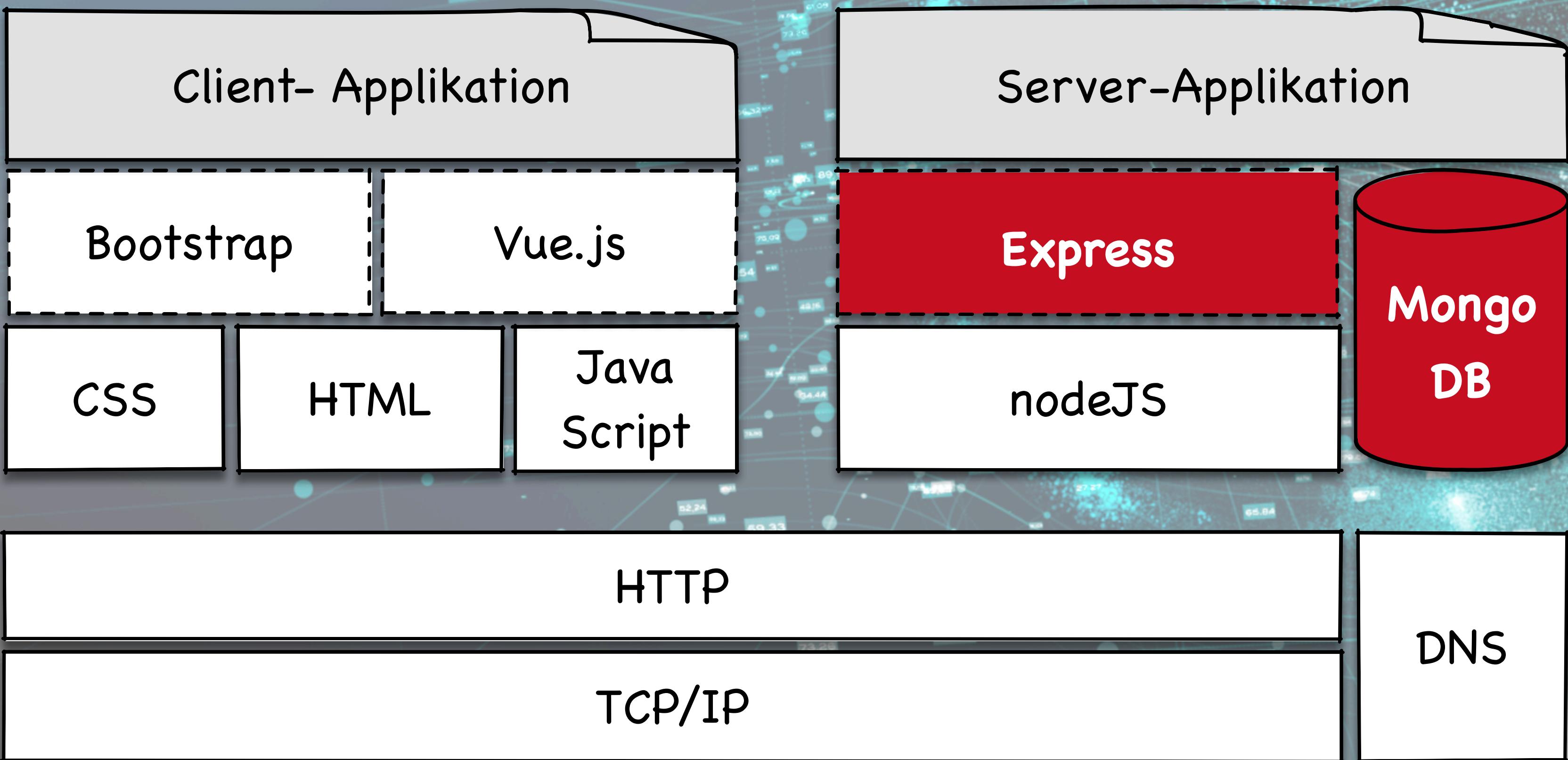


WEB TECHNOLOGIEN 2022

KAPITEL 8: SERVER FRAMEWORKS

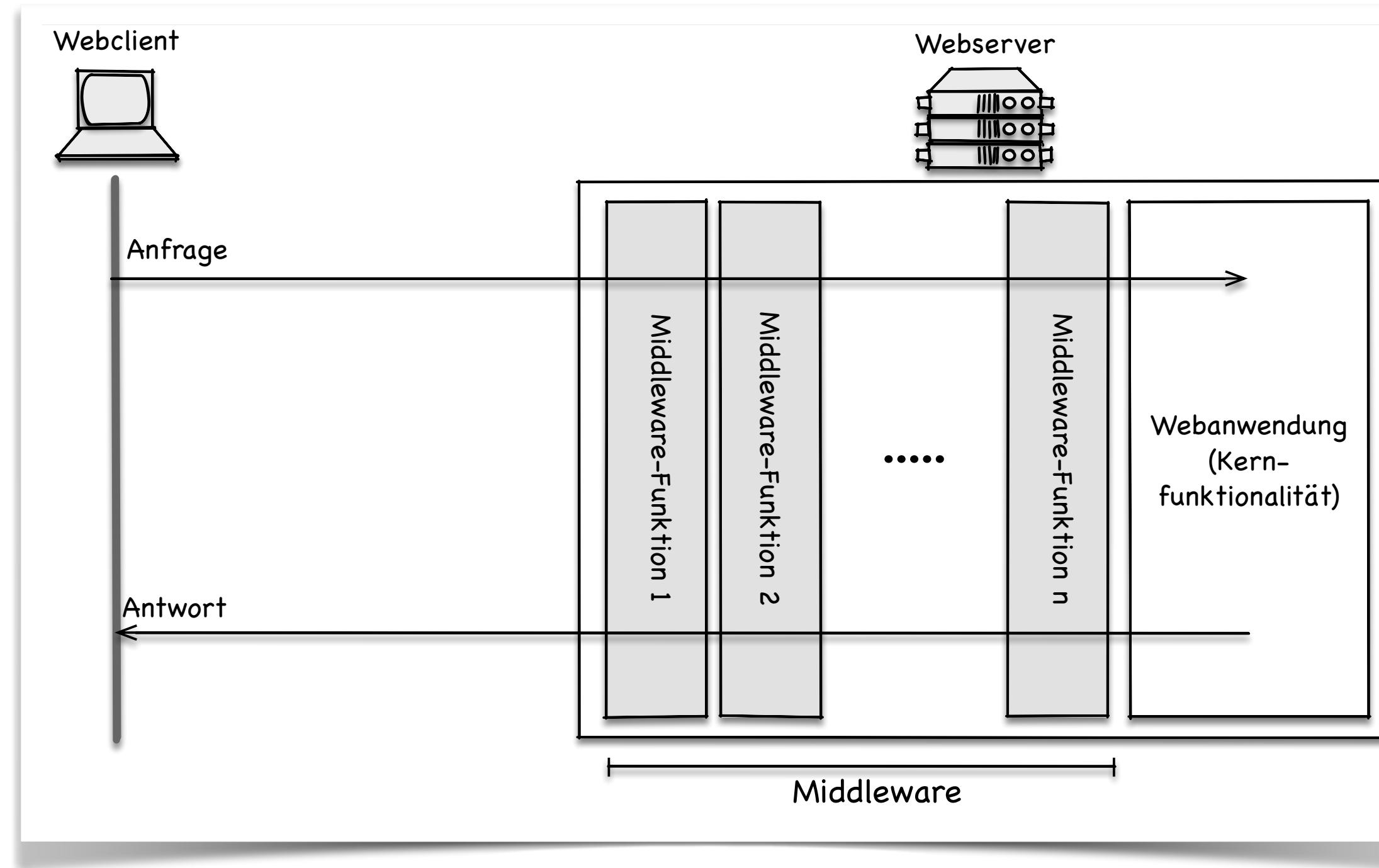
PROF. DR. AXEL KÜPPER
FACHGEBIET SERVICE-CENTRIC NETWORKING | TU BERLIN & T-LABS

WEBTECHNOLOGIEN ÜBERBLICK



8.1 DAS CONNECT-FRAMEWORK

WHAT IT IS AND WHY WE NEED IT



FUNKTIONALITÄTEN EINER ANWENDUNG

- Kernfunktionalitäten: primäre Aufgaben zur Umsetzung einer domänenspezifischen Geschäftslogik
- Infrastrukturaufgaben: sekundäre, anwendungsunabhängige Managementaufgaben

MIDDLEWARE

- Verschiedene Middleware-Funktionen zur Wahrnehmung unterschiedlicher Aufgaben
- Beispiele: Logging, Authentifizierung, Sitzungsmanagement, Internationalisierung
- Sequentielle Ausführung von Funktionen bevor eine Anfrage die Anwendung erreicht

CONNECT-FRAMEWORK

- Middleware für Node.js
- Ziel: komponentenorientierte Entwicklung und Integration von Infrastrukturcode
- Zwei Arten von Middleware-Funktionen
 - Filter: werten eingehende Anfrage (ohne Veränderung) aus und leiten sie an das nächste Modul weiter
 - Provider: fungieren als Stellvertreter der Webanwendung und leiten Anfrage nach Bearbeitung nicht weiter

8.1 DAS CONNECT-FRAMEWORK

STRUKTUR EINER CONNECT-ANWENDUNG

```
1 var connect = require('connect');
2 var app = connect();
3 app.listen(8080);
4 console.log('Server läuft unter http://localhost:8080');

1 var connect = require('connect');
2 var app = connect();
3
4 var helloWorld = function(req, res, next) {
5   res.setHeader('Content-Type', 'text/plain');
6   res.end('Hello World');
7 }
8
9 app.use(helloWorld);
10
11 app.listen(8080);
```

GRUNDSTRUKTUR EINER CONNECT-ANWENDUNG

- Import des Connect-Moduls
- Aufruf der **connect**-Methode liefert einen Server, der die jeweilige Anwendung repräsentiert
- Ersetzt die bisherige **createServer**-Methode aus dem **http**-Modul

MIDDLEWARE-FUNKTIONEN

- Middleware-Funktionen sind Callback-Funktionen mit der Signatur **function(req, res, next)**
 - **req**: Objekt der HTTP-Anfrage
 - **res**: Objekt der HTTP-Antwort
 - **next**: nächste auszuführende Middleware-Funktion
- Hinzufügen der Middleware-Funktion mit **use**-Methode

8.1 DAS CONNECT-FRAMEWORK

SEQUENTIALISIERUNG UND STRUKTUR EINER FUNKTION

```
1 var connect = require('connect');
2 var app = connect();
3
4 var logger = function(req, res, next) {
5   console.log(req.method, req.url);
6   next();
7 }
8
9 var helloWorld = function(req, res, next) {
10   res.setHeader('Content-Type', 'text/plain');
11   res.end('Hello World');
12 }
13
14 app.use(logger);
15 app.use(helloworld);
16
17 app.listen(8080);
```



SEQUENTIALISIERUNG VON MIDDLEWARE-FUNKTIONEN

- Sequentialisierung durch wiederholte Ausführung der `use`-Methode
- Achtung: First-in-First-Out, d.h. Callbacks werden bei eingehenden Anfragen in der Reihenfolge abgearbeitet, in der sie mit `use` hinzugefügt wurden
- Weitergabe einer Anfrage zwischen Middleware-Funktionen durch Aufruf von `next` (ohne Übergabe von `req` und `res`)

STRUKTUR EINER MIDDLEWARE-FUNKTION

- Anweisungen vor `next` werden bei der "Durchleitung" der Anfrage bearbeitet
- Anweisungen nach `next` werden beim Zurückspielen der Antwort bearbeitet, d.h. nach Bearbeitung der Anfrage durch die Webanwendung

```
var middlewareFunction = function(req, res, next) {
  console.log('Ausführung bei Weiterleitung der Anfrage');
  next();
  console.log('Ausführung bei Zurückleitung der Antwort');
};
```

...

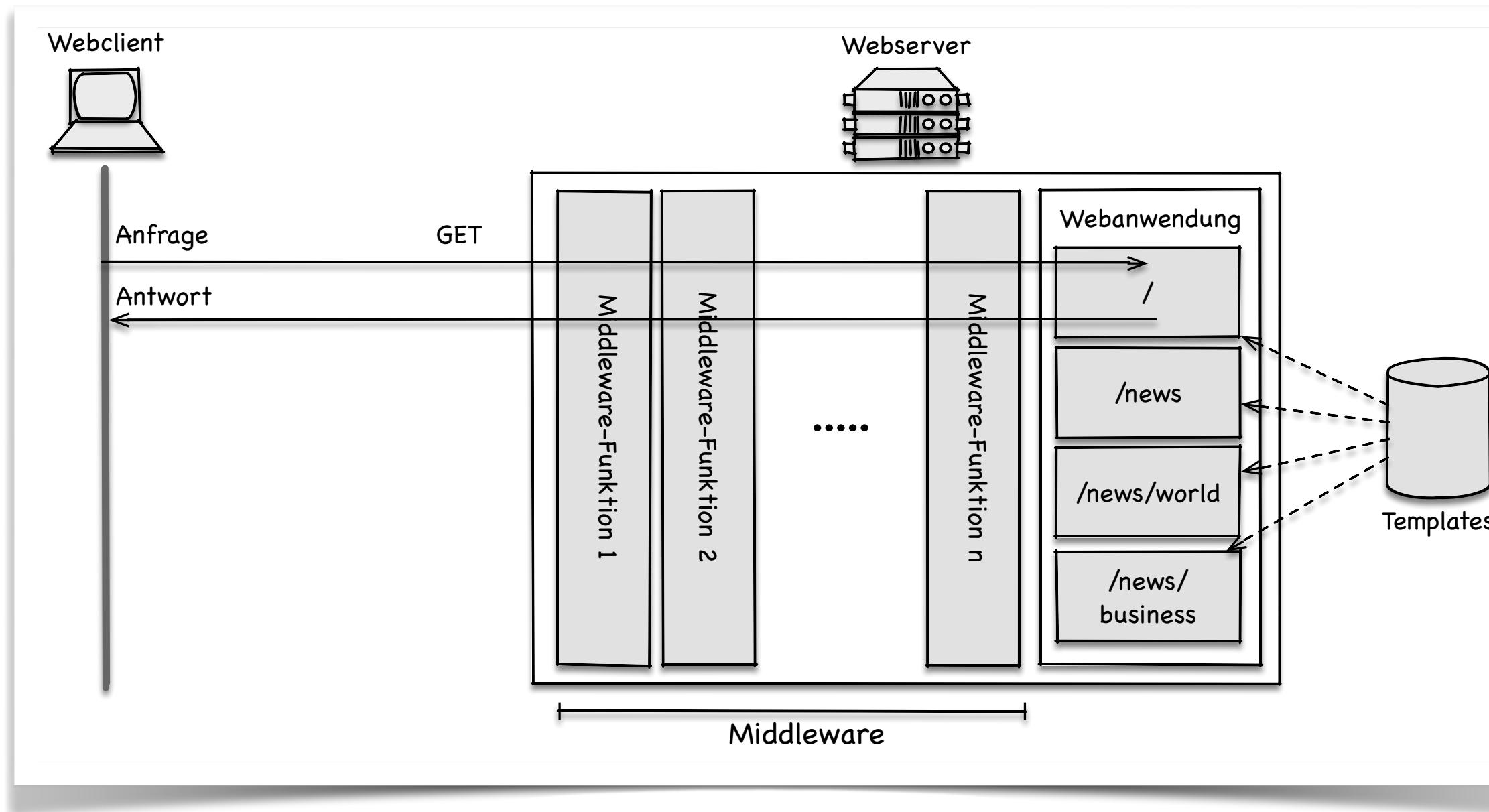
8.1 DAS CONNECT-FRAMEWORK MOUNTING

```
1 var connect = require('connect');
2 var app = connect();
3
4 var logger = function(req, res, next) {
5   console.log(req.method, req.url);
6   next();
7 }
8
9 var helloWorld = function(req, res, next) {
10   res.setHeader('Content-Type', 'text/plain');
11   res.end('Hello World');
12 }
13
14 var goodbyeWorld = function(req, res, next) {
15   res.setHeader('Content-Type', 'text/plain');
16   res.end('Goodbye World');
17 }
18
19 app.use(logger);
20 app.use('/hello', helloWorld);
21 app.use('/goodbye', goodbyeWorld);
22
23 app.listen(8080);
```



- Vorherige Beispiele: Middleware-Funktionen werden bei jedem HTTP-Aufruf ausgeführt
- Mounting: Middleware-Funktion wird in ein bestimmtes Verzeichnis des Webservers "eingehängt"
- Ausführung der "gemounteten" Middleware-Funktion nur dann, wenn die aufgerufene Ressource unterhalb des Verzeichnisses liegt, in dem die Middleware-Funktion eingehängt wurde
- Bedingung zur Ausführung der Middleware-Funktion: Mount-Verzeichnis muss ein Präfix des Pfadnamens der aufgerufenen URL sein
- Beispiel: Zunächst Ausführung der Funktion **logger** bei jeder Anfrage, dann Ausführung von **helloWorld** und **goodbyeWorld** wenn Ziel der Anfrage im Verzeichnis **hello** bzw. **goodbye** liegt bzw. deren Unterverzeichnissen
- Beispiel: Ausführung der Middleware-Funktion **goodbyeWorld** bei Aufruf von **xyz.de/goodbye**, **xyz.de/goodbye/alex**, **xyz.de/goodbye/berta**, **xyz.de/goodbye/visitors/charly**, usw.

