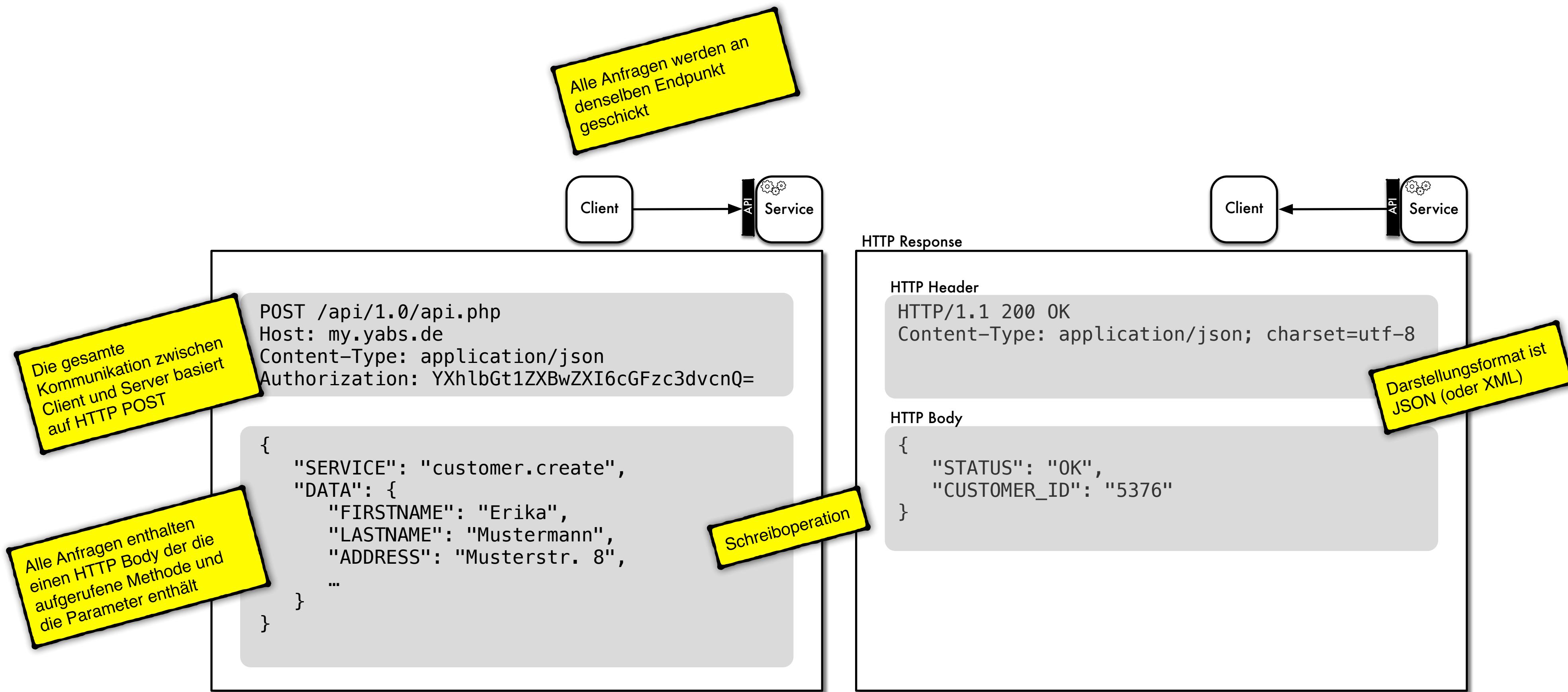
7.4 REST

BEISPIEL EINES RESTLESS-DIENSTES (I)



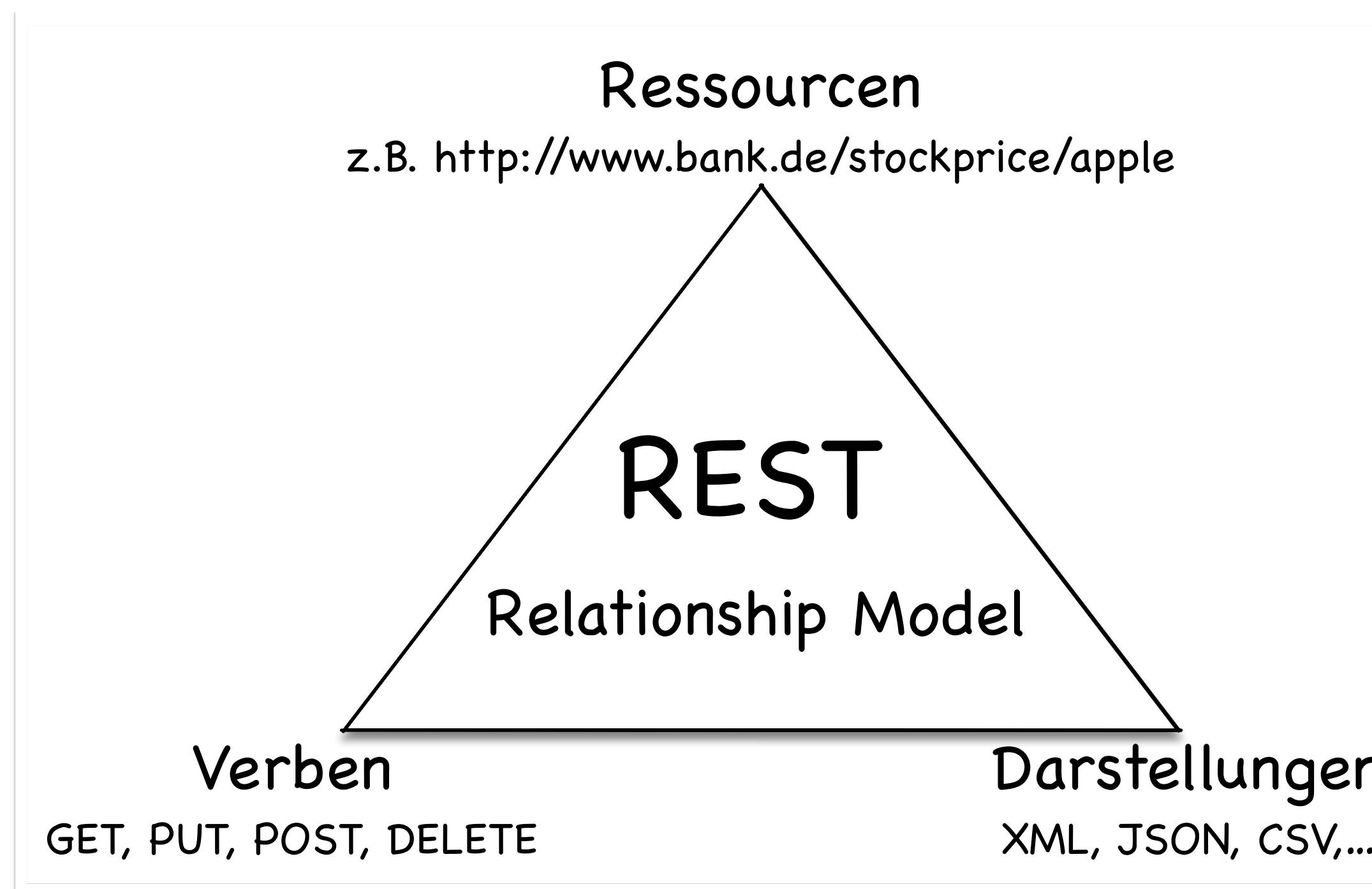
7.4 REST

BEISPIEL EINES RESTLESS-DIENSTES (II)