8.2 DAS EXPRESS-FRAMEWORK ERWEITERUNG VON CONNECT



EXPRESS-FRAMEWORK

- Connect: Einbindung von Middleware-Funktionen für Infrastrukturaufgaben
- Express: Erweiterung von Connect um Mechanismen zur Erledigung von Infrastrukturaufgaben
- Express kapselt Connect, d.h. Objekte und Methoden von Connect stehen auch in Express zur Verfügung

ROUTE PATH

- Endpunkt einer Webanwendung (Pfadnamen einer URI)

ROUTE METHOD

- HTTP-Methode für die eine Route definiert ist

ROUTING

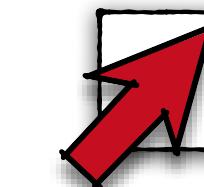
- Bearbeitung einer HTTP-Anfrage und Erstellung der Antwort
- genauer: Durchleiten einer Anfrage durch Middleware-Funktionen und verschiedene Module der Webanwendung und Zurückleitung der Antwort

ROUTE HANDLER

- Callback-Funktion einer Middleware-Funktion bzw. eines Moduls der Webanwendung

TEMPLATE ENGINE

- Erstellung von dynamischen HTML-Ausgaben basierend auf Templates



<http://expressjs.com/en/4x/api.html>

8.2 DAS EXPRESS-FRAMEWORK

INTEGRIERTE MODULE (I)

ANFRAGEN UND ANTWORTEN

Modul	Beschreibung
basicAuth	<ul style="list-style-type: none">Fügt einer Anwendung Unterstützung für die http-Authentifizierung hinzuAuthentifizierung erfolgt über einen Callback, so dass beliebige Strategien implementiert werden können
compress	<ul style="list-style-type: none">Komprimierung von ausgehenden Antworten auf Basis von gzip und deflateModul prüft eigenständig, ob der vom Benutzer verwendete Webbrowser komprimierte Daten akzeptiert
csrf	<ul style="list-style-type: none">Schützt Webanwendungen vor Cross Site Request ForgeryGeneriert für jeden Benutzer ein Token, welches bei jeder Anfrage validiert wirdBenötigt die Module cookieParser und session
limit	<ul style="list-style-type: none">Begrenzt das vom Webserver akzeptierte Datenvolumen von eingehenden AnfragenÜberschreitet die Größe einer Anfrage das zulässige Datenvolumen, wird diese abgebrochen
methodOverride	<ul style="list-style-type: none">Fügt einer Anwendung Unterstützung für unechte http-Methoden hinzu
responseTime	<ul style="list-style-type: none">Berechnet die Zeit, die zur Beantwortung einer eingehenden Anfrage benötigt wird, und sendet diese als Header in der ausgehenden Antwort an den Webbrowser zurück

PARSER

Modul	Beschreibung
bodyParser	<ul style="list-style-type: none">Verarbeitet den Body von eingehenden AnfragenUnterstützt die Formate json, urlencoded und multipart
cookieParser	<ul style="list-style-type: none">Analysiert die in einer eingehenden Anfrage enthaltenen CookiesStellt Cookies im Objekt req.cookies zur Verfügung
json	<ul style="list-style-type: none">Verarbeitet den Body von eingehenden AnfragenUnterstützt das Format jsonWird intern vom bodyParser-Modul verwendet
multipart	<ul style="list-style-type: none">Verarbeitet den Körper von eingehenden AnfragenUnterstützt das Format multipart/form-dataWird intern vom bodyParser-Modul verwendet
urlencoded	<ul style="list-style-type: none">Verarbeitet den Körper von eingehenden AnfragenUnterstützt das Format application/x-www-form-urlencodedWird intern vom bodyParser-Modul verwendet

8.2 DAS EXPRESS-FRAMEWORK INTEGRIERTE MODULE (II)

WEBSERVER UND SESSIONS

Modul	Beschreibung
cookieSession	<ul style="list-style-type: none">• Erstellt Sitzungen (Sessions) auf Basis von Cookies• Gesamte Session wird in ein Cookie serialisiert und von dort wieder serialisiert
directory	<ul style="list-style-type: none">• Formatierte Ausgabe einer Dateiliste für ein angegebenes Verzeichnis• Verwendung von Filtern möglich• Möglichkeit der Ausgabe versteckter Dateien
favicon	<ul style="list-style-type: none">• Sendet die Datei favicon• favicon: Kofferwort von favorite und icon (d.h. Favoritensymbol)• Verwendung eines Caches, so dass die Datei nicht für jede Anfrage aus dem Dateisystem gelesen werden muss
session	<ul style="list-style-type: none">• Fügt einer Webanwendung Unterstützung für Sessions hinzu• Sessions werden durch SessionID repräsentiert, die durch Cookies transportiert werden• Zustand einer Session wird im Arbeitsspeicher gehalten• Alternative Einbindung persistenter Speicherlösungen möglich• Erfordert cookieParser-Modul
static	<ul style="list-style-type: none">• Senden von statischen Dateien aus dem Dateisystem• Kein Caching
staticCache	<ul style="list-style-type: none">• Cache für das static-Modul• Umsetzung einer Least-Recently-Used-Strategie• Begrenzung der Anzahl und der Größe der gecacheten Dateien

WERKZEUGE

Modul	Beschreibung
errorHandler	<ul style="list-style-type: none">• Vorgefertigtes Fehler-Modul• Liefert im Falle eines Fehlers die entsprechende Fehlermeldung als Antwort an den Webbrowser
logger	<ul style="list-style-type: none">• Protokolliert alle eingehenden Anfragen in einem übergebenen Stream

8.2 DAS EXPRESS-FRAMEWORK

WICHTIGE METHODEN UND OBJEKTE IM ÜBERBLICK

METHODEN UND OBJEKTE DES APPLICATION-OBJEKTES

Methoden	Beschreibung
app.set(name, value)	Setzt eine Umgebungsvariable zur Konfiguration von Express
app.get(name)	Liest den Wert einer Umgebungsvariablen
app.use([path], callback)	Fügt eine Middleware-Funktion in Form eines Callbacks in die Route der Webanwendung ein und bindet sie optional in ein bestimmtes Verzeichnis
app.VERB(path, [callback...], callback)	Fügt eine oder mehrere Module der Webanwendung hinzu und bindet sie an ein bestimmtes Verzeichnis und eine bestimmte HTTP-Methode

METHODEN UND OBJEKTE DES REQUEST-OBJEKTES

Methoden und Objekte	Beschreibung
req.query	Enthält die Query-String-Parameter einer aufgerufenen URL
req.params	Enthält die Routing-Parameter einer URL
req.body	Enthält den Body-Teil einer Anfrage, unterstützt durch bodyParser
req.param(name)	Liefert den Wert eines bestimmten Anfrageparameters
req.path, .host und .ip	Liefern Pfad, Hostname und IP-Adresse
req.cookies	Liefern Cookies einer Anfrage, bestimmt durch das cookieParser-Modul

mit Änderungen übernommen von <http://expressjs.com/>

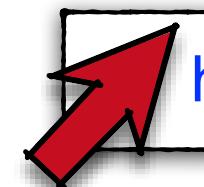
METHODEN UND OBJEKTE DES RESPONSE-OBJEKTES

Methoden und Objekte	Beschreibung
res.status(code)	Setzt den Status-Code einer HTTP-Antwort
res.set(field, [value])	Setzt Parameter des Antwort-Headers
res.cookie(name, value, [options])	Setzt Antwort-Cookie
res.redirect([status], url)	Anweisung eines Redirects zur der angegebenen URL
res.send([body] status, [body])	Zusammenstellung der HTTP-Antwort und automatische Definition wichtiger Header-Parameter, z.B. Content-Type und Content-Length
res.render(view, [locals], callback)	Zusammenstellung einer HTML-Seite basierend auf einem Template

8.2 DAS EXPRESS-FRAMEWORK ROUTE HANDLERS

```
1 var express = require('express');
2 var app = express();
3
4 app.get('/', function (req, res) {
5   res.send('GET request to the homepage');
6 });
7
8 app.post('/', function (req, res) {
9   res.send('POST request to the homepage');
10});
...
11 app.all('/secret', function (req, res, next) {
12   console.log('Accessing the secret section ...');
13   next();
14 });
...
15
```

- **app.VERB** mit **VERB=get, post, put usw.** übergibt Route Handler (Callback-Funktion) für die Beantwortung von HTTP-Anfragen einer bestimmten Methode an einen bestimmten Pfad
- **app.all** bindet alle Methoden an einen Route Handler, d.h. Ausführung geschieht unabhängig von der verwendeten HTTP Methode



<http://expressjs.com/en/4x/api.html>

8.2 DAS EXPRESS-FRAMEWORK

ROUTE PATHS (I)

ZEICHENKETTEN

Anfragen an Root: /

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

Anfragen an /about

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

Anfragen an /random.text

```
app.get('/random.text', function (req, res) {  
  res.send('random.text');  
});
```

- Pfade werden intern als reguläre Ausdrücke verarbeitet
- Verschiedene Möglichkeiten der Definition von Pfaden
 - Zeichenketten
 - Muster von Zeichenketten
 - Reguläre Ausdrücke
 - Parameter

MUSTER VON ZEICHENKETTEN

Anfragen an /acd und /abcd

```
app.get('/ab?cd', function(req, res) {  
  res.send('ab?cd');  
});
```

Anfragen an /abcd, /abbcd,
/abbbcd, usw.

```
app.get('/ab+cd', function(req, res) {  
  res.send('ab+cd');  
});
```

Anfragen an /abcd, /abxcd,
/abDQ0PJ34cd, usw.

```
app.get('/ab*cd', function(req, res) {  
  res.send('ab*cd');  
});
```

Anfragen an /abe, /abcde

```
app.get('/ab(cd)?e', function(req, res) {  
  res.send('ab(cd)?e');  
});
```

8.2 DAS EXPRESS-FRAMEWORK

ROUTE PATHS (II)

Anfragen mit einem **a** im Pfadnamen

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

Anfragen z.B. an **butterfly** und **dragonfly**, aber nicht **butterflyman**

```
app.get('/*fly$', function(req, res) {  
  res.send('/*fly$');  
});
```

REGULÄRE AUSDRÜCKE

PARAMETER

```
app.get('/foo/:id', function(req, res) {  
  res.send('/foo'+req.params.id);  
});
```

Anfragen an **/foo/23** usw.

```
app.get('/foo/:id/:subid', function(req, res) {  
  res.send('/foo'+req.params.id+'/'+req.params.subid);  
});
```

Anfragen an **/foo/23/5** usw.

8.2 DAS EXPRESS-FRAMEWORK

REGISTRIERUNG EINES ROUTE HANDLERS

Ein Route Handler

```
app.get('/example/a', function (req, res) {
  res.send('Hello from A!');
});
```

Mehrere Route Handler

```
app.get('/example/b', function (req, res, next) {
  console.log('response will be sent by next function');
  next();
}, function (req, res) {
  res.send('Hello from B!');
});
```

Mehrere Route Handler als Array

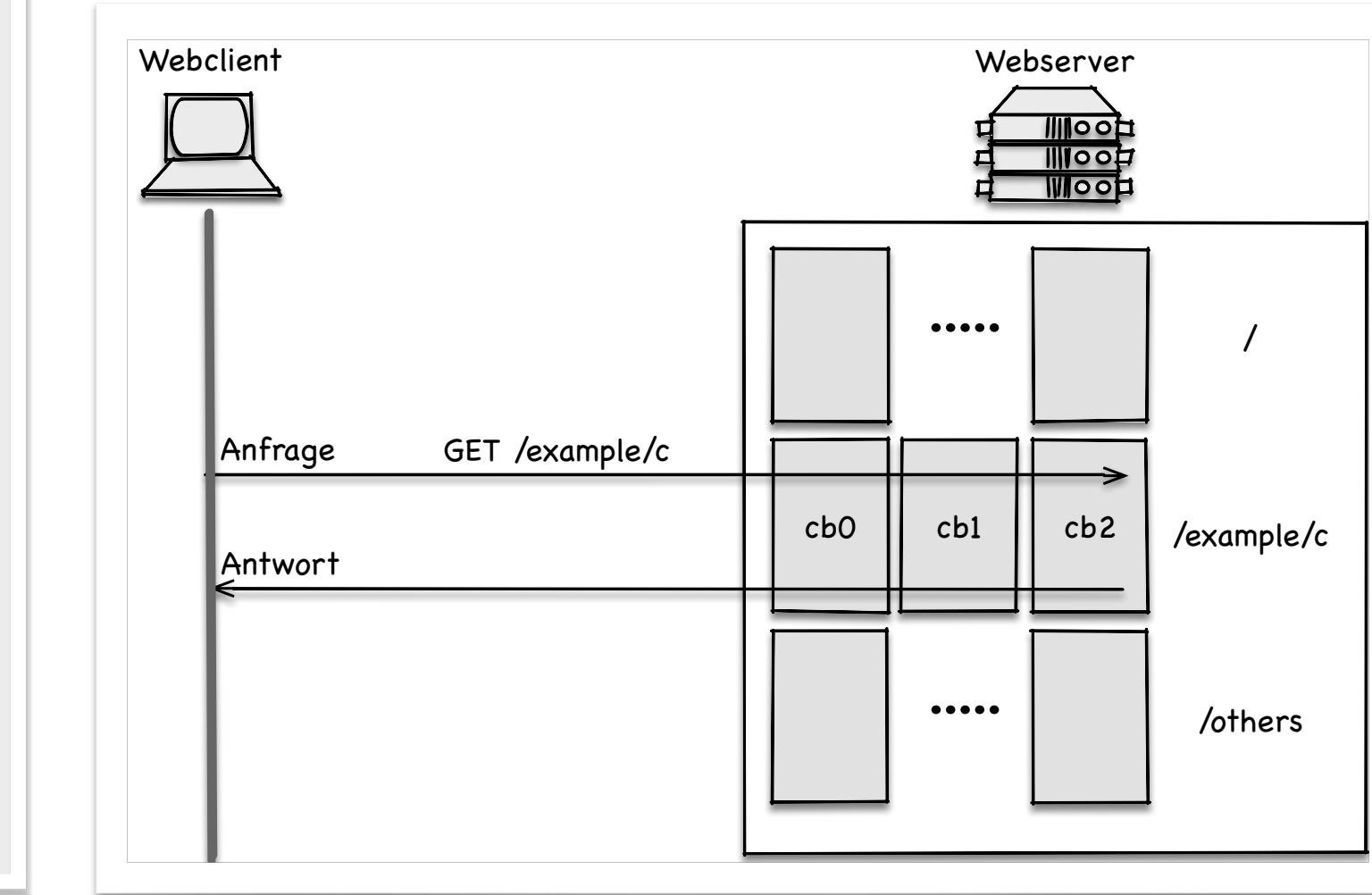
```
var cb0 = function (req, res, next) {
  console.log('CB0');
  next();
}

var cb1 = function (req, res, next) {
  console.log('CB1');
  next();
}

var cb2 = function (req, res) {
  res.send('Hello from C!');
}

app.get('/example/c', [cb0, cb1, cb2]);
```

- **app.VERB** können beliebig viele Route Handler (Callback-Funktionen) übergeben werden
- Zwei Möglichkeiten
 - Aneinanderreihung von Funktions-Übergabeparametern
 - Übergabe als Array
- Beachte: Aufruf der nächsten Funktion per **next()** bis Ende der Route erreicht



8.2 DAS EXPRESS-FRAMEWORK

KONFIGURATION VON ROUTE HANDLERS MIT HILFE VON ROUTER

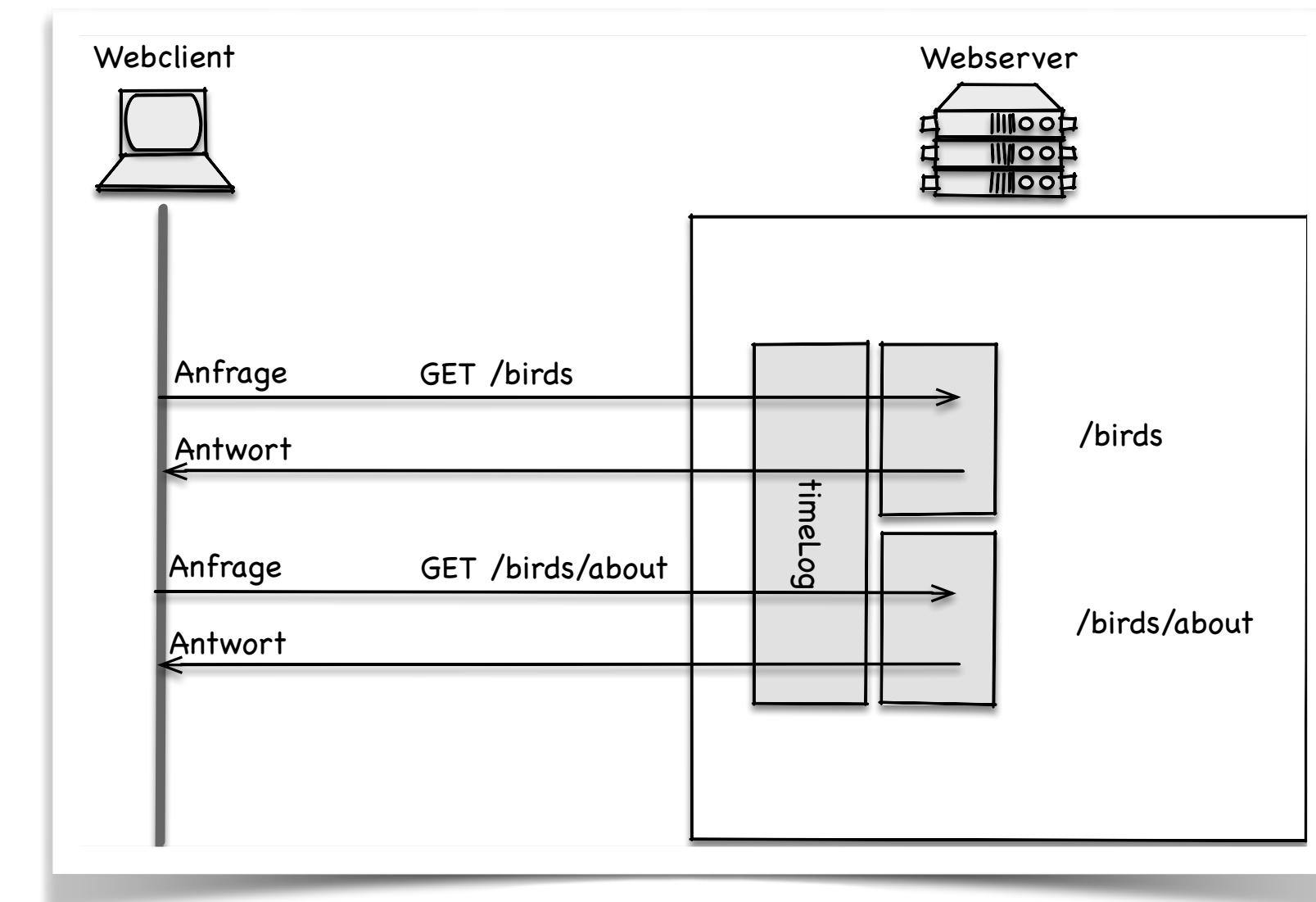
```
1 var express = require('express');
2 var router = express.Router();
3
4 router.use(function timeLog(req, res, next) {
5   console.log('Time: ', Date.now());
6   next();
7 });
8
9 router.get('/', function(req, res) {
10   res.send('Birds home page');
11 });
12
13 router.get('/about', function(req, res) {
14   res.send('About birds');
15 });
16
17 module.exports = router;
```

Zusammenstellung einer
Middleware mittels Router

```
var birds = require('./birds');
...
app.use('/birds', birds);
```

Verknüpfung mit einer
Anwendung unter dem
Verzeichnis /birds

- **express.Router** ermöglicht die Zusammenstellung einer Middleware aus mehreren Route Handlers zur späteren Verknüpfung mit einer Anwendung unter einem bestimmten Pfad



8.2 DAS EXPRESS-FRAMEWORK

HTML TEMPLATES UND TEMPLATE ENGINES

Jade

```
html
  head
    title!= title
  body
    h1!= message
```

EJS

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
  </head>
  <body>
    <h1><%= message %></h1>
  </body>
</html>
```

Verwendung von Templates

```
...
app.set('view engine', 'ejs');
app.set('views', '/app/views/');
...
app.get('/', function (req, res) {
  res.render('index', { title: 'Hey', message: 'Hello there!'});
});
```

- *Template Engines* unterstützen die Generierung dynamischer HTML-Seiten mit Hilfe vorgefertigter *Templates* (Vorlagen)
- Templates enthalten statischen HTML-Code sowie Parameter, die vor der Auslieferung mit Werten belegt werden (vergleiche PHP oder Java Server Pages)
- Unterstützte Template-Formate bzw. Template Engines
 - Jade
 - EJS
 - JsHtml
- Einbindung einer Template Engine in eine Webanwendung durch Umgebungsvariable **view engine**

```
app.set('view engine', 'jade');
app.set('view engine', 'ejs');
```

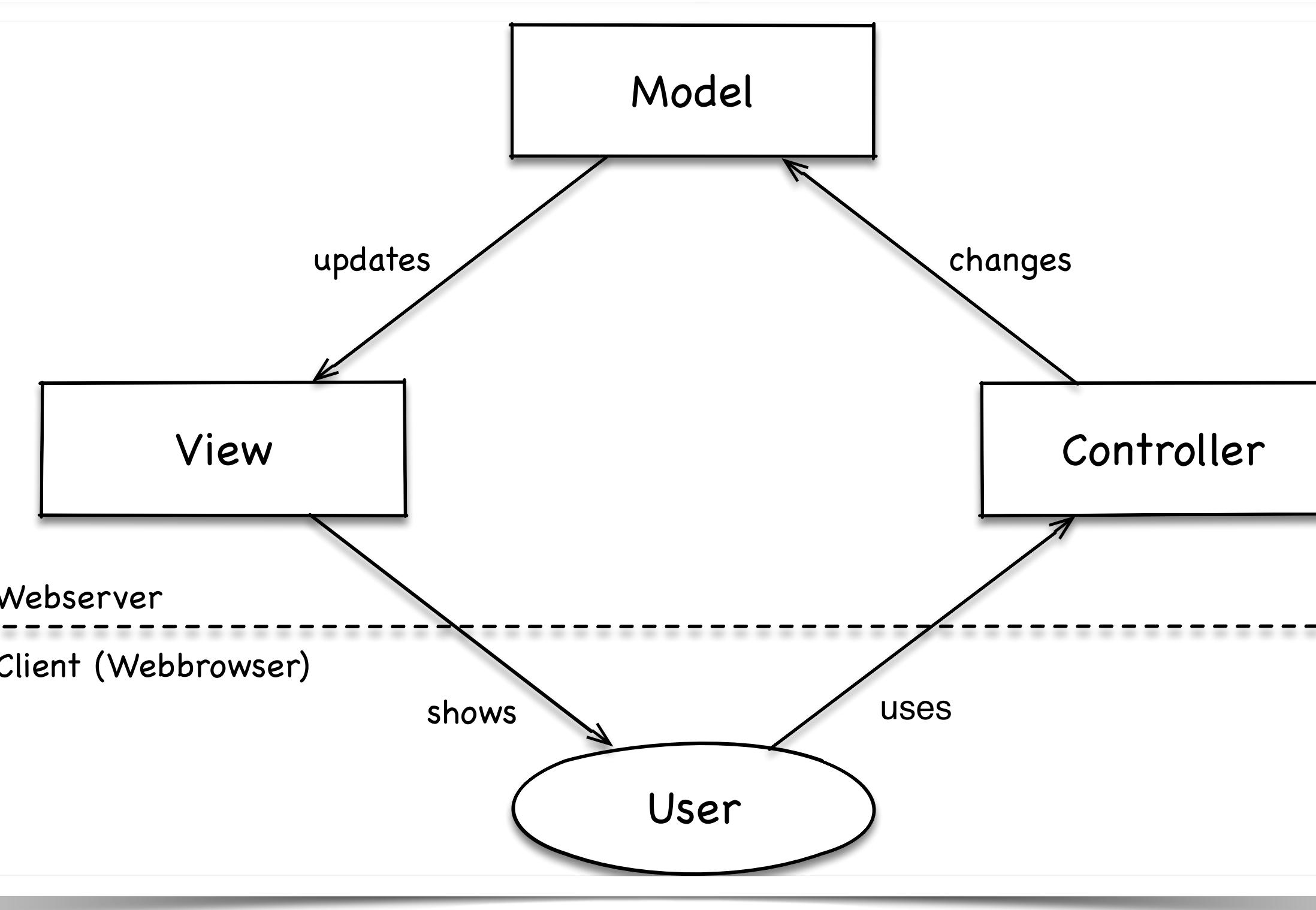
- Übergabe des Template-Verzeichnisses durch Umgebungsvariable **views**

```
app.set('views', '/app/views/');
```



<http://expressjs.com/en/4x/api.html>

8.3 MODEL-VIEW-CONTROLLER WHAT IT IS AND WHY WE NEED IT

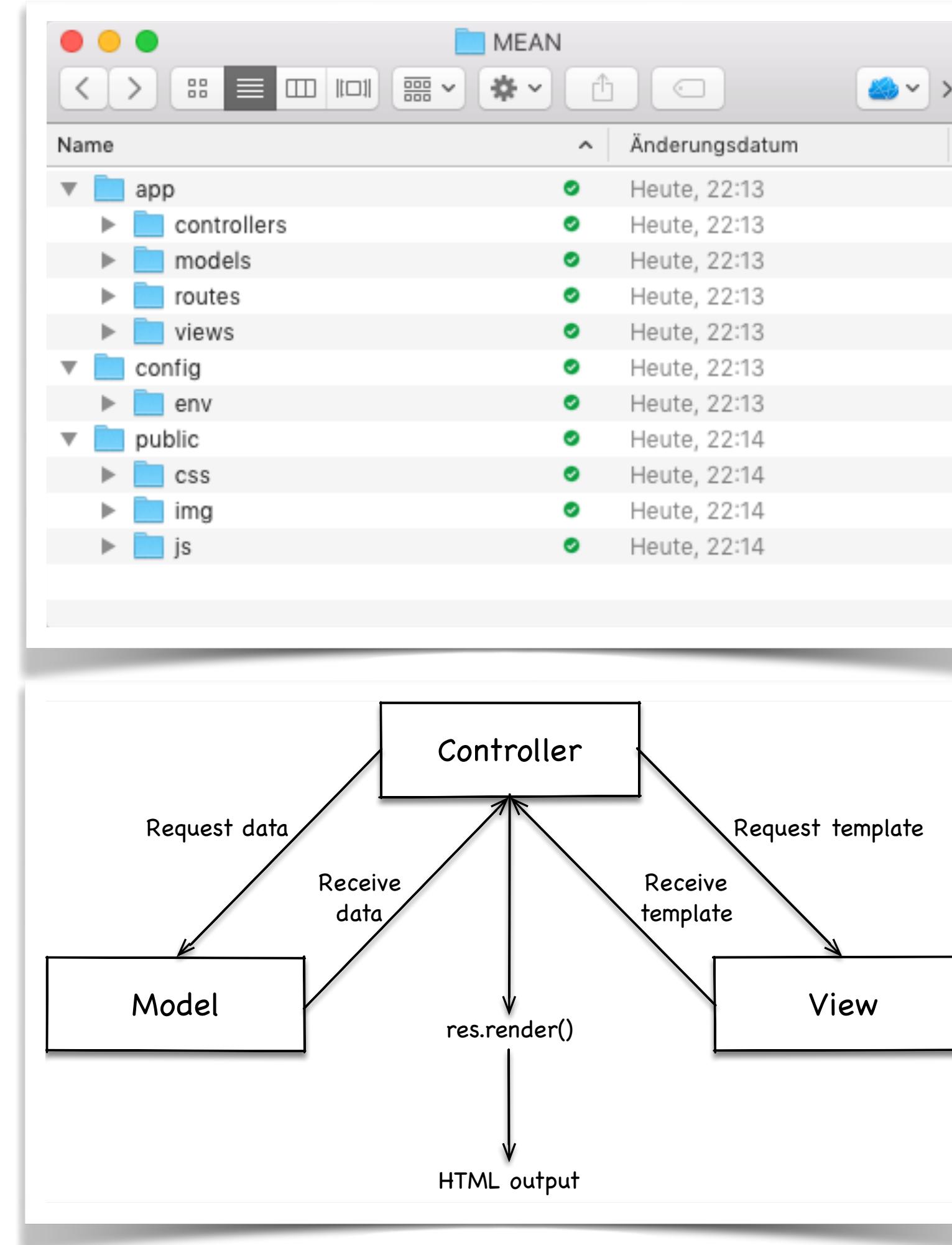


- Problem: Komplexität moderner Webanwendungen
- Unterschiedliche Aufgaben
 - Anwendungslogik und Verarbeitung von Daten
 - Persistente Datenspeicherung
 - Präsentation und Darstellung
 - Kommunikation

MVC PATTERN

- MVC: Model-View-Controller
- Funktionale Aufteilung einer Webanwendung in getrennte Module
- View: Nutzerinterface, d.h. Darstellung einer Webanwendung und Interaktion mit dem Nutzer
- Model: Datenstrukturen einer Webanwendung und assoziierte Aufgaben, d.h. Lesen und Schreiben von Daten sowie Persistenz
- Controller: Bearbeitung von Anfragen, Sitzungsmanagement sowie Verknüpfung von Model und View

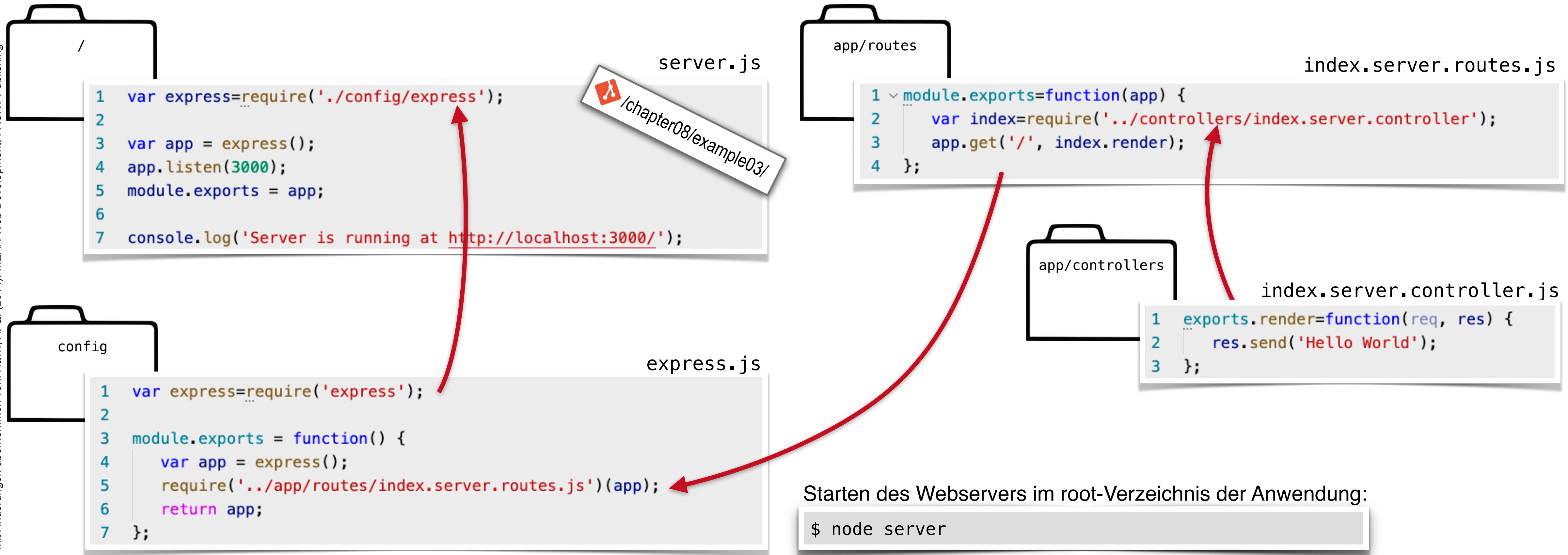
8.3 MODEL-VIEW-CONTROLLER UMSETZUNG VON MVC MIT NODE.JS UND EXPRESS



VERZEICHNISSE

- **app**: Express-Anwendungslogik
 - **controllers**: Controllers der Webanwendung
 - **models**: Models der Webanwendung
 - **routes**: Routing-Middleware der Webanwendung
 - **views**: Templates der Webanwendung
- **config**: Konfigurationsdateien der Webanwendung für unterschiedliche Umgebungen (Development, Testing, Production)
- Hier: Fokussierung auf production
- **public**: Aufbewahrung der statischen Dateien zur Auslieferung an den Webbrowser
 - **css**: Style-Sheet-Dateien
 - **img**: Bilddateien
 - **js**: Client-seitige JavaScripts

8.3 MODEL-VIEW-CONTROLLER EINFACHES BEISPIEL



8.3 MODEL-VIEW-CONTROLLER

BEISPIEL: DYNAMISCHE WEBSEITEN (I)

```
server.js
1 var express = require('../config/express');
2
3 var app = express();
```

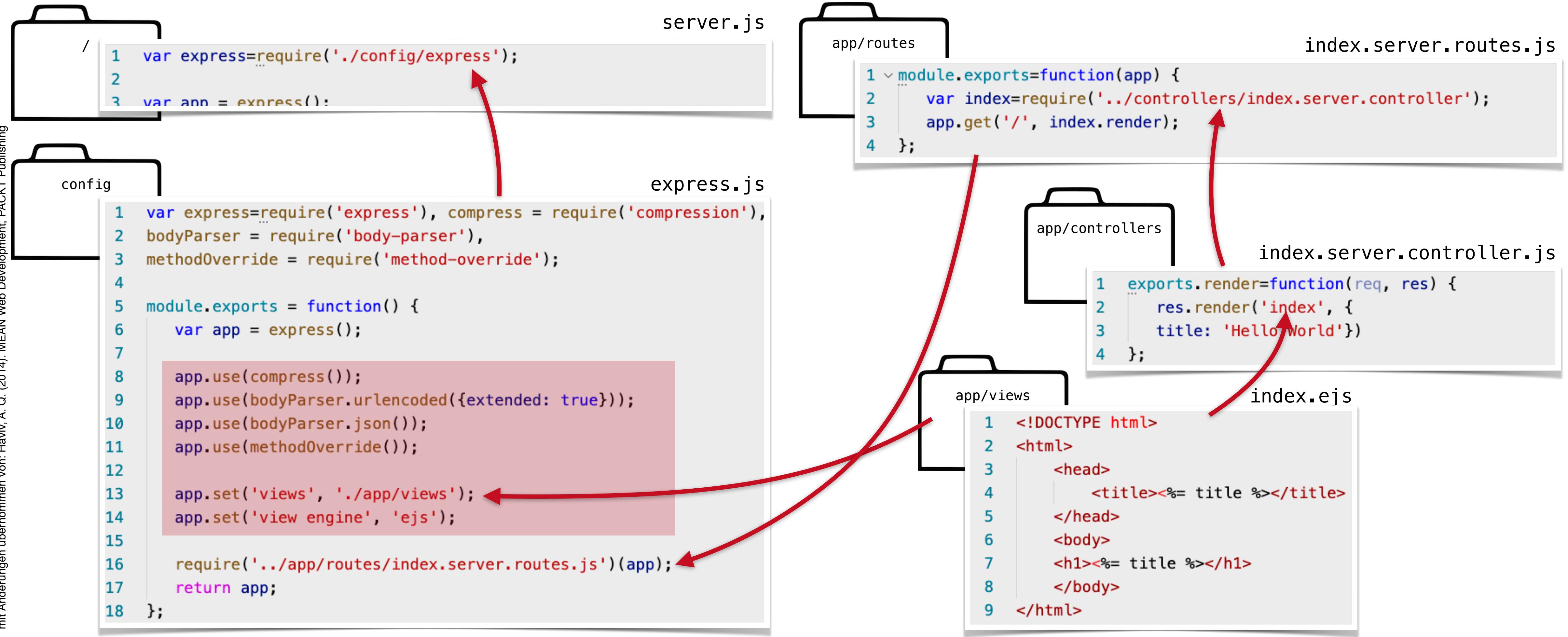
```
config
1 var express = require('express'), compress = require('compression'),
2 bodyParser = require('body-parser'),
3 methodOverride = require('method-override');
4
5 module.exports = function() {
6   var app = express();
7
8   app.use(compress());
9   app.use(bodyParser.urlencoded({extended: true}));
10  app.use(bodyParser.json());
11  app.use(methodOverride());
12
13  app.set('views', './app/views');
14  app.set('view engine', 'ejs');
15
16  require('../app/routes/index.server.routes')(app);
17  return app;
18};
```

ERWEITERUNG DER KONFIGURATION DER WEBAWENDUNG

- Einbindung diverser Middleware-Funktionen zur Erledigung von Infrastrukturaufgaben
- Konfiguration der Template Engine, d.h. Verwendung von EJS und setzen des Verzeichnisses, welches die Templates enthält

8.3 MODEL-VIEW-CONTROLLER

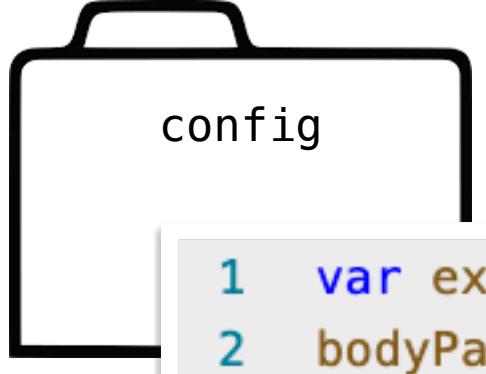
BEISPIEL: DYNAMISCHE WEBSEITEN (II)



8.3 MODEL-VIEW-CONTROLLER

BEISPIEL: AUSLIEFERUNG STATISCHER DATEIEN

mit Änderungen übernommen von: Haviv, A. Q. (2014). MEAN Web Development, PACKT Publishing

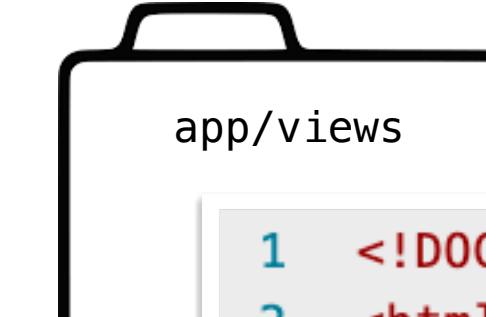


config