7.4 REST

DAS REST-RELATIONSHIP-MODELL

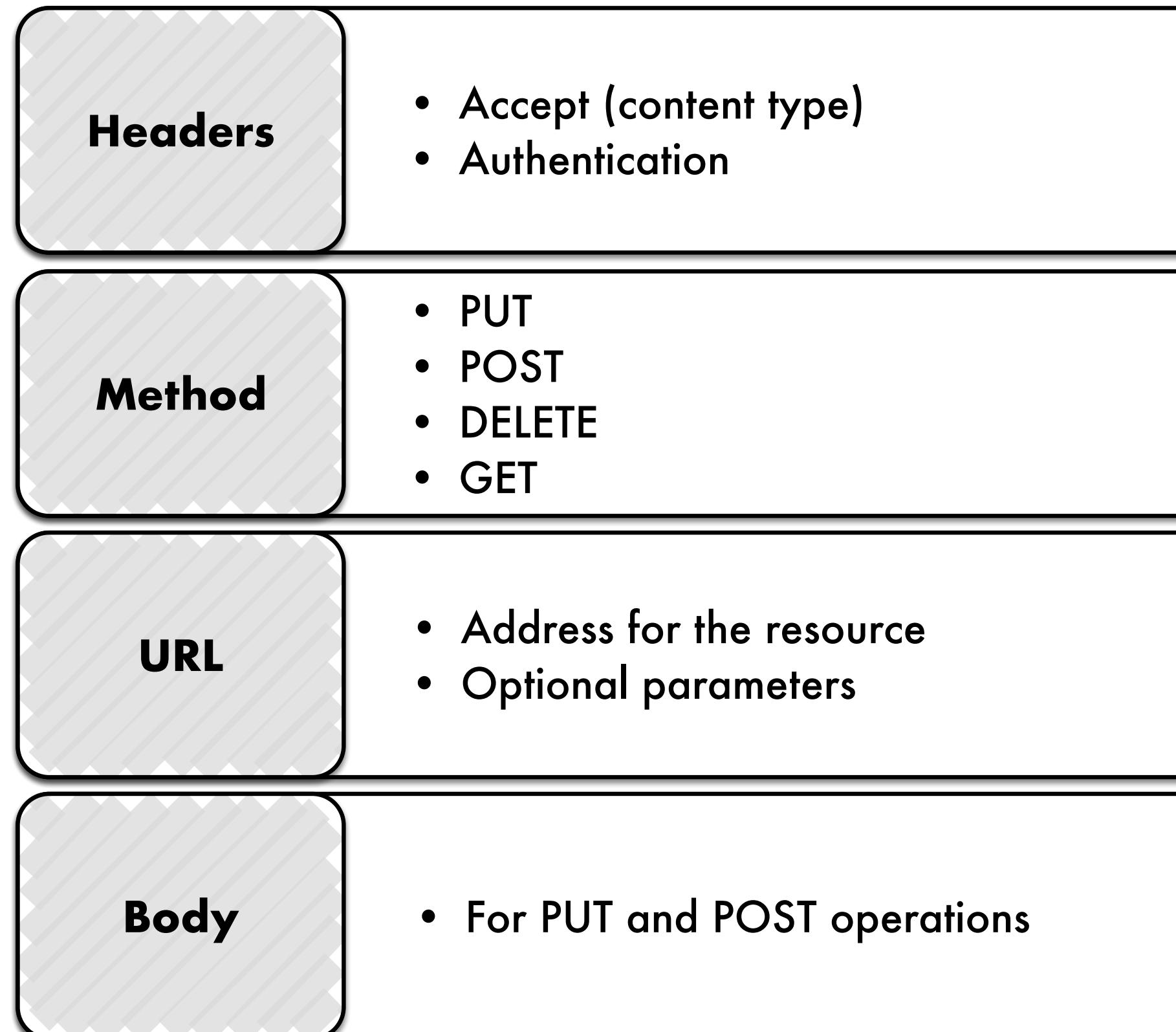


- Dienstanfragen und -antworten basieren auf der Repräsentation von Ressourcen
- Ein *Ressource* ist jede Art von kohärenter und sinnvoller Information (z.B. eine Liste oder Elemente einer Liste), die über eine Adresse (URI) zur Verfügung gestellt werden kann
- Eine *Repräsentation* ist die Darstellungsform einer Ressource (z.B. HTML, XML oder JSON)
- Eine REST-konformer Server kann je nach Anforderung verschiedene Repräsentationen einer Ressource ausliefern
- Veränderungen von Ressourcen erfolgen i.d.R. über eine Repräsentation
- Jede Ressource verfügt über eine eigene, dedizierte Adresse in Form eines *Uniform Resource Identifiers* (URI)
- RESTful Services definieren keine eigenen Methoden und Operationen in ihrer API, sondern verwenden die Methoden von HTTP