```
1 var express = require('express'), compress = require('compression'),
2 bodyParser = require('body-parser'),
3 methodOverride = require('method-override');
4
5 module.exports = function() {
6   var app = express();
7
8   app.use(compress());
9   app.use(bodyParser.urlencoded({extended: true}));
10  app.use(bodyParser.json());
11  app.use(methodOverride());
12
13  app.set('views', './app/views');
14  app.set('view engine', 'ejs');
15
16  require('../app/routes/index.server.routes.js')(app);
17
18  app.use(express.static('./public'));
19
20  return app;
21};
```

express.js

- Problem: Auslieferung statischer Dateien mit Node.js muss bisher von Hand programmiert werden
- express.static-Middleware ermöglicht die automatische Auslieferung statischer Dateien
- Beachte: Hinzufügen von **static** zur Webanwendung sollte nach Definition der Routen erfolgen
- Andernfalls: alle HTTP-Anfragen müssten **static** durchlaufen und verursachen Verzögerungen beim Prüfen des Verzeichnisses für statische Dateien



app/views

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title><%= title %></title>
5   </head>
6   <body>
7     <img scr="img/logo.png" alt="Logo">
8     <h1><%= title %></h1>
9   </body>
10 </html>
```

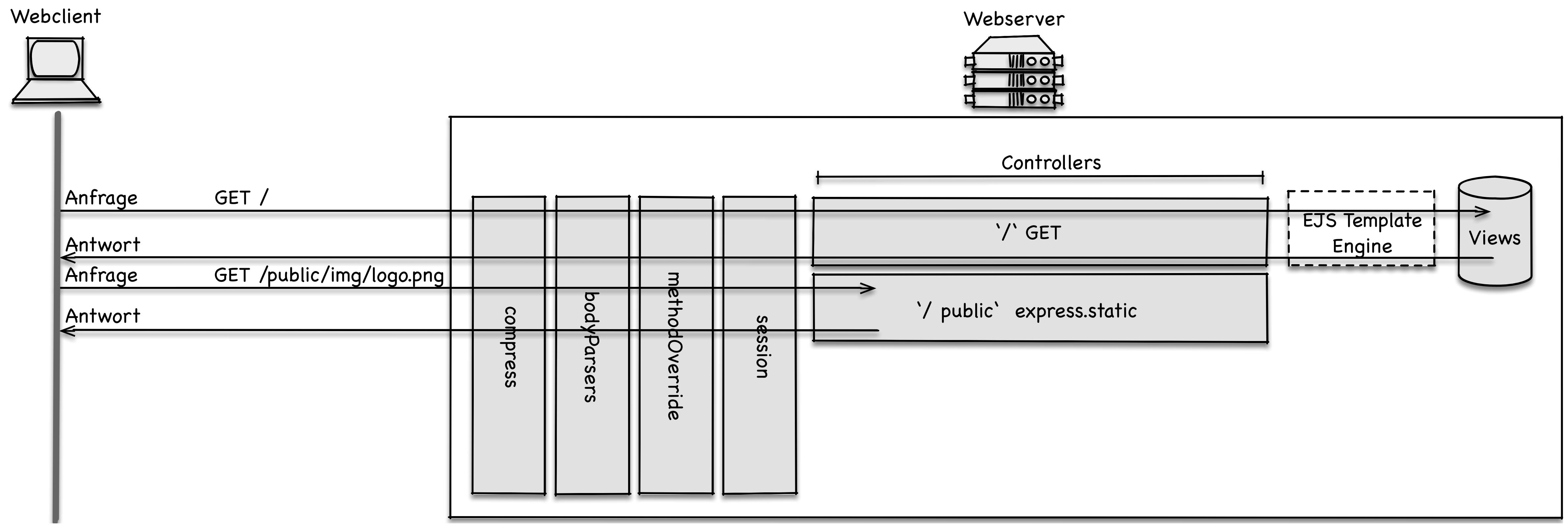
index.ejs

Einfache Auslieferung statischer Dateien in Node.js, z.B. von Bildern



<http://expressjs.com/en/4x/api.html>

8.3 MODEL-VIEW-CONTROLLER ZUSAMMENFASSUNG



8.4 SITZUNGSMANAGEMENT

ARTEN DES SITZUNGSMANAGEMENTS

- Sitzung: Verbindung zwischen Client und Server, die mehrere Anfragen umfasst
- Sitzungen werden durch Session Identifiers (SessionIds) repräsentiert
- Problem: HTTP ist zustandslos, d.h. Zustand wird nach Bearbeitung einer HTTP-Anfrage gelöscht
- Sitzungsmanagement: Verwaltung, Speicherung und Wiederherstellen des Zustandes einer Sitzung im Backend
- Drei Optionen für das Sitzungsmanagement
 - Client-basiertes Sitzungsmanagement
 - In-Memory-Sitzungsmanagement
 - Datenbank-basiertes Sitzungsmanagement

CLIENT-BASIERTES SITZUNGSMANAGEMENT

- Sitzungszustand wird in einem oder mehreren Cookies speichert - keine SessionId
- Übertragung des Cookies bei jeder Anfrage an den Server
- Aktualisierung des Cookies mit neuem Sitzungszustand bei jeder Antwort vom Server an den Client

IN-MEMORY-SITZUNGSMANAGEMENT

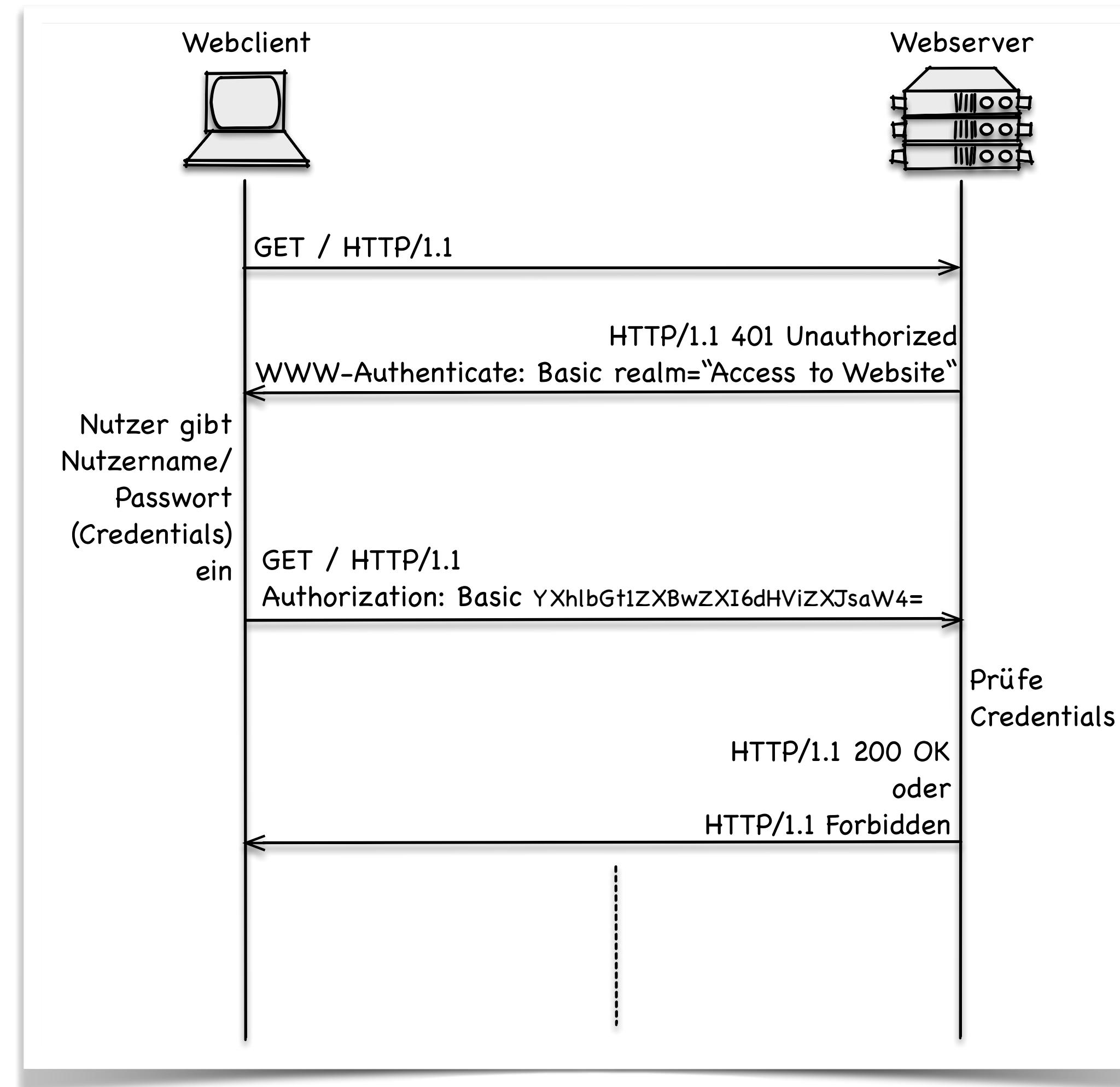
- Sitzungszustand wird im Hauptspeicher des Servers gespeichert
- Cookies transportieren die SessionId
- Schnell, aber schlecht skalierbar da kein Load Balancing

DATENBANK-BASIERTES SITZUNGSMANAGEMENT

- Sitzungszustand wird in einem externen Session Store gespeichert
- Cookies transportieren die SessionId
- Beispiel: MongoDB

8.4 SITZUNGSMANAGEMENT

BASIC AUTHENTICATION

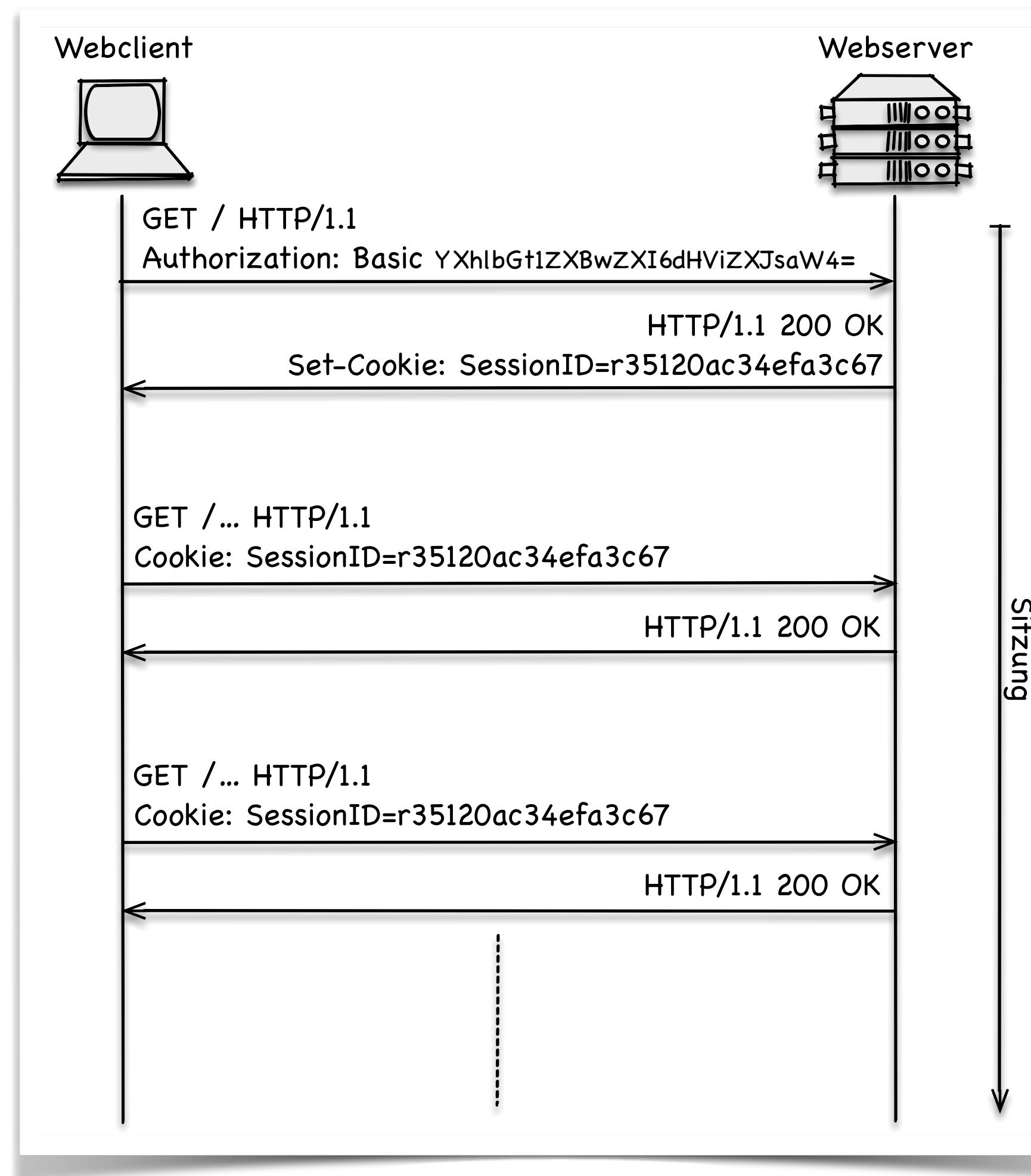


- Zugang zu geschützten Bereichen eines Webservers mittels Nutzername und Passwort zur Authentifizierung des Nutzers
- Verschiedene Möglichkeiten der Authentifizierung: Basic Authentication, Digest Access Authentication,...

BASIC AUTHENTICATION

- Webserver zeigt dem Client die Notwendigkeit einer Authentifizierung mittels Status Code **403** sowie die Authentifizierungsart und den geschützten Bereich (Realm) an
- Nutzername/Password und Art der Authentifizierung gelten innerhalb eines Realm
- Nutzername und Passwort werden vom Client an den Server Base64-codiert übertragen
- Base64: Codierung von Binärdaten durch die Zeichen **A-Z**, **a-z**, **0-9**, **+** und **/** sowie **=** am Ende
- Achtung: base64 stellt keine Vertraulichkeit her
- **Basic Authentication nur mit HTTPS verwenden**

8.4 SITZUNGSMANAGEMENT SESSION-IDS



IN-MEMORY SITZUNGSMANAGEMENT MIT EXPRESS-SESSION

- Express-Session-Middleware ermöglicht Sicherung des Sitzungszustands im Hauptspeicher des Servers
- Middleware erzeugt ein Objekt **session** innerhalb des **req**-Objektes
- **session**-Objekt kann genutzt werden, um Parameter des Sitzungszustandes (z.B. Bestückung eines Einkaufswagens im eCommerce) zu sichern
- Middleware erkennt wiederkehrenden Nutzer und fügt das nutzerspezifische **session**-Objekt dem **req**-Objekt hinzu
- Middleware verlangt einen Schlüssel zum Signieren der im Cookie transportierten SessionId (strittig und fragwürdig!!!)



<http://expressjs.com/en/4x/api.html>

8.4 SITZUNGSMANAGEMENT

SITZUNGSMANAGEMENT MIT EXPRESS



express.js

```
1 var express=...require('express'), compress = require('compression'),
2 bodyParser = require('body-parser'),
3 methodOverride = require('method-override');
4 session = require('express-session');

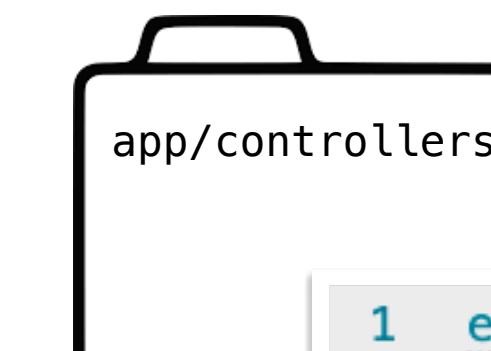
5

6 module.exports = function() {
7     var app = express();
8     app.use(compress());
9     app.use(bodyParser.urlencoded({extended: true}));
10    app.use(bodyParser.json());
11    app.use(methodOverride());
12

13    app.use(session({
14        secret: '1234567890QWERTY'
15    }));

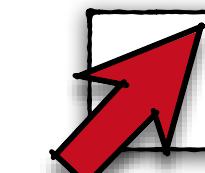
16

17    app.set('views', './app/views');
18    app.set('view engine', 'ejs');
19    require('../app/routes/index.server.routes.js')(app);
20    app.use(express.static('./public'));
21    return app;
22};
```