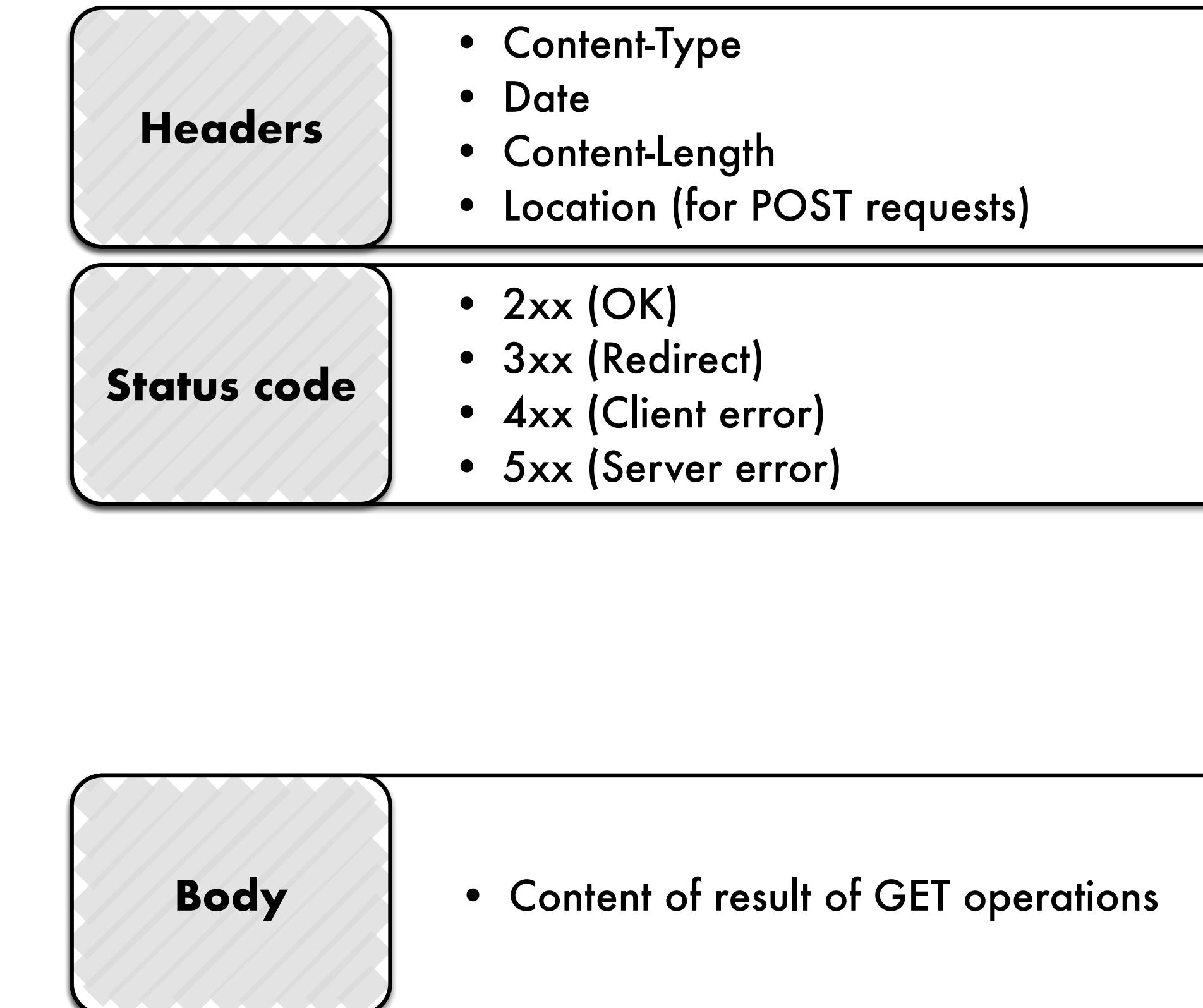
7.4 REST

WIEDERVERWENDUNG DER HTTP-METHODEN

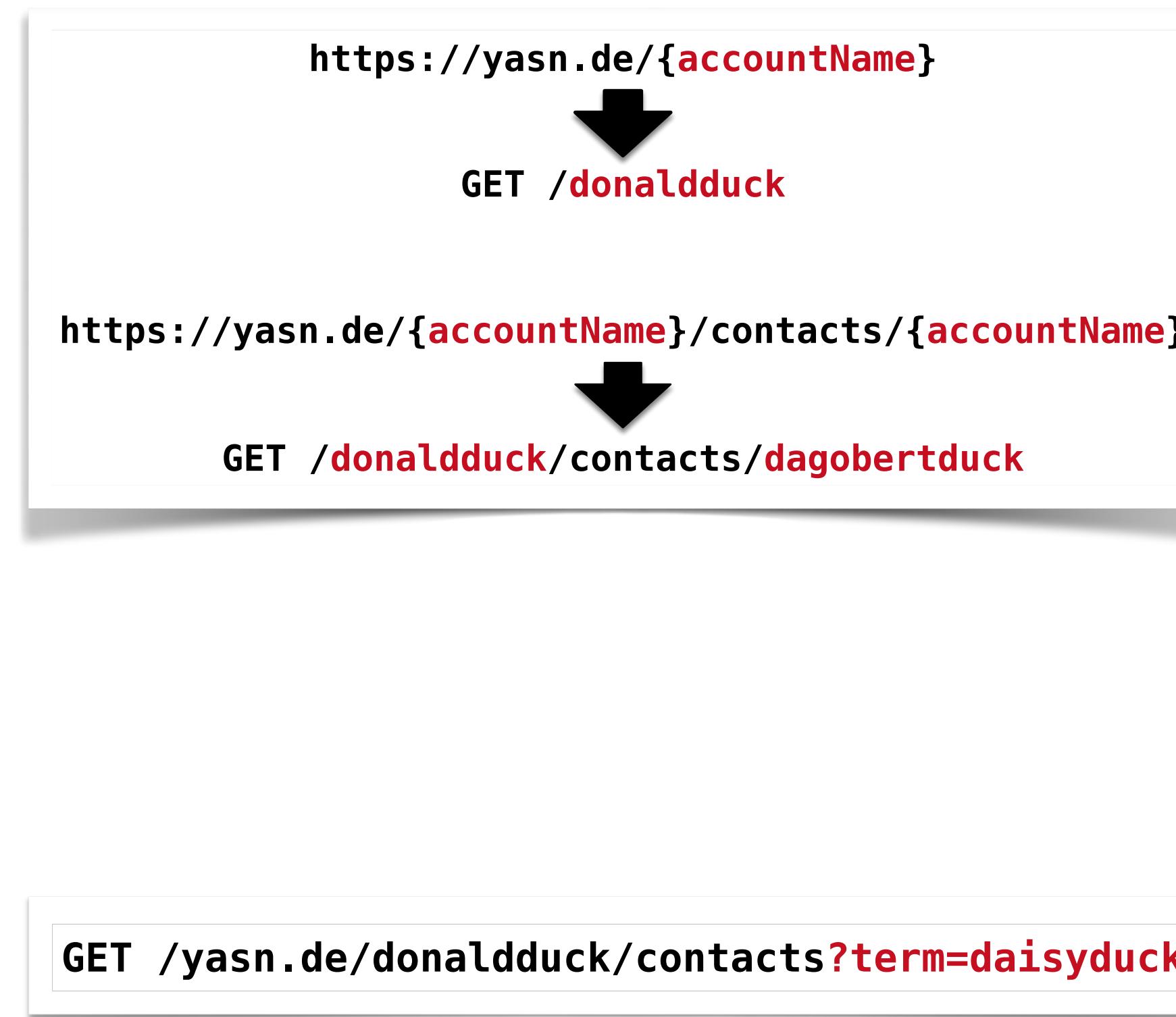
STRUCTURE OF A HTTP REQUEST



STRUCTURE OF A HTTP RESPONSE



7.4 REST REST-PARAMETER (I)



PATH PARAMETERS

- Platzhalter für den variablen Teil eines URL-Pfades
- Zeigt auf eine bestimmte Ressource in einer Kollektion von Ressourcen
- Wird durch einen Wert ersetzt, wenn der Client einen Request an die API sendet

QUERY PARAMETERS

- Schlüssel/Wertpaare, die am Ende einer URL mit einem Fragezeichen angehängt werden
- Werden üblicherweise verwendet, um Filter- oder Suchkriterien auf einer mit GET angefragten Kollektion von Ressourcen zu definieren

7.4 REST REST-PARAMETER (II)

HEADER PARAMETERS

- Werden im Header einer HTTP-Anfrage transportiert
- Werden zu Kontroll- und Steuerzwecken verwendet, z.B. Authentifizierung, Mitteilung über die URL einer Ressource, Anzeige von Inhaltsformaten, usw.

COOKIE PARAMETERS

- Spezielles Header-Feld zur Übertragung einer oder mehrerer Schlüssel-/Wertpaare
- **Set-Cookie**-Header sendet ein neues oder aktualisiertes Cookie in einer HTTP-Antwort an den Client
- **Cookie**-Header sendet das Schlüssel-/Wertpaar in den darauf folgenden HTTP-Anfragen
- Wird u.a. verwendet um wiederkehrende Nutzer einer Webanwendung beim Server zu erkennen (Sitzungsmanagement)

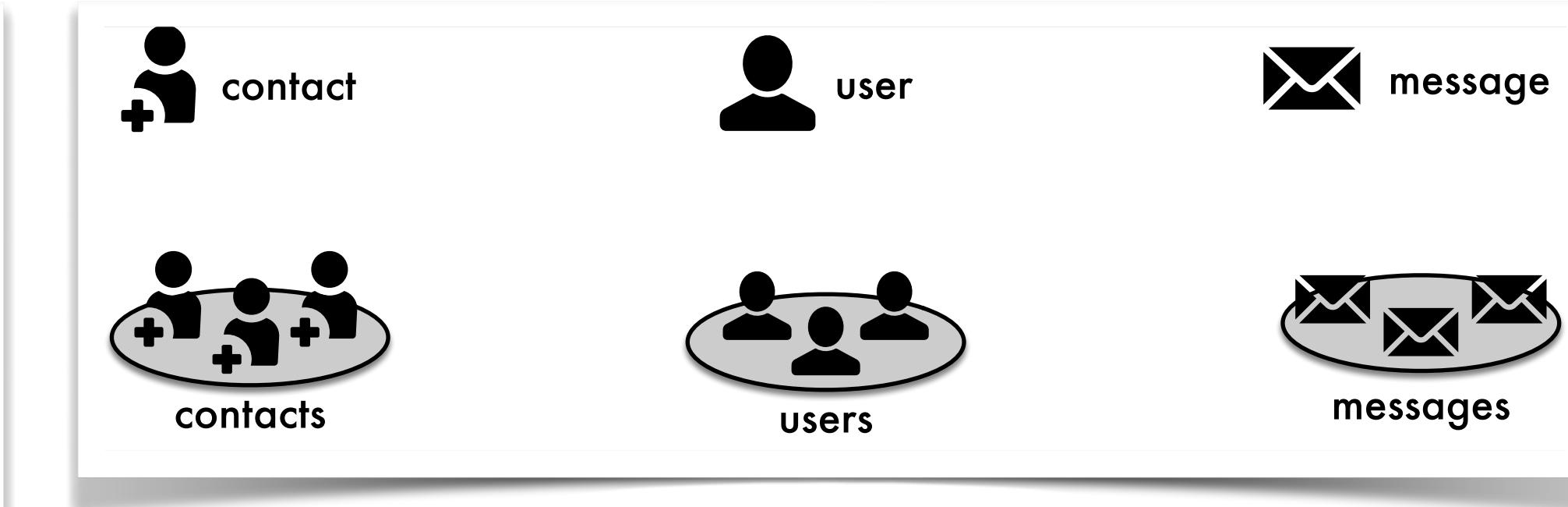
```
GET /donaldduck/timeline/123456
Host: yasn.de
Accept: application/json
Accept-Language: de-DE, de
Connection: keep-alive
Cookie: sessionId=dk81qvJ843Qyf653
```

7.4 REST REST-PARAMETER (II)

EINE RESSOURCE IN JSON-DARSTELLUNG

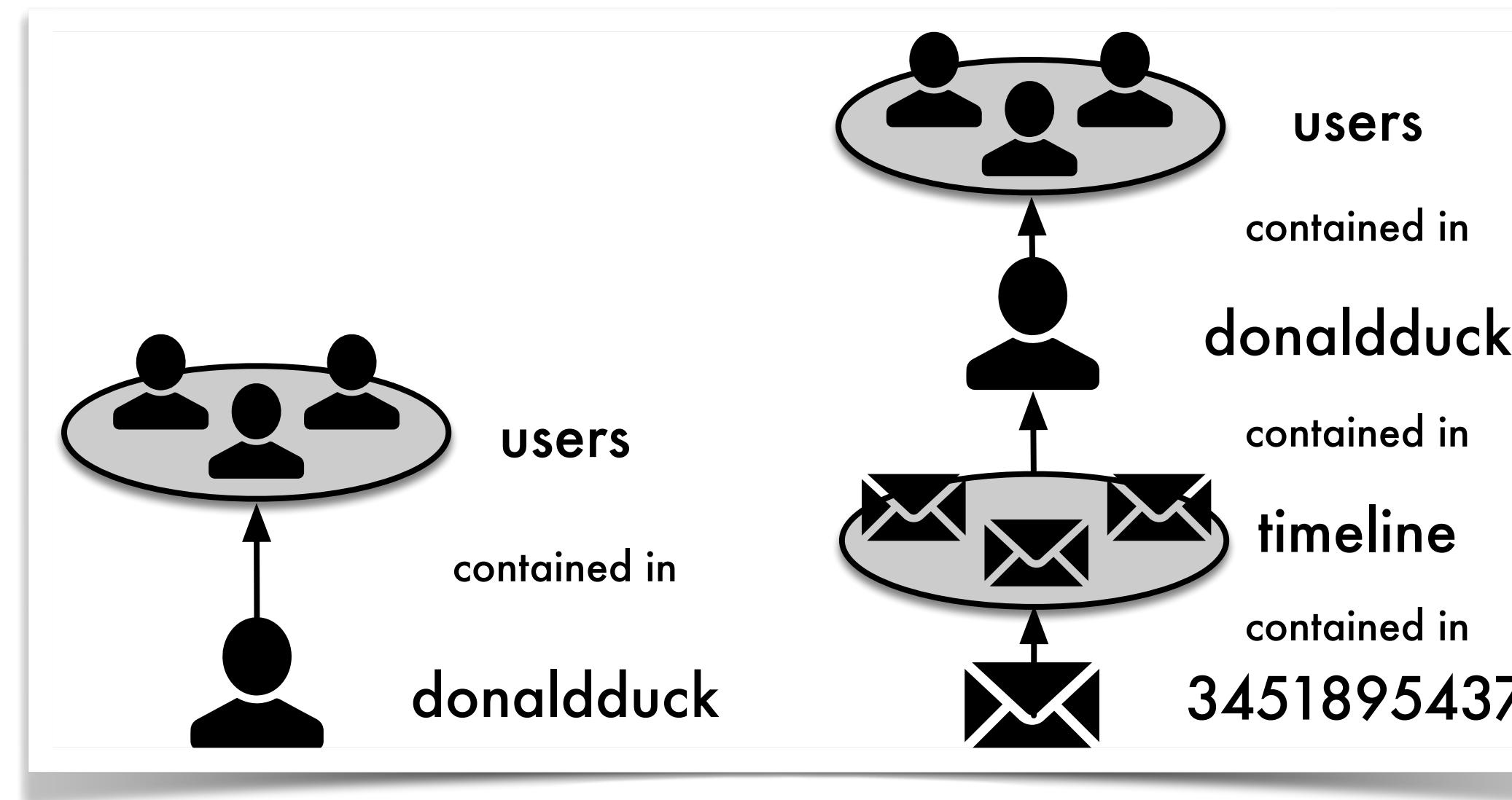
```
{  
    "accountName": "donaldduck",  
    "accountType": "premium",  
    "firstname": "Donald",  
    "lastname": "Duck",  
    "address": "Musterstr. 8",  
    "zip": "10405",  
    "city": "Entenhausen",  
    "countryCode": "de",  
    "phone": "+4916090990909",  
    "email": "donald@duck.de",  
    "vatId": "1234567890",  
    "paymentType": "PayPal",  
    "bankName": "ING",  
    "IBAN": "DE01 1234 5678 9123 4567 89"  
}
```