index.server.controller.js

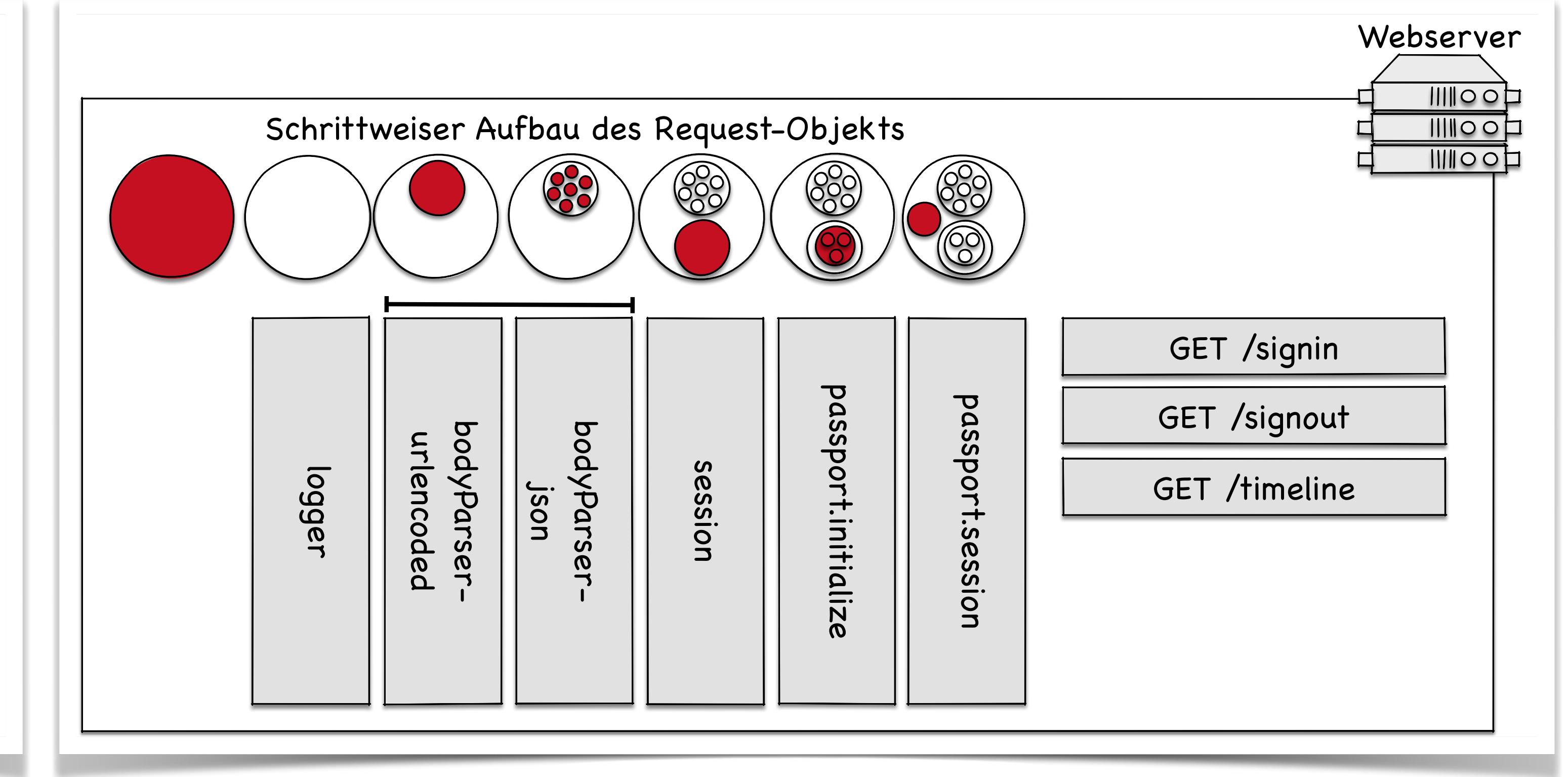
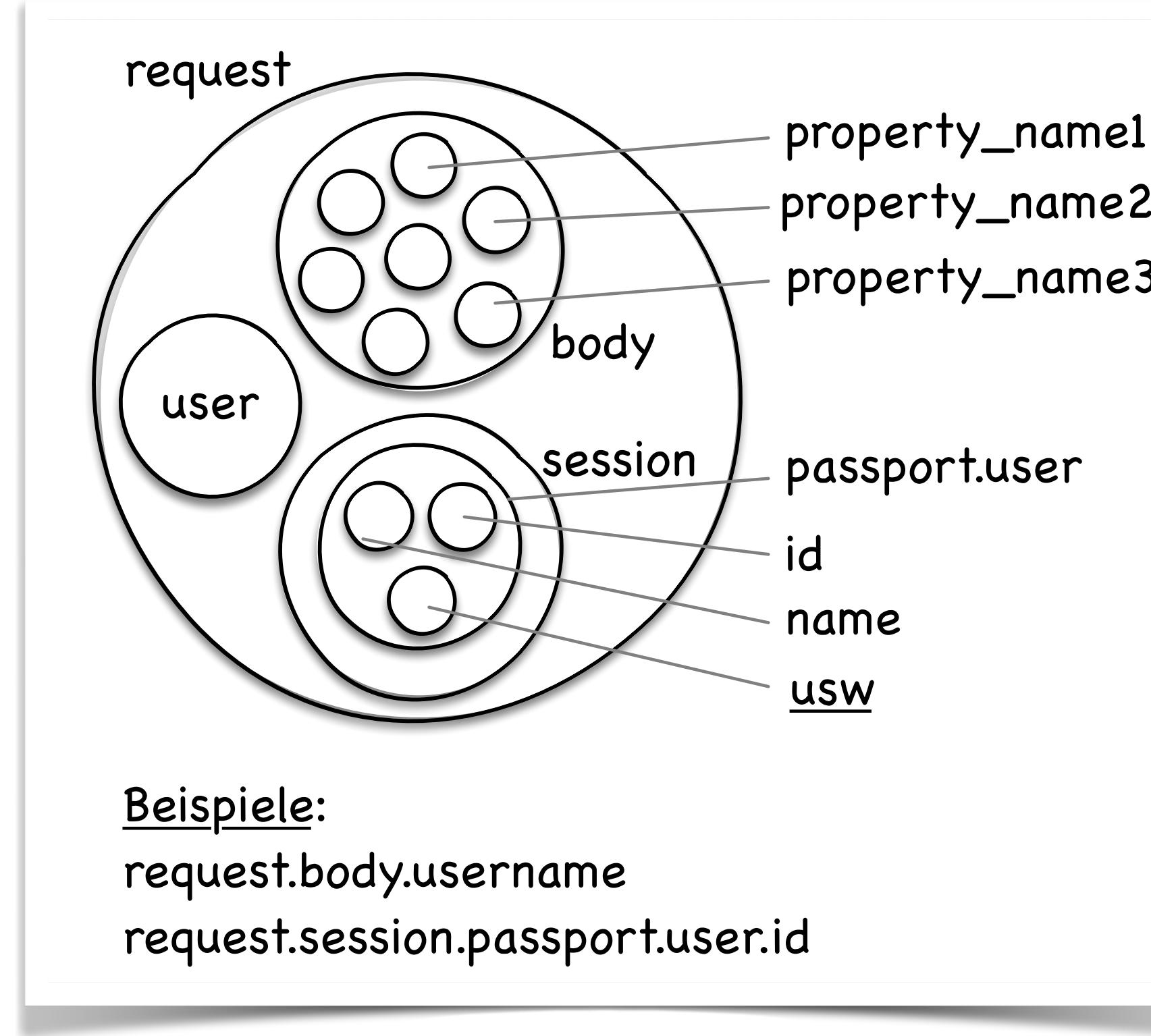
```
1 exports.render=function(req, res) {
2     if (req.session.lastVisit) {
3         console.log(req.session.lastVisit);
4     }
5
6     req.session.lastVisit = new Date();
7
8     res.render('index', {
9         title: 'Hello World'
10});
```



<http://expressjs.com/en/4x/api.html>

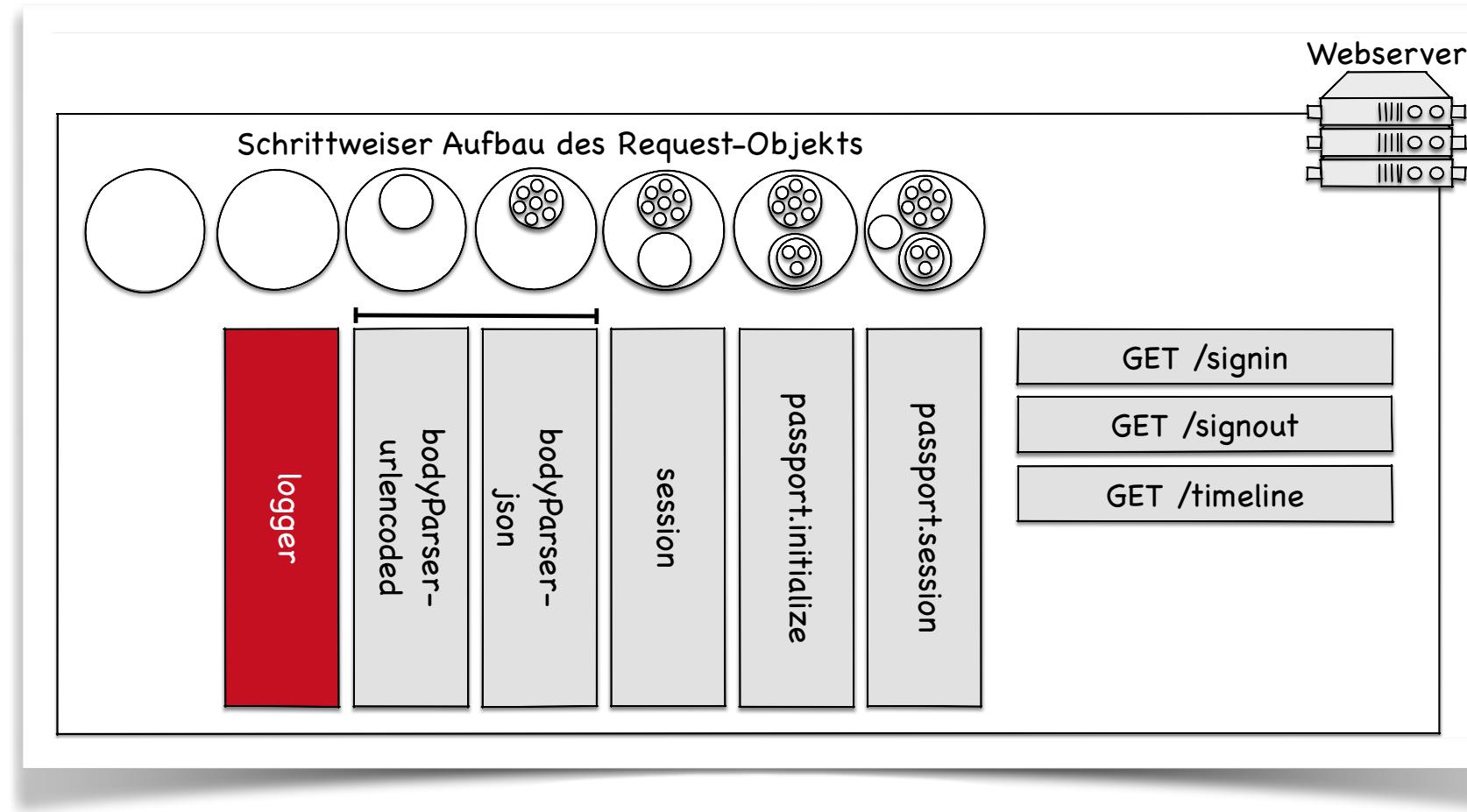
8.4 SITZUNGSMANAGEMENT

YASN-BEISPIEL - VGL. CHAPTER08/EXAMPLE04 (I)



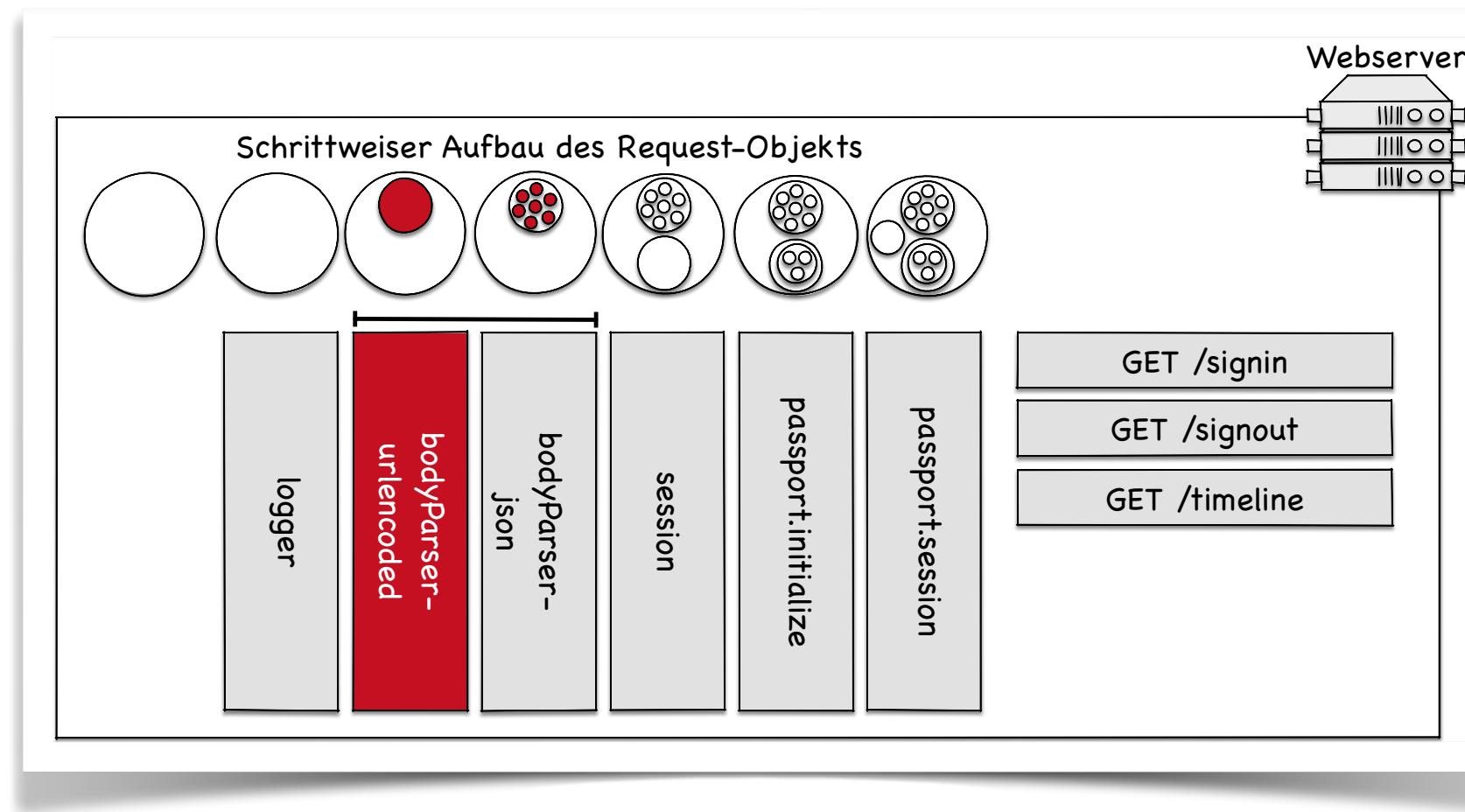
8.4 SITZUNGSMANAGEMENT

YASN-BEISPIEL - VGL. CHAPTER08/EXAMPLE04 (II)



LOGGER

- Ausgabe der gegenwärtigen Anfrage auf der Konsole des Webservers zu Kontrollzwecken

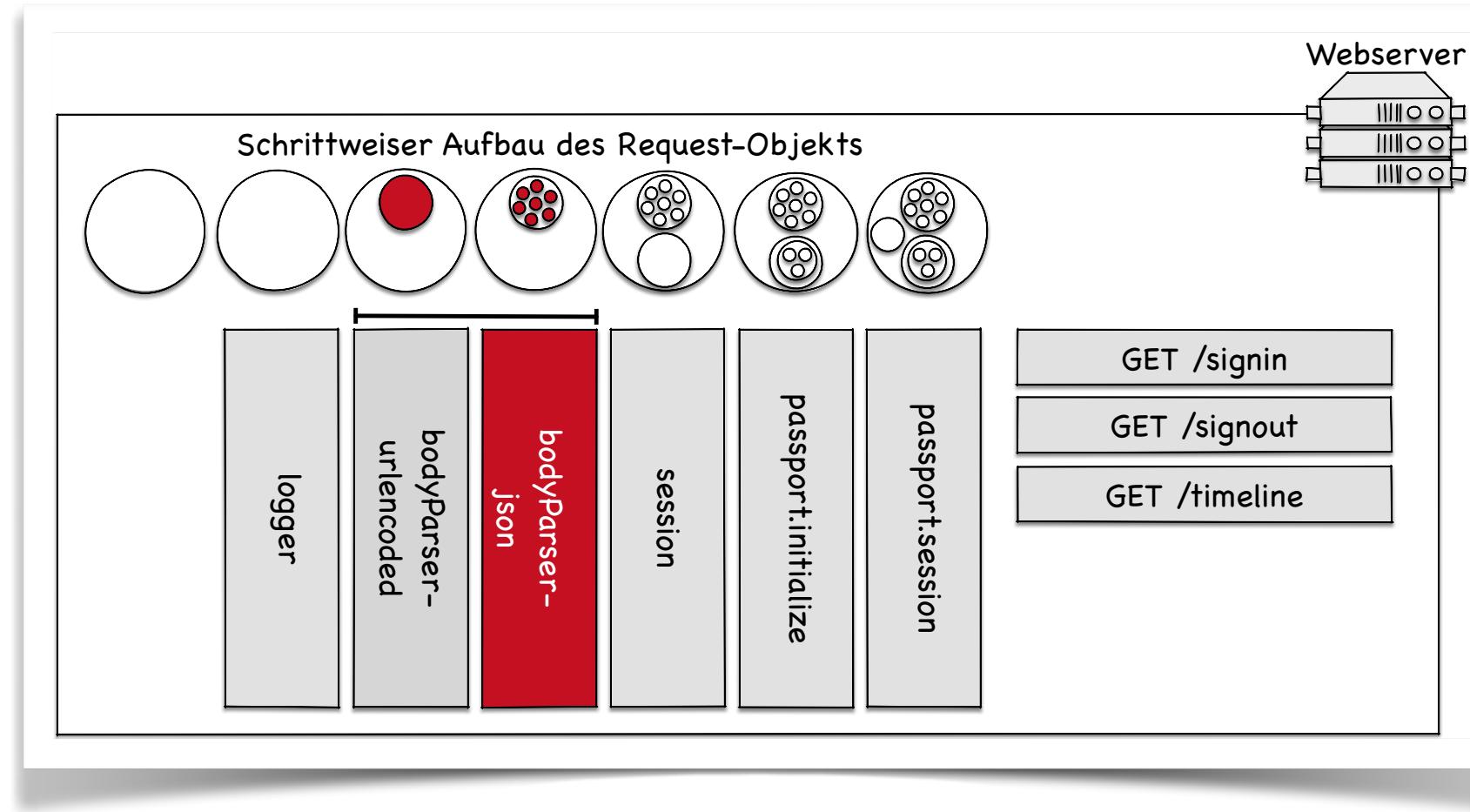


BODYPARSER-URLENCODED

- Parser zum Auslesen von Body-Inhalten die mit **application/x-www-form-urlencoded** codiert wurden
- Standardcodierung von Browsern zur Übertragung von Formularinhalten
- Formularinhalte sind anschließend unter **request.body.property_name** verfügbar

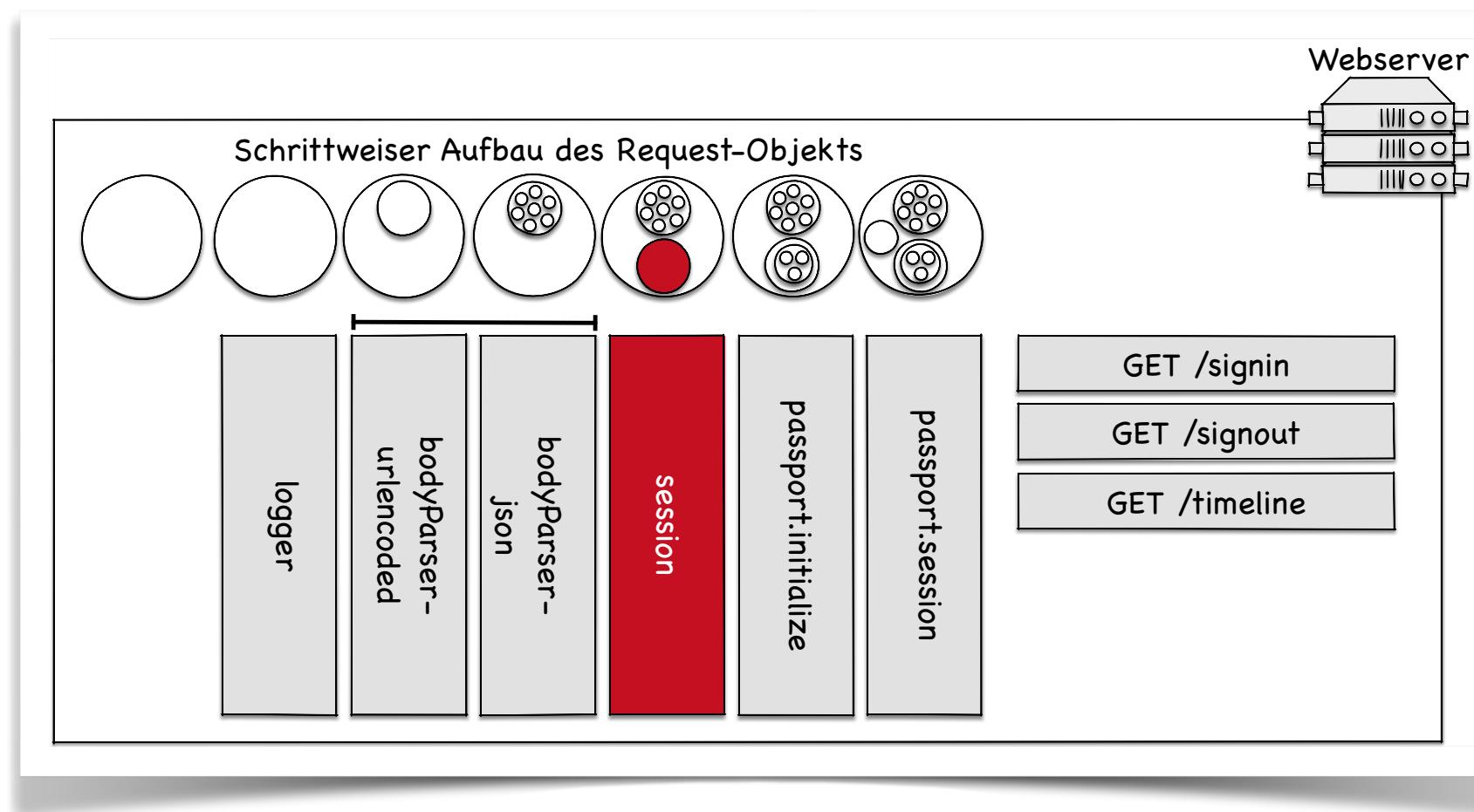
8.4 SITZUNGSMANAGEMENT

YASN-BEISPIEL - VGL. CHAPTER08/EXAMPLE04 (III)