BEISPIELE VON RESSOURCEN-TYPEN



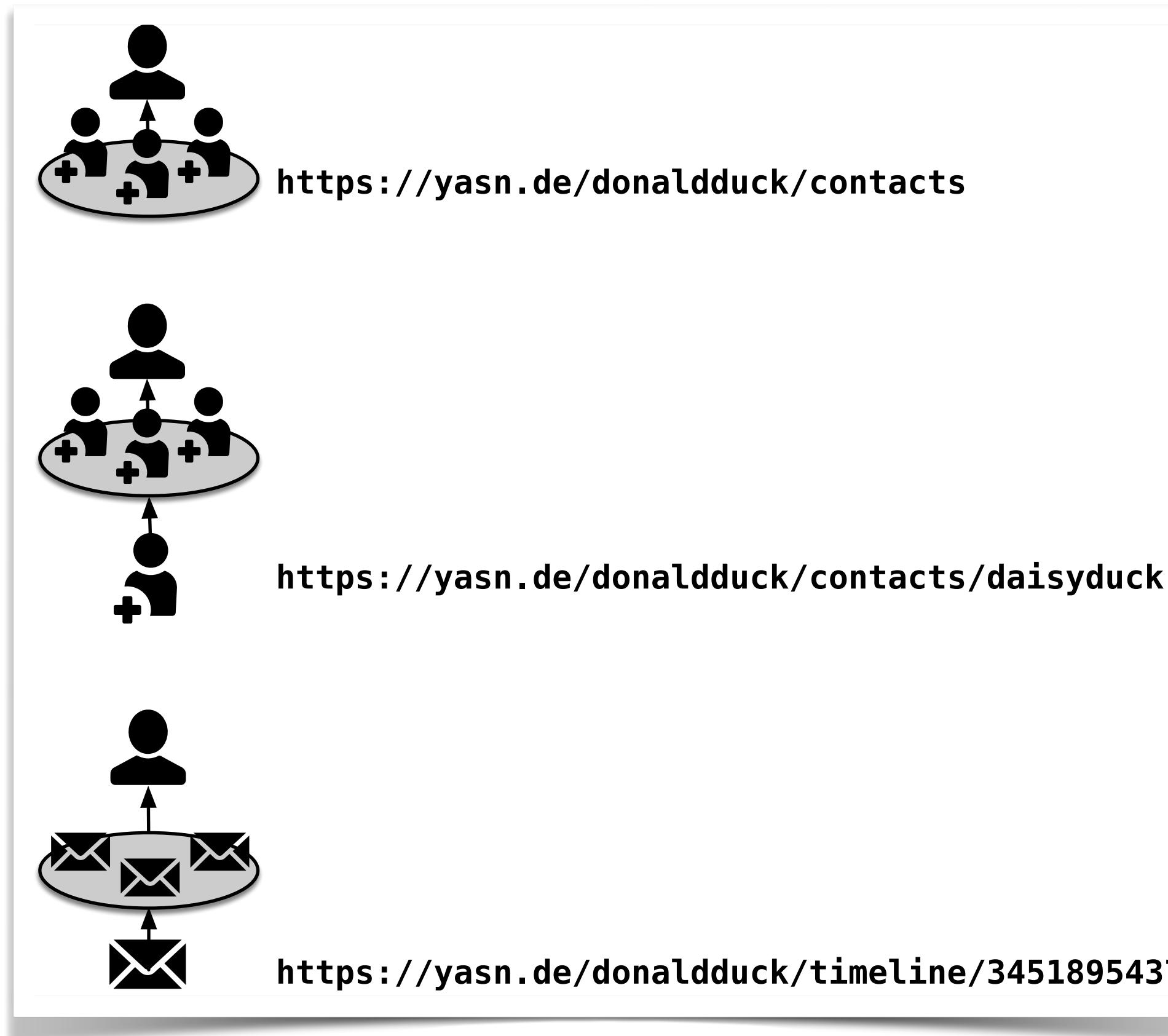
- Ein *RESTful Service* ist eine Kollektion von Ressourcen
- Eine *Ressource* ist ein kohärenter Datensatz basierend auf einem abstrakten Datenmodell, der im Kontext der Webanwendung von Relevanz ist
- Ein *Ressourcen-Typ* ist eine Kollektion von Ressourcen, die auf demselben Datenmodell basieren, d.h. Ressourcen desselben Typs werden durch dieselben Attribute beschrieben, aber unterscheiden sich durch die Attributwerte voneinander

7.4 REST ENTHALTENSEINSBEZIEHUNG ZWISCHEN RESSOURCEN (I)



- Eine Ressource kann ein oder mehrere Unterressourcen oder Kindressourcen (Sub Resources, Child Resources) enthalten
- Eine Ressource kann keine oder eine übergeordnete Superressource oder Elternressource (Super Resource) haben
- Kindressourcen und ihre Elternressource begründen eine Enthaltsseinsbeziehung

7.4 REST ENTHALTENSEINSBEZIEHUNG ZWISCHEN RESSOURCEN (II)



- Eine RESTful-Dienst hat keinen einzelnen Endpunkt, stattdessen verfügt jede Ressource über einen eigenen, durch eine URL spezifizierten Endpunkt
- Die URL setzt sich aus der Domain des RESTful-Dienstes und dem Pfad der Ressource zusammen
- Der Pfad der URL setzt sich aus den Namen der Ressource und ihrer Superressourcen zusammen

EXAMPLES

- **/donaldduck/contacts/** repräsentiert die Ressource der Kontakte des Nutzers Donald Duck
- **/donaldduck/contacts/daisyduck** repräsentiert einen einzelnen Kontakt des Nutzers Donald Duck
- **/donaldduck/timeline/3451895437** repräsentiert eine Nachricht in der Timeline des Nutzers Donald Duck

7.4 REST

REPRÄSENTATION VON RESSOURCEN

JSON

```
{  
    "accountName": "donaldduck",  
    "accountType": "premium",  
    "firstname": "Donald",  
    "lastname": "Duck",  
    "address": "Musterstr. 8",  
    "zip": "10405",  
    "city": "Entenhausen",  
    "countryCode": "de",  
    "phone": "+4916090990909",  
    "email": "donald@duck.de",  
    "vatId": "1234567890",  
    "paymentType": "PayPal",  
    "bankName": "ING",  
    "IBAN": "DE01 1234 5678 9123 4567 89"  
}
```

XML

```
<User>  
    <accountName>donaldduck</accountName>  
    <accountType>premium</accountType>  
    <firstname>Donald</firstname>  
    <lastname>Duck</lastname>  
    <address>Musterstr. 8</address>  
    <zip>10405</zip>  
    <city>Entenhausen</city>  
    <countryCode>de</countryCode>  
    <phone>+4916090990909</phone>  
    <email>donald@duck.de</email>  
    <vatId>1234567890</vatId>  
    <paymentType>PayPal</paymentType>  
    <bankName>ING</bankName>  
    <IBAN>DE01 1234 5678 9123 4567 89</IBAN>  
</User>
```

- Eine Repräsentation oder Darstellung ist eine serialisierte Ressource, d.h. eine Zeichenkette basierend auf einem Darstellungsformat
- Jede Ressource eines RESTful-Dienstes kann verschiedene Darstellungen haben
- Ein Darstellungsformat definiert die Syntax einer Darstellung
- Übliche Darstellungsformate: Java Script Object Notation (JSON) and eXtended Markup Language (XML)

7.4 REST

VERWENDUNG VON HTTP-METHODEN IN REST

SEMANTIK DER HTTP-METHODEN IM KONTEXT VON REST

HTTP-Methode	CRUD-Operation	Beschreibung
GET	Read	Fordert die angegebene Ressource vom Server an.
PUT	Update	Die durch die URL spezifizierte Ressource wird mit den Daten aus dem Rumpf aktualisiert oder es wird eine neue Ressource mit den Daten aus dem Rumpf angelegt, falls die angegeben URL nicht existiert.
POST	Create	Fügt eine neue Ressource unterhalb einer angegebenen Ressource ein. Da die neue Methode noch keine URL besitzt, wird die URL der Elternressource angegeben. Als Ergebnis der Operation wird die URL der neu angelegten Ressource zurückgegeben.
DELETE	Delete	Löscht die angegebene Ressource.
OPTIONS		Prüft, welche Methoden auf einer Ressource zur Verfügung stehen.
HEAD		Fordert die Metadaten (HTTP Header-Attribute) einer Ressource an.

- RESTful-Dienste basieren auf HTTP und verwenden HTTP-Methoden als Standardvokabular der API (siehe Tabelle)
- Beachte: REST ist nicht auf HTTP beschränkt, sondern kann auch mit anderen Protokollen wie CoAP (Constrained Application Protocol) genutzt werden
- HTTP-Methoden, die keine Daten verändern, werden als sicher (safe) bezeichnet
- Sichere Methoden: **GET** und **HEAD**
- HTTP-Methoden, die bei wiederholten Aufruf (mit den gleichen Parametern) das gleiche Ergebnis erzielen, werden als idempotent bezeichnet
- Idempotente Methoden: **GET**, **HEAD**, **PUT**, and **DELETE**

7.4 REST

PUT VS POST

Request	Description
PUT /users	Eine Ressource, welche eine Menge von Nutzern repräsentiert, wird mit den Daten im HTTP-Body aktualisiert wenn die Ressource bereits existiert oder unter /users angelegt wenn sie nicht existiert
PUT /users/donaldduck	Eine Ressource, die den Nutzer Donald Duck repräsentiert, wird mit den Daten im HTTP-Body aktualisiert oder unter /users/donaldduck neu angelegt
POST /users/	Legt eine neue Ressource unter der Elternressource /users mit den Daten aus dem HTTP-Body an. Der Name der Ressource wird dem aufrufenden Client im Location -Header-Feld der Antwort mitgeteilt.
POST /users/donaldduck	Legt eine neue Ressource unter der Elternressource /users/donaldduck mit den Daten aus dem HTTP-Body an. Der Name der Ressource wird dem aufrufenden Client im Location -Header-Feld der Antwort mitgeteilt.

POST **/users** Adds a new user to YASN and uploads her/his user profile. ✓ ↕

PUT **/users/{userId}** Updates the profile of the user specified by userId. ✓ ↕

7.4 REST

EINE REST-API FÜR YASN SPEZIFIZIERT MIT SWAGGER (I)

users

GET /users Returns a list of user profiles that fulfill some search criteria.

PUT /users Updates the YASN user profile database with the list of user profiles contained in the request.

POST /users Adds a new user to YASN and uploads her/his user profile.

DELETE /users Deletes the YASN database of all user profiles.

/users/{userId}

GET /users/{userId} Returns the user profile of the user with the specified userId.

PUT /users/{userId} Updates the profile of the user specified by userId.

DELETE /users/{userId} Deletes the profile of the user specified by userId.

/users/{userId}/inbox

GET /users/{userId}/inbox Returns a list with an overview of all messages contained in the inbox of the user specified by userId.

/users/{userId}/contacts

GET /users/{userId}/contacts Gets a list of contacts of the user.

POST /users/{userId}/contacts Adds a new contact to the user's list of contacts.

/users/{userId}/contacts/{contactId}

DELETE /users/{userId}/contacts/{contactId} Removes the contact from the list of contacts of a user.

/users/{userId}/inbox/{messageId}

GET /users/{userId}/inbox/{messageId} Returns a message contained in the inbox of the user.

DELETE /users/{userId}/inbox/{messageId} Removes the message the inbox of the user.