BODYPARSER-JSON

- Parser zum Auslesen von Body-Inhalten die mit **application/json** codiert wurden
- Standardcodierung bei RESTful-Diensten
- Formularinhalte sind anschließend unter **request.body.property_name** verfügbar

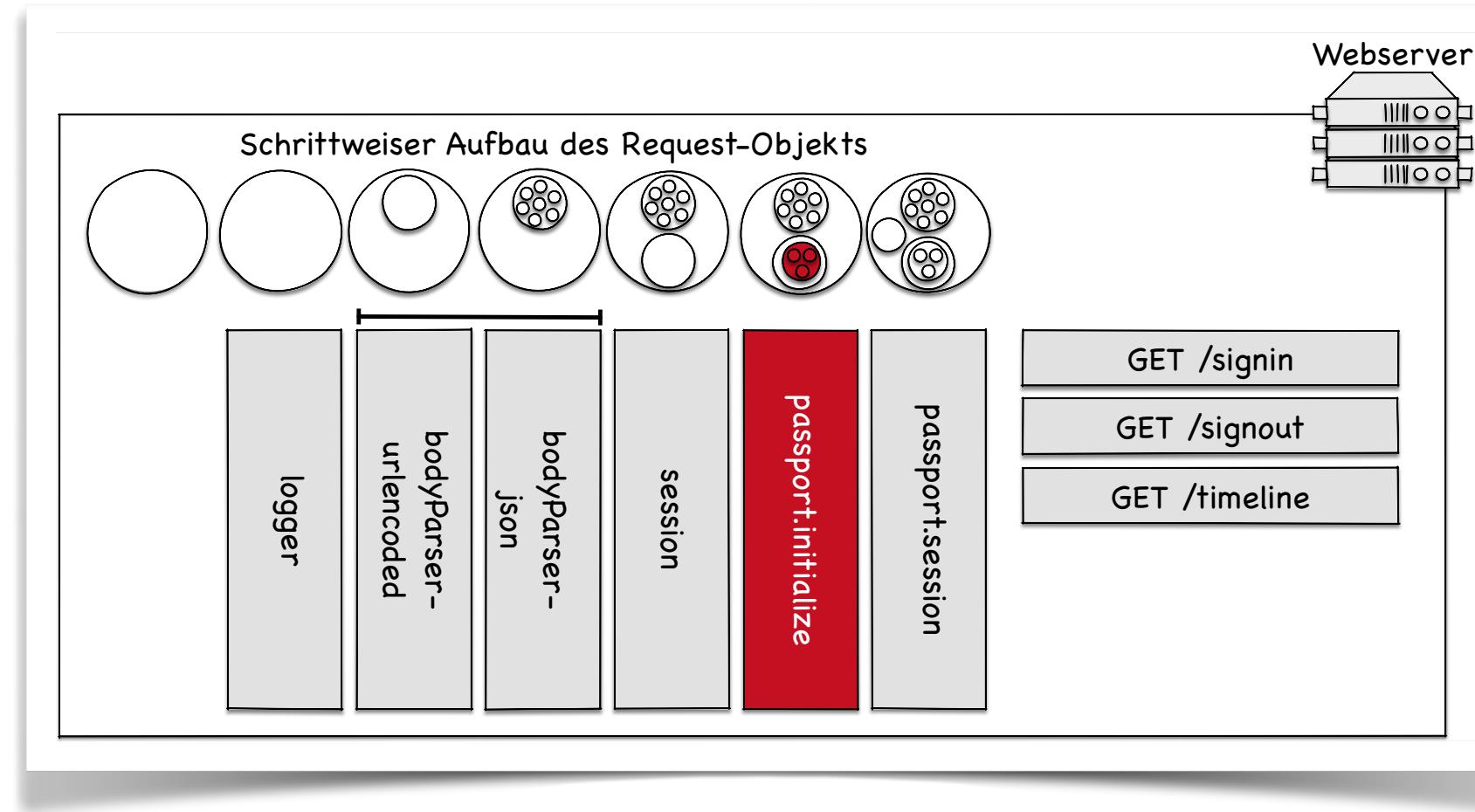


SESSION

- Prüft ob HTTP-Request ein Session-Cookie enthält
- Wenn, dann wird dem Request-Objekt das im Speicher befindliche Session-Objekt der letzten Anfrage angehängt und das Cookie in der HTTP-Antwort ggf. erneuert
- Wenn nicht, dann wird dem Request-Objekt ein neues, leeres Session-Objekt angehängt und in der HTTP-Antwort ein neues Cookie gesetzt

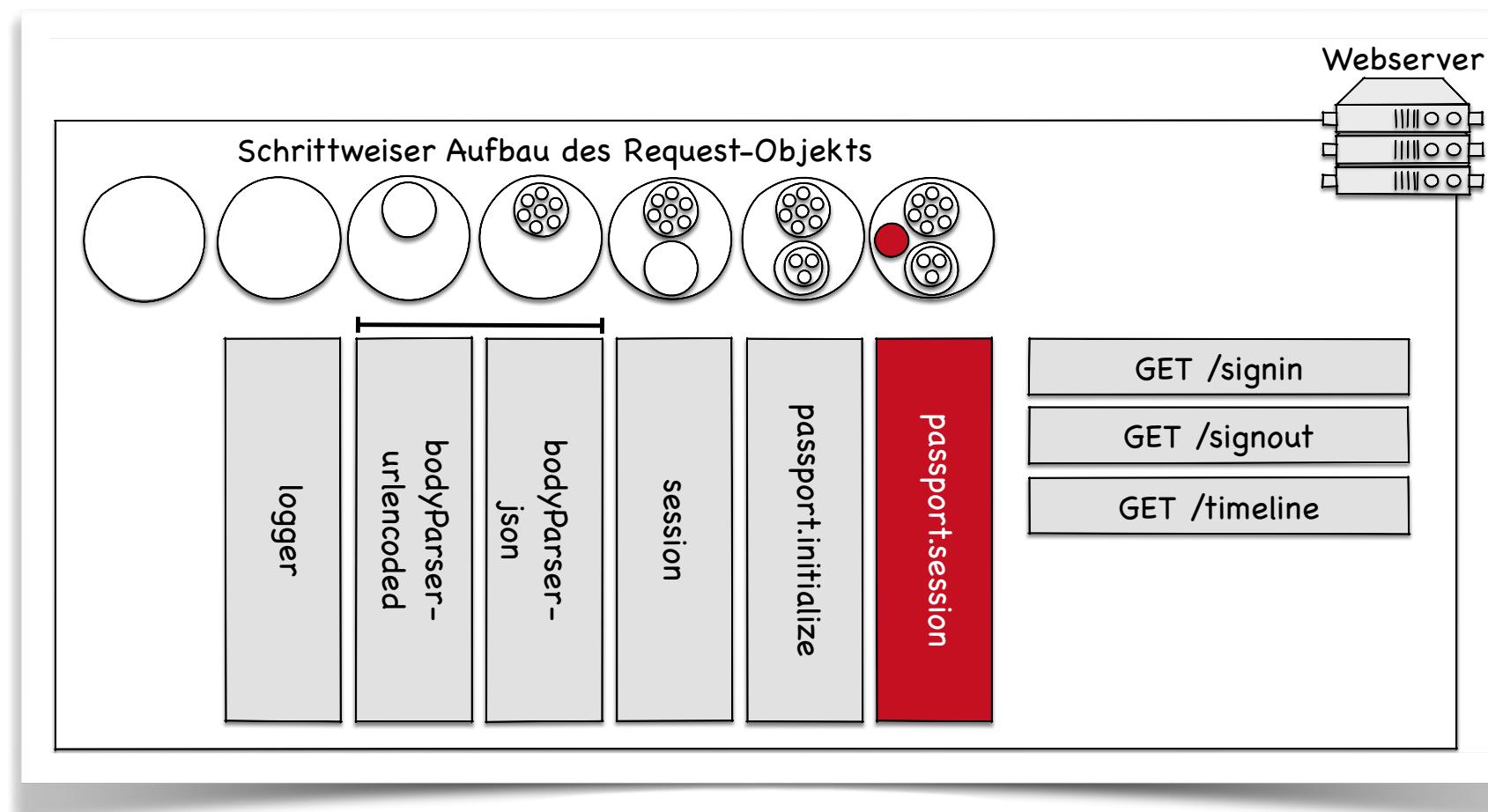
8.4 SITZUNGSMANAGEMENT

YASN-BEISPIEL - VGL. CHAPTER08/EXAMPLE04 (IV)



PASSPORT.INITIALIZE

- Middleware zur Überprüfung, ob **request.session.passport.user** existiert
- Wenn nicht, wird **request.session.passport.user={}** gesetzt, d.h. ein leeres User-Objekt der Session hinzugefügt

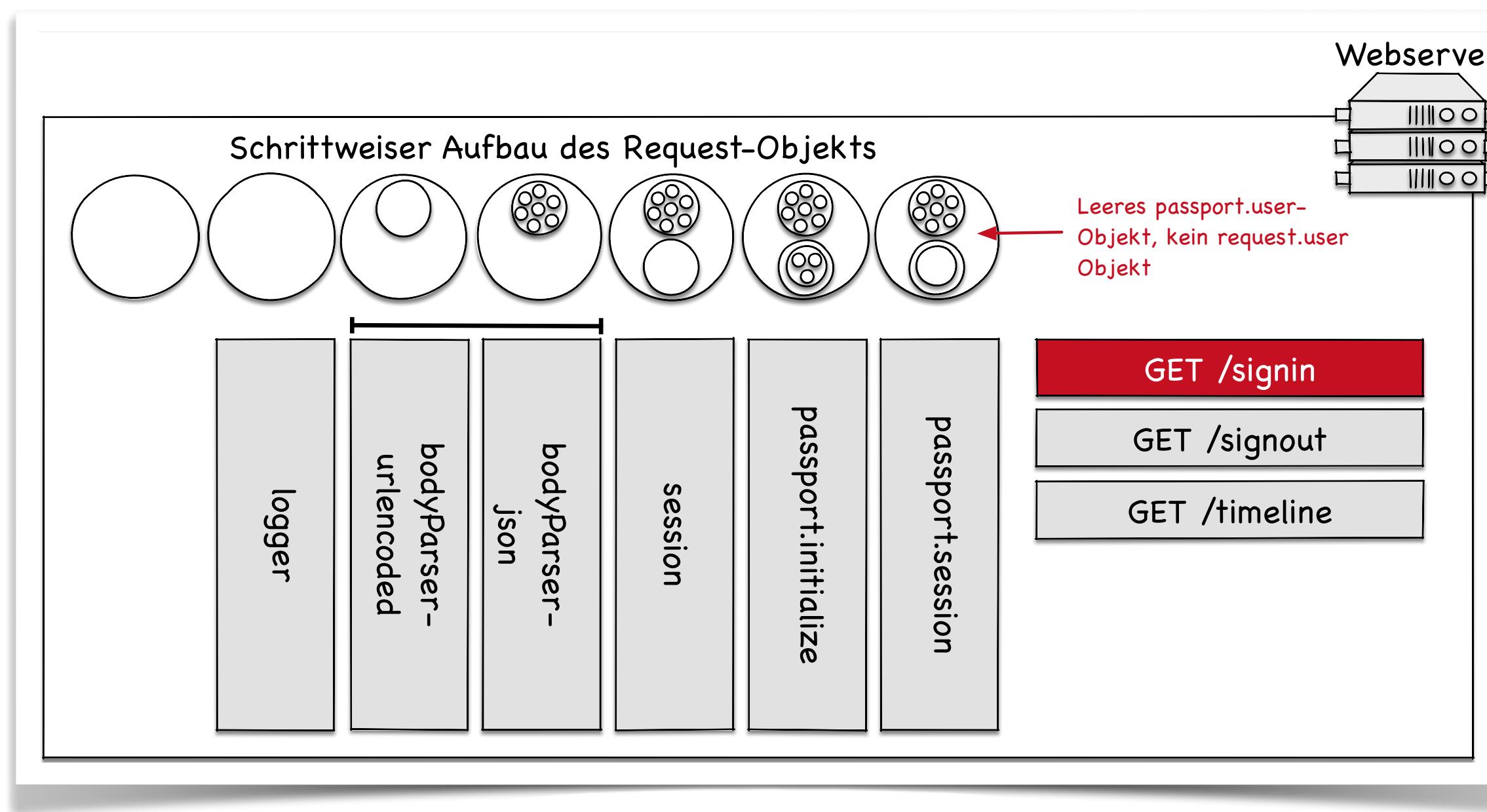


PASSPORT.SESSION

- Wenn die Middleware ein nicht-leeres User-Objekt in Session findet, wird die Methode **passport.deserializeUser** aufgerufen
- Methode stellt ein Objekt **request.user** zur Verfügung, zum Beispiel mit den Nutzerdaten, die aus einer Datenbank geladen werden

8.4 SITZUNGSMANAGEMENT

YASN-BEISPIEL - VGL. CHAPTER08/EXAMPLE04 (V)



AUFRUF VON POST /SIGNIN

- Nutzernname und Passwort werden urlencoded und mittels **POST /signin** zum Server übertragen
- Zuständiger Controller ruft die Passwortstrategie auf, die beim Start des Servers mittels **passport.use** übergeben wurde
- Passwortstrategie validiert **request.body.username** und **request.body.password**
- Bei Erfolg werden Nutzerdaten aus der Datenbank geladen, im Fehlerfall wird ein Fehler-Code zurückgegeben
- Die Methode **passport.serializerUser** wird aufgerufen, um die User-Id (optional weitere Parameter) in das Objekt **request.session.passport.user** einzufügen
- Die Methode **passport.deserializeUser** wird aufgerufen, um **request.user** mit Daten aus der Nutzerdatenbank zu füllen

8.4 SITZUNGSMANAGEMENT

YASN-BEISPIEL - VGL. CHAPTER08/EXAMPLE04 (VI)



```
1 var express=require('express');
2 var passport=require('passport');
3 var localStrategy=require('passport-local').Strategy
4 var path=require('path');
5 var bodyParser=require('body-parser');
6 var session=require('express-session');
7 var url=require('url');
8
9 module.exports=function() {
10
11     //eigene Middleware zur Ausgabe der Anfrage auf der Konsole
12     var logger=function(req, res, next) {
13         console.log(req.method, req.url);
14         next();
15     }
16
17     //Passwortstrategie zum Signin – Nutzer loggt sich mit Username und
18     //Passwort ein
19     passport.use(new localStrategy(
20         function(username, password, done) {
21             if (username==='jamesbond' && password==='abc') {
22                 console.log('>>>Login success');
23                 return done(null, {id: '123', username: 'jamesbond'});
24             }
25             else {
26                 console.log('>>>Login failure');
27                 return done('401', false);
28             }
29         });
30     );
31 }
```

8.5 MONGODB

EINE DOKUMENTENORIENTIERTE DATENBANK



MERKMALE

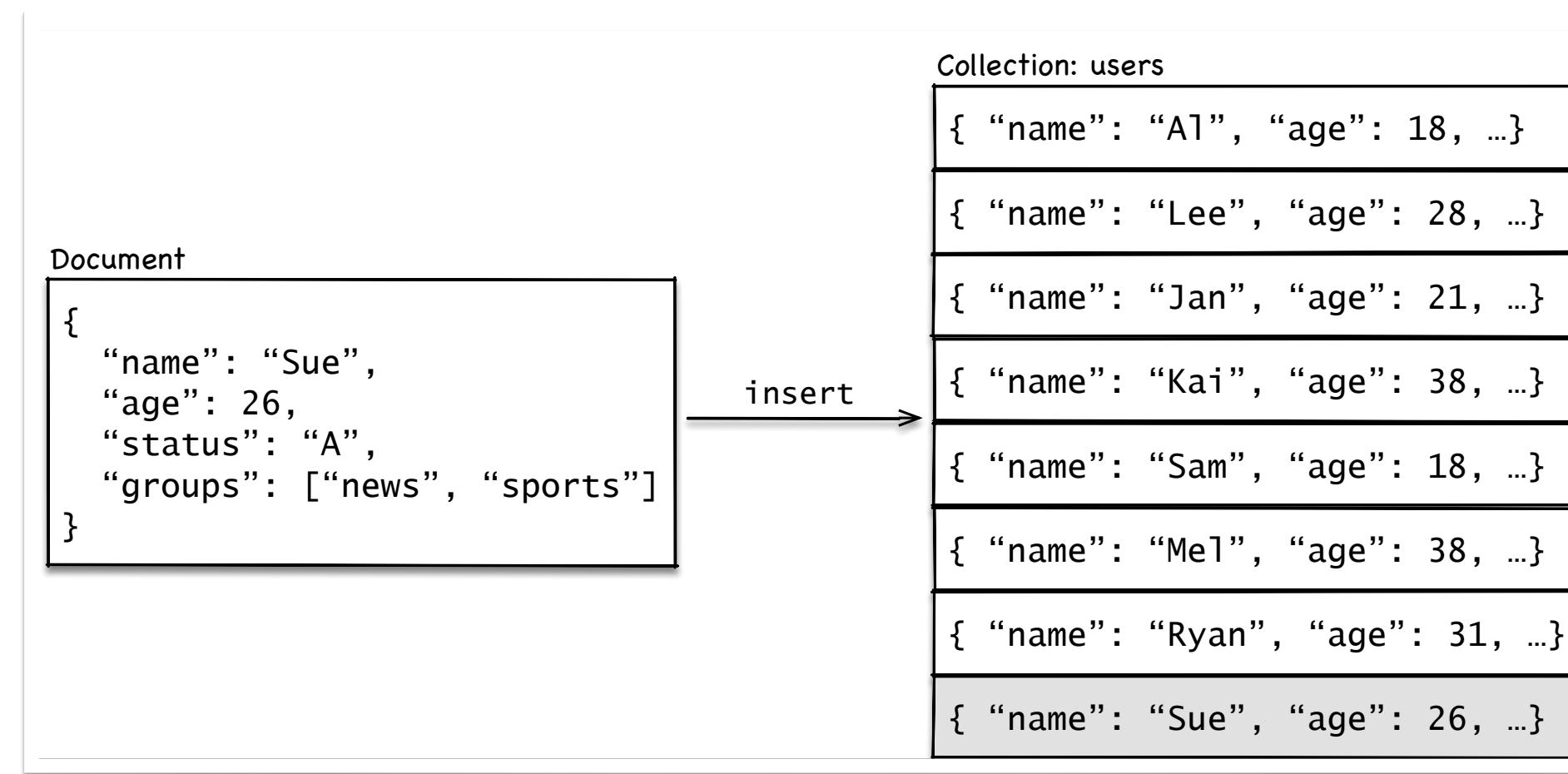
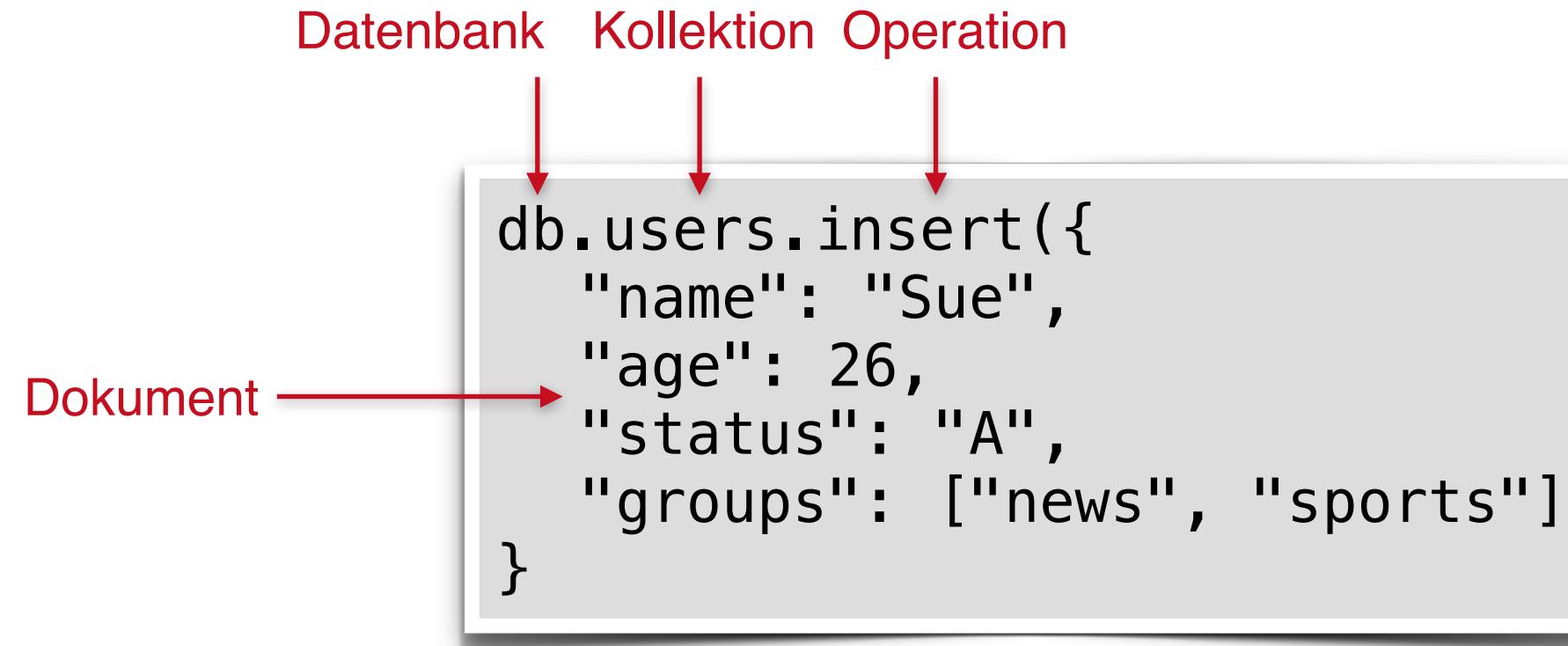
- MongoDB ist eine dokumentenorientierte Datenbank
- Daten werden in Dokumenten gespeichert
- Dokumente werden in Kollektionen verwaltet
- Dokumente entsprechen Zeilen einer Tabelle in einer relationalen Datenbank
- Kollektionen entsprechen Tabellen in einer relationalen Datenbank
- MongoDB ist schemafrei - Dokumente einer Kollektion müssen nicht einem bestimmten, vorgegebenen Schema folgen

BSON

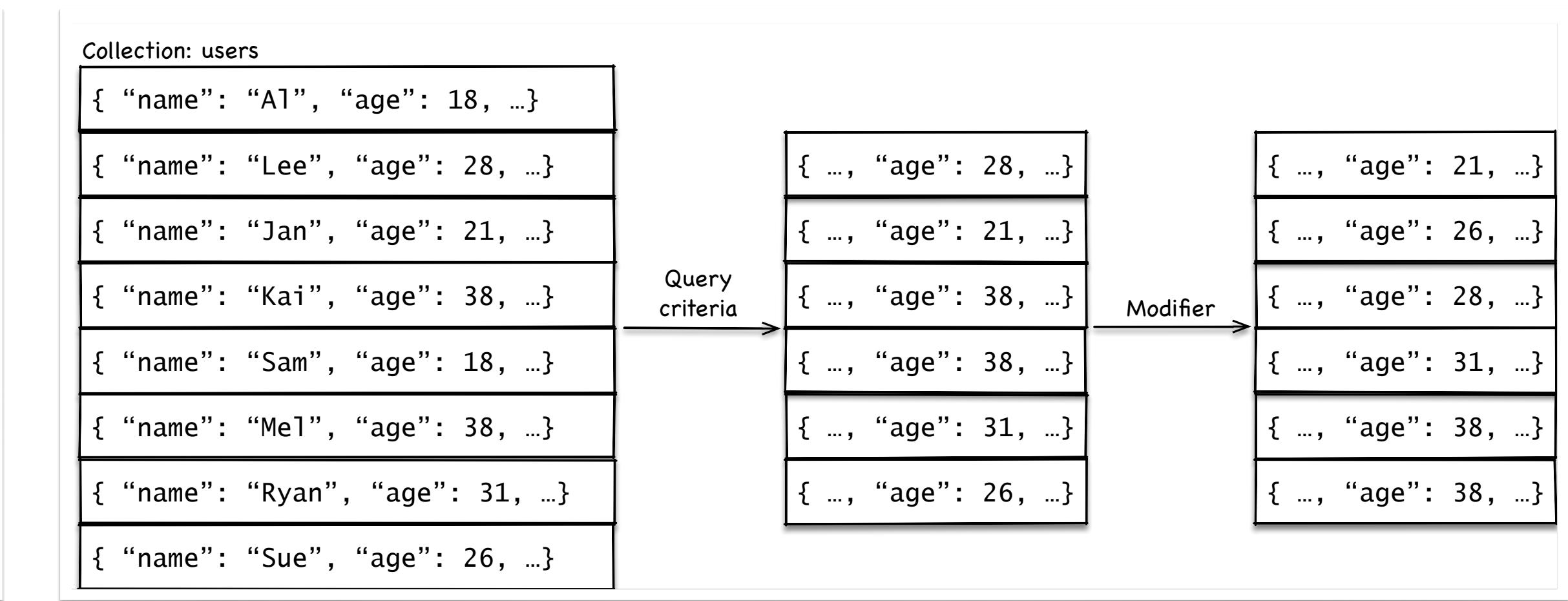
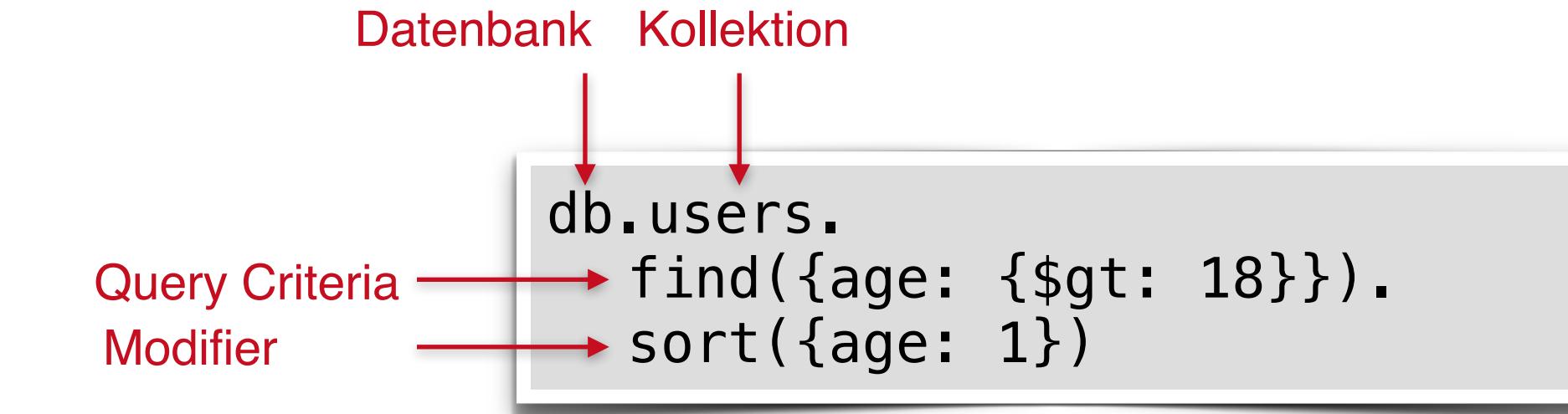
- Binary JSON: binäre codierte Serialisierung von JSON
- Zugrunde liegendes Dokumentenformat
- Unterstützung der JSON-Datenformate, plus zusätzlicher Formate (z.B. **date** oder **byte array**)

8.5 MONGODB EINFÜHRENDE BEISPIELE

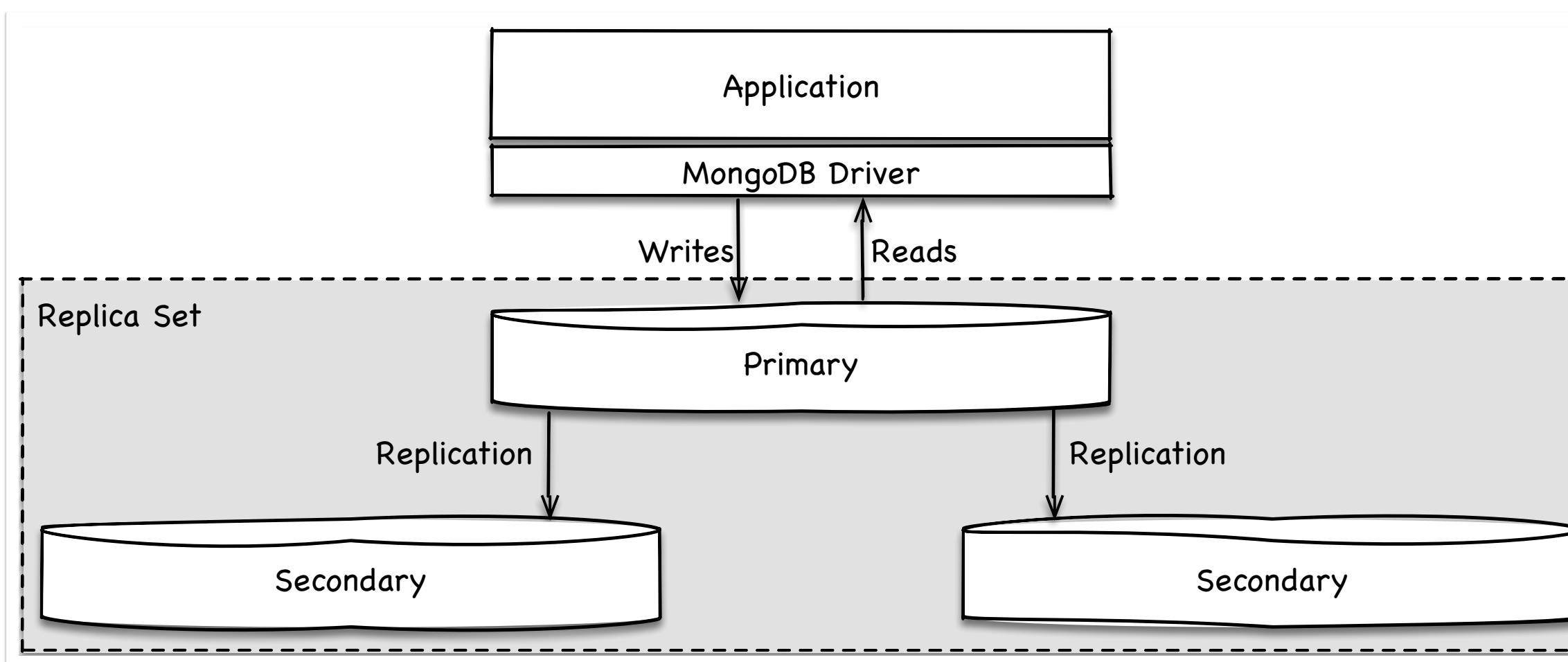
BEISPIEL FÜR SCHREIBOPERATIONEN



BEISPIEL FÜR LESEOPERATIONEN

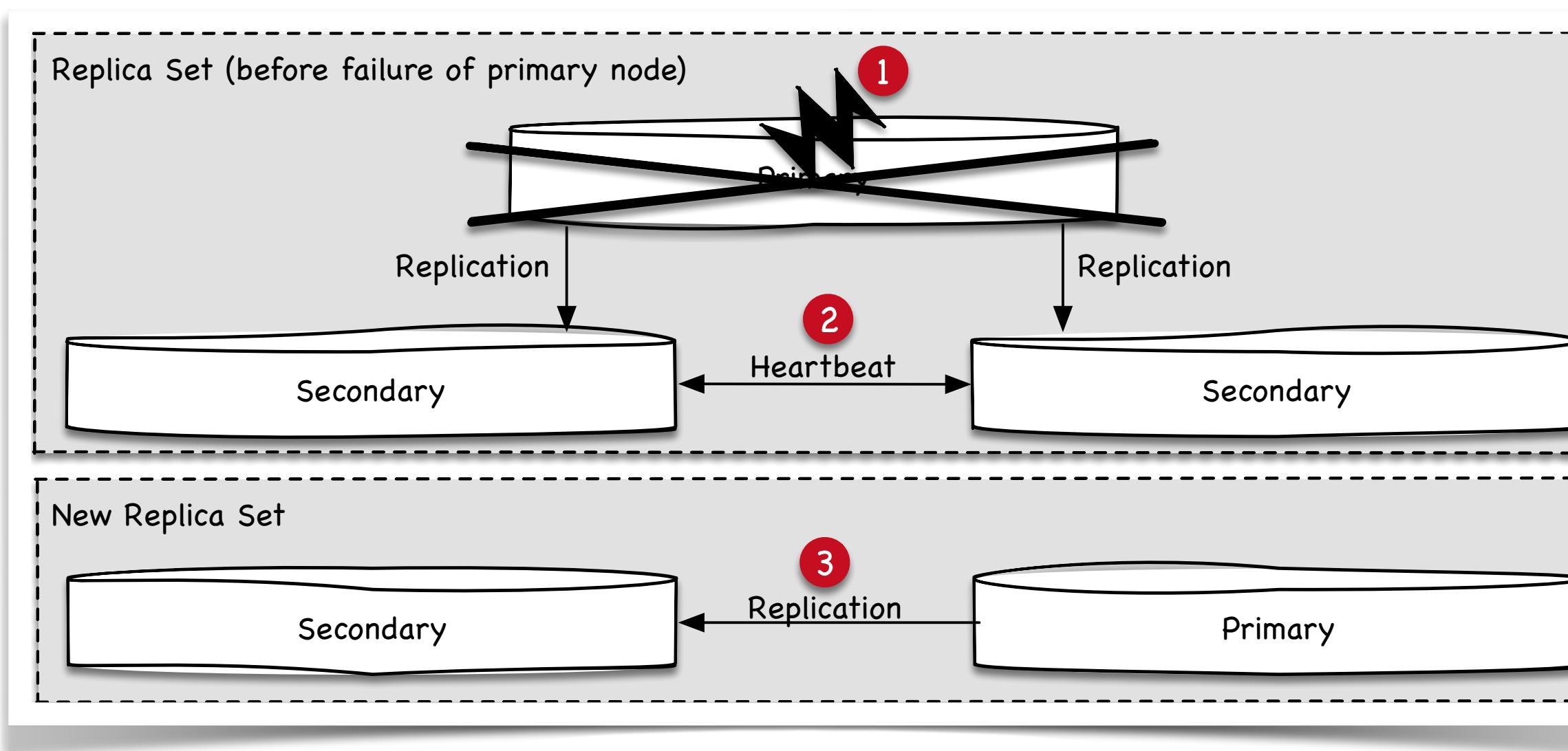


8.5 MONGODB REPLIKATION (I)



- *Replikation*: Vorhalten mehrfacher Kopien von Daten auf verschiedenen Servern
- Replikation ermöglicht Datenredundanz und erhöht Datenverfügbarkeit
- Voraussetzung für Fehlertoleranz: Schutz vor Datenverlust und Nichtverfügbarkeit bei Ausfall eines Datenbank-Servers
- Erhöhung der Lesekapazität: Leseoperation können von verschiedenen Datenbank-Servern ausgeführt werden
- Verminderung von Latenzen durch Lokalitätsprinzip: Speicherung von Kopien der Daten in geographisch verteilten Rechenzentren in Kombination mit Geo-Routing der Leseanfragen