/messages

POST /messages Sends a new message

7.4 REST

EINE REST-API FÜR YASN SPEZIFIZIERT MIT SWAGGER (II)

Schemas

userProfile ▾ {

- userId* integer example: 78128923
- userName* string example: jamesbond
- firstName string example: James
- lastName string example: Bond
- email* string(\$url) example: mailto:jamesbond@mi6.gov.uk
- dateOfBirth string(\$date)
- thumbnail string example: 78128923.png
- contacts ▾ [
 - example: List [91678259, 19002347, 66126211]
 - > [...]
- address userProfile_address > {...} ↵

}

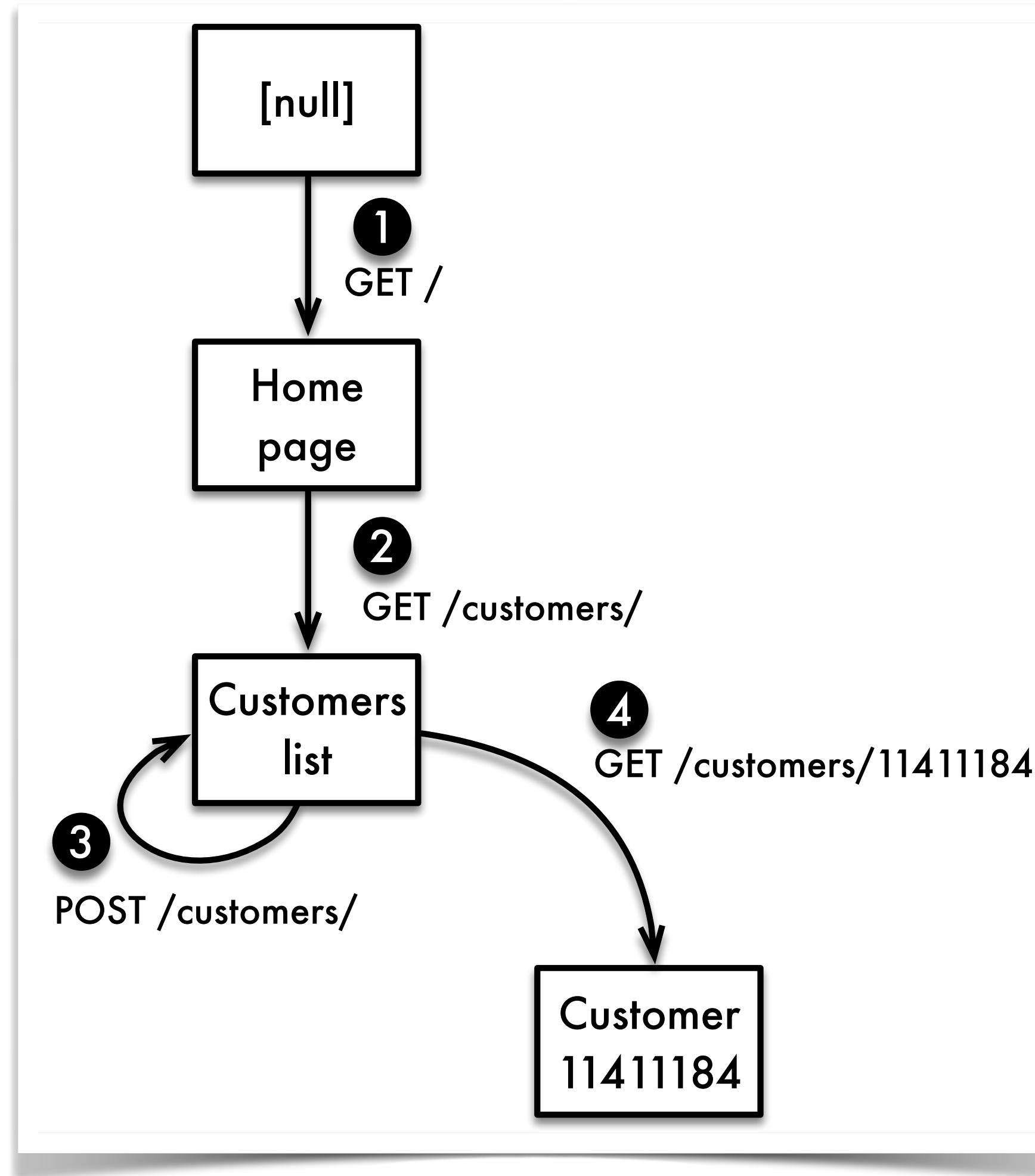
inboxItem >

inbox >

message >

userProfile_address >

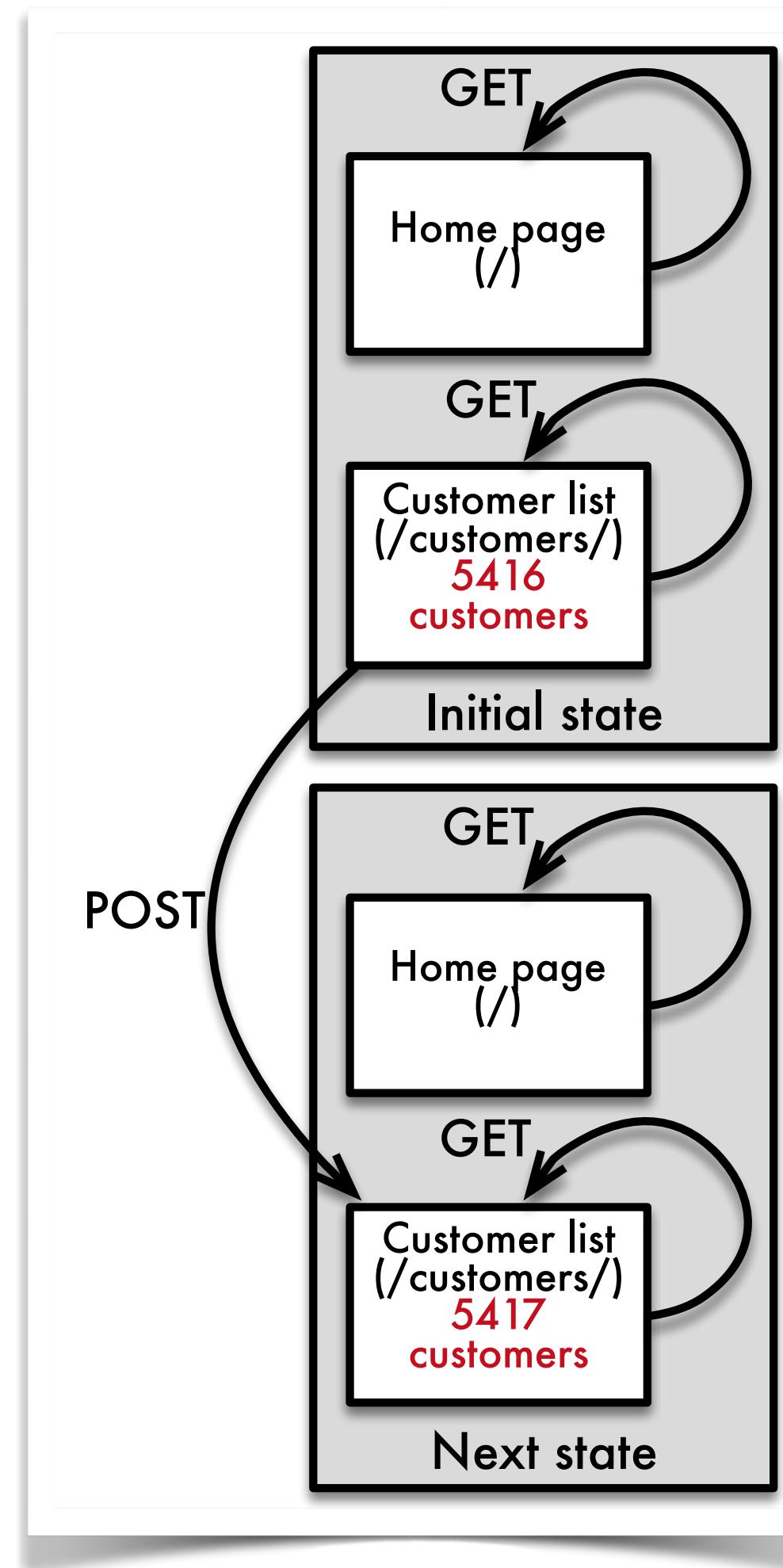
7.4 REST APPLICATION STATE



APPLICATION STATE

- Der Zustand einer Anwendung (Application State) korrespondiert mit der im Client geladenen Ressource (entspricht im WWW der geladenen Webseite)
- Zustandsübergänge entsprechen Verweisen denen der Client folgt, um weitere Ressourcen zu laden

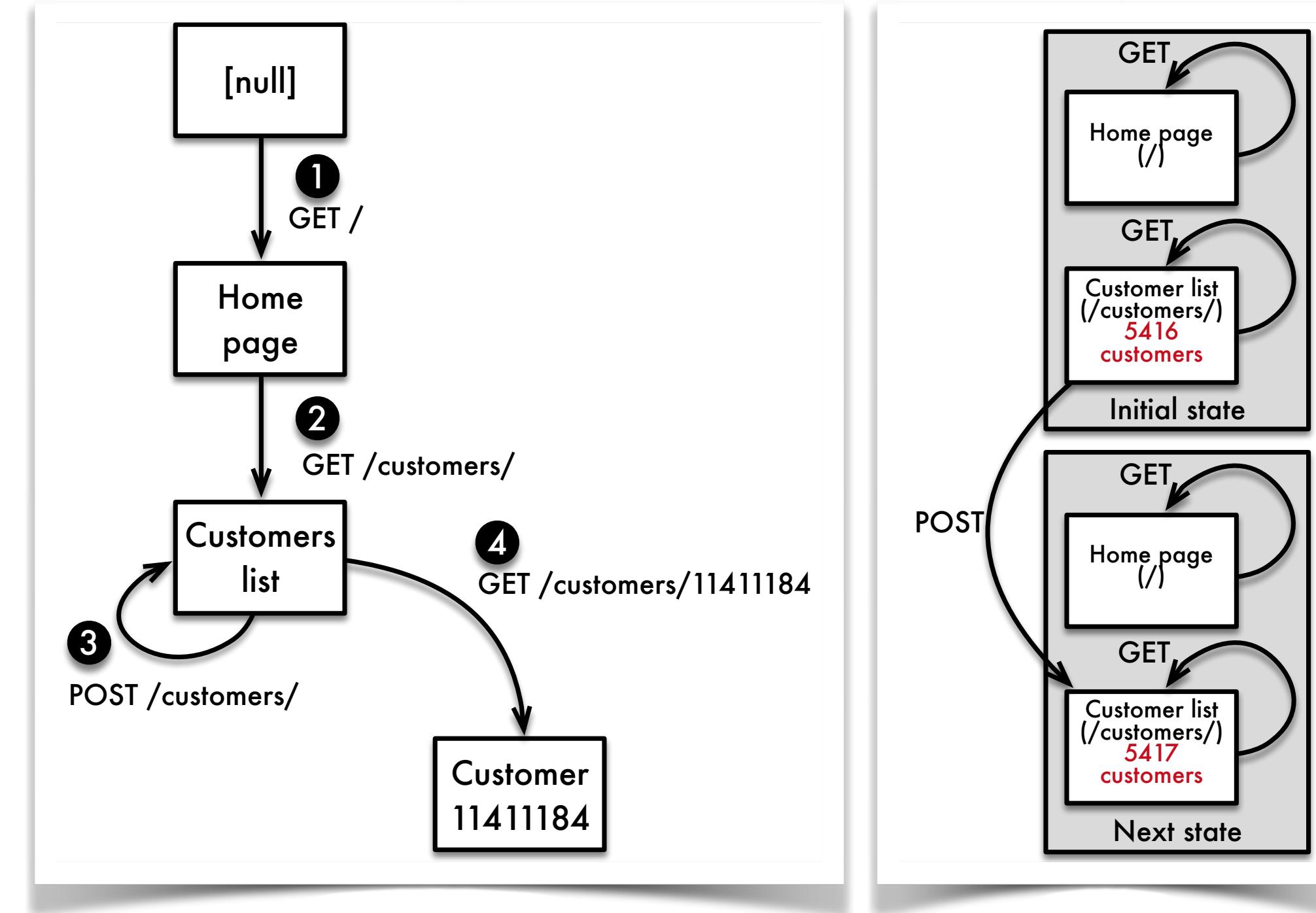
7.4 REST RESOURCE STATE



RESOURCE STATE

- Der Ressourcen-Zustand (Resource State) entspricht dem Zustand, den Daten oder dem Wert aller Ressourcen eines Webservers
- Zustandsübergänge ergeben sich durch die Aktualisierung oder die Löschung von existierenden Ressourcen sowie die Einrichtung neuer Ressourcen

7.4 REST REPRESENTATIONAL STATE TRANSFER



REPRESENTATIONAL STATE TRANSFER

- Der Zustand der Anwendung liegt im Client - Der Server kann durch Senden von Repräsentationen Zustandsübergänge auslösen und den Zustand der Anwendung beeinflussen
- Der Ressourcen-Zustand liegt im Server - Der Client kann durch Senden durch Repräsentationen den Zustand beeinflussen



VORLESUNG

Prof. Dr. Axel Küpper

TU Berlin | T-Labs | Fachgebiet Service-centric Networking
Ernst-Reuter-Platz 7 | 10587 Berlin | Germany

 axel.kuepper@tu-berlin.de

 <https://twitter.com/kuepp>

 <https://www.linkedin.com/in/axelkuepper/>

 <http://www.snet.tu-berlin.de/kuepper>

ÜBUNGSLEITER

- Thomas Cory
- Sanjeet Raj Pandey
- Christian René Sechting

TUTOREN

- Nastassia Lukyanovich
- Maximilian Oliver Fisch
- Leonhardt Frederik Hollatz
- Adrian Siebing