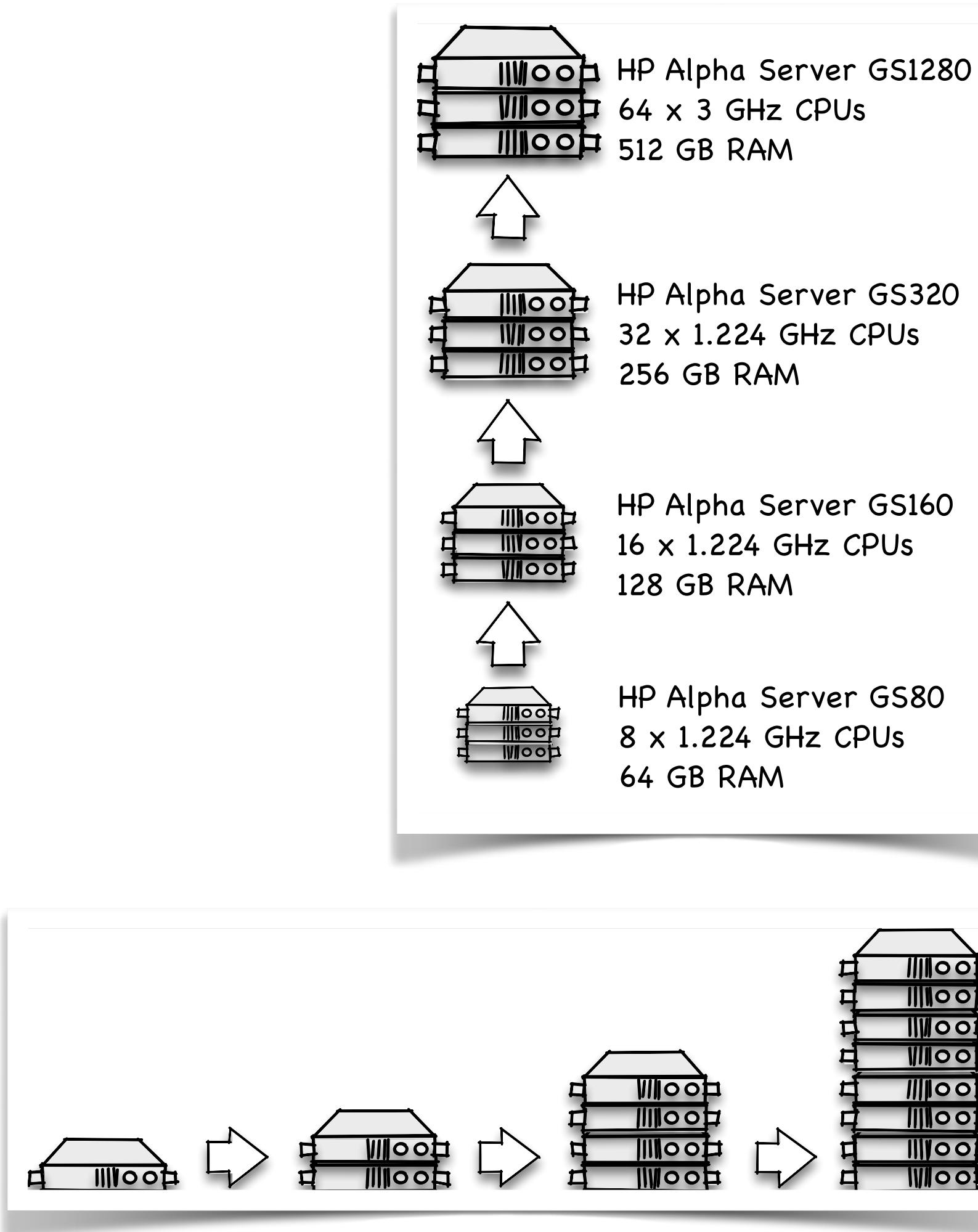
8.5 MONGODB REPLIKATION (II)



REPLIKATION MIT MONGODB

- Replica Set: Gruppe von MongoDB-Instanzen mit den gleichen Datensätzen
- Primary Node: MongoDB-Instanz eines Replica Set, welche Schreiboperationen entgegennimmt
- Secondary Nodes: ein oder mehrere MongoDB-Instanzen eines Replica Set, welche Datensätze regelmäßig vom Primary Node replizieren und optional Leseanfragen entgegennehmen
- Bei Ausfall des Primary Node wird aus der Gruppe der verbleibenden Secondary Nodes ein neuer Primary gewählt

8.5 MONGODB SKALIERBARKEIT (I)



- Skalierbarkeit: Fähigkeit eines Systems seine Leistung durch Hinzufügen von Ressourcen zu steigern

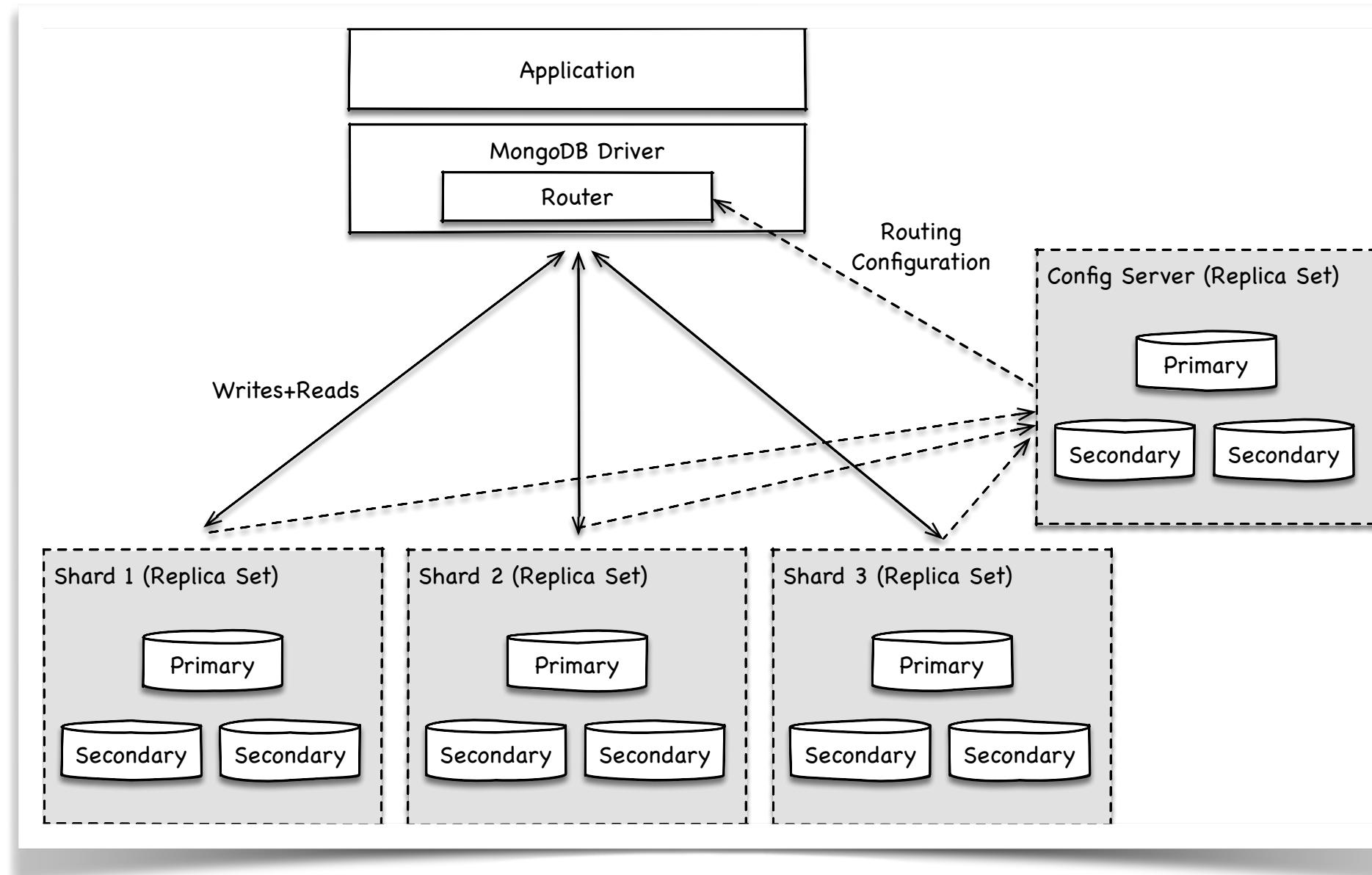
VERTIKALE SKALIERBARKEIT

- Steigerung der Leistung eines Servers durch Hinzufügen oder den Austausch zusätzlicher, leistungsfähigerer Hardware-Ressourcen (CPU, Hauptspeicher, Bandbreite,...)
- Unabhängig von der Softwarearchitektur und Implementierung
- Überproportionale Hardware-Kosten ab einer bestimmten Ausbaustufe
- Begrenzt durch die Leistungsfähigkeit verfügbarer Hardware

HORIZONTALE SKALIERBARKEIT

- Steigerung der Leistung eines Systems durch Hinzufügen zusätzlicher Server
- Erfordert verteilte, auf Skalierbarkeit ausgelegte Softwarearchitektur
- Theoretisch unbegrenzte Skalierbarkeit

8.5 MONGODB SKALIERBARKEIT (II)



- Sharding: Partitionierung der Datenmenge und Verteilung auf mehrere Server zur Umsetzung horizontaler Skalierbarkeit
- Range-based Partitioning versus Hash-based Partitioning
- Sharding reduziert die Datenbankoperationen, die ein Server ausführen muss, und reduziert das Datenvolumen pro Server

SHARDS

- Speichern jeweils eine Teilmenge der gesamten Daten
- Realisierung als einzelne Instanz oder als Replica Set

ROUTERS

- Bestandteil des MongoDB-Treibers bei der Anwendung
- Leiten Anfragen der Anwendung an einen oder mehrere zuständige Shards basierend auf Routing-Informationen
- Horizontale Skalierbarkeit durch Hinzufügen mehrerer Router

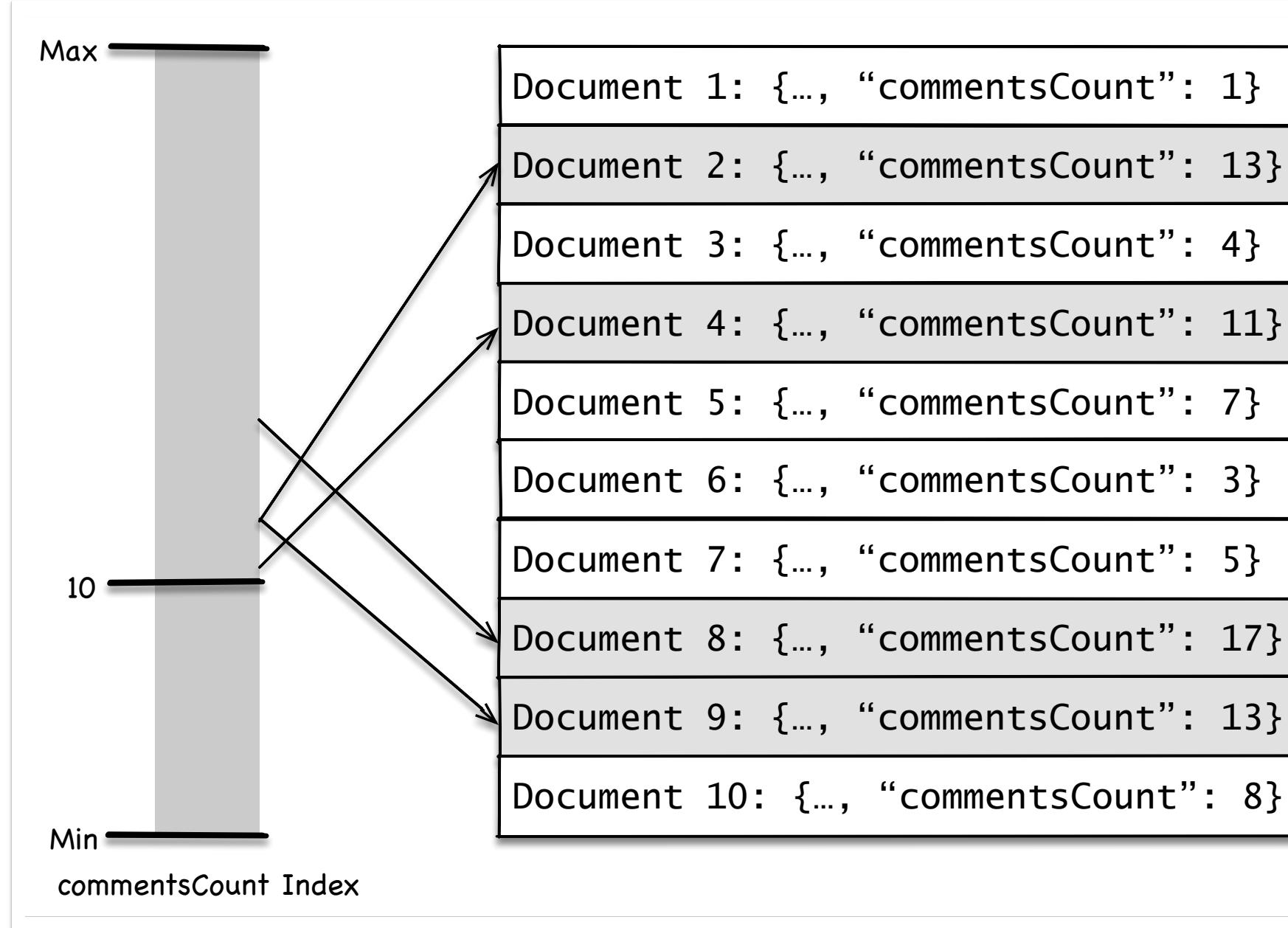
SHARD KEY

- Indiziertes Feld in jedem Dokument der Datenbank
- Grundlage für die Partitionierung der Datenmenge in Chunks
- Gleichmäßige Verteilung der Chunks auf verfügbare Shards

CONFIG SERVER

- Verwaltung der Routing-Informationen basierend auf Zuweisung der Chunks zu Shards

8.5 MONGODB INDEXIERUNG



- **Datenbankindex:** von der Datenstruktur getrennte Indexstruktur in einer Datenbank, welche die Suche und das Sortieren nach bestimmten Datenbankfeldern beschleunigt
- Anstelle alle Felder eines Dokumentes bei einer Anfrage zu überprüfen, liefert ein Index eine schnelle Referenz auf alle Dokumente, welche die Bedingungen der Anfrage erfüllen

BEISPIEL

- MongoDB-Datenbank zur Verwaltung von Einträgen eines Blogs: pro Dokument werden ein Blögeintrag und alle dazu eingegangenen Kommentare gespeichert
- Dediziertes Feld für den schnellen Zugriff auf die Anzahl der eingegangenen Kommentare

```
{  
  "_id": ObjectId("52d02240eb01d67d71ad577"),  
  "title": "First Blog Post",  
  "comments": [  
    ...., ...., ....  
  ],  
  "commentsCount": 12  
}
```

- Index auf dem Feld commentsCount macht Dokumente schnelle auffindbar:

```
db.posts.find({ "commentsCount": { $gt: 10 } });
```

8.5 MONGODB CRUD-OPERATIONEN (I)

```
db.posts.insert({  
    "title": "Second Post",  
    "user": "Alice"  
})
```

```
db.posts.update({  
    "user": "Alice"  
, {  
    "title": "Second Post",  
    "user": "Alice"});  
, {  
    upsert: true  
})
```

```
db.posts.save({  
    "title": "Second Post",  
    "user": "Alice"  
})
```

```
db.posts.save({  
    "_id":  
    Object("45c230a123d45efab2340",  
    "title": "Second Post",  
    "user": "Alice"  
})
```

EINFÜGEN NEUER DOKUMENTE

- **insert(document)**: häufigste Methode zum Einfügen neuer Dokumente in eine Kollektion
- **document** als einziges Argument der Methode
- **_id** wird automatisch generiert und dem Dokument zugewiesen
- Verursacht ggfs. das Anlegen einer neuen Kollektion, wenn die angegebene nicht existiert
- **update(selectionCriteria, updateStatement, options)**: alternative Methode zum Einfügen von Dokumenten
- **upsert** ist **true** wenn ein neues Dokument angelegt werden soll wenn kein Dokument mit den Suchkriterien in der Kollektion vorhanden ist
- **save(document)**: weitere Alternative, die optional die Definition einer eigenen **_id** erlaubt



<https://docs.mongodb.org/manual/reference/>

8.5 MONGODB CRUD-OPERATIONEN (II)

Liefert alle Dokumente einer Kollektion

```
db.posts.find()
```

```
db.posts.find({})
```

Liefert alle Dokumente mit
user=Alice

```
db.posts.find({ "user":  
    "Alice" })
```

Liefert alle Dokumente mit
user=Alice **oder** user=Bob

```
db.posts.find(  
    { "user": { $in: ["Alice",  
        "Bob"] } }  
)
```

Liefert alle Dokumente mit
user=Alice **und**
commentsCount>=10

```
db.posts.find(  
    { "user": "Alice",  
        "commentsCount": { $gt: 10 } }  
)
```

Liefert alle Dokumente mit
user=Alice **oder** user=Bob

```
db.posts.find(  
    { $or: [ { "user": "Alice" },  
        { "user": "Bob" } ] }  
)
```

LESEN VON DOKUMENTEN

- **find(selectionCriteria)**: Methode zum Finden und Lesen von Dokumenten einer Kollektion
- **selectionCriteria** enthält einfache oder zusammengesetzte Anfrageselektoren
- Beachte: alle MongoDB-Operatoren werden mit einem \$ eingeleitet



<https://docs.mongodb.org/manual/reference/>

8.5 MONGODB CRUD-OPERATIONEN (III)

```
db.posts.update({  
  "user": "Alice"  
}, {  
  $set: {  
    "title": "Second  
    Post",  
  }  
}, {  
  multi: true  
})
```

Entfernt alle Dokumente der Kollektion posts

```
db.posts.remove()
```

Entfernt alle Dokumente für die user=Alice gilt

```
db.posts.remove({"user": "Alice"})
```

Entfernt ein Dokument für das user=Alice gilt

```
db.posts.remove({"user": "Alice"},  
true)
```

AKTUALISIEREN VON DOKUMENTEN

- **update(selectionCriteria, updateStatement, options)**: Methode zum Aktualisieren von Dokumenten einer Kollektion (oder zum Einfügen eines neuen Dokuments falls **selectionCriteria** durch kein existierendes Dokument erfüllt werden)
- **selectionCriteria**: enthält einfache oder zusammengesetzte Anfrageselektoren
- **updateStatement**: spezifiziert die zu aktualisierenden Felder
- **options**: verschiedene Optionen, zum Beispiel ob ein oder mehrere Dokumente aktualisiert werden sollen

LÖSCHEN VON DOKUMENTEN

- **remove(selectionCriteria, justOne)**: Methode zum Löschen von Kollektionen oder einer oder mehrerer Dokumente
- **selectionCriteria**: s.o.
- **justOne**: zeigt an, dass bei mehreren Dokumenten, die **selectionCriteria** erfüllen, nur eines gelöscht wird



<https://docs.mongodb.org/manual/reference/>

8.6 MONGOOSE

DEFINITION VON MODELLEN FÜR ANWENDUNG DES MVC-PATTERN

```
server.js
1 var mongoose = require('./config/mongoose'),
2 express = require('./config/express');
3
4 var db = mongoose();
5 var app = express();
6 app.listen(3000);
7 module.exports = app;
8
9 console.log('Server is running at http://localhost:3000/');

config
mongoose.js
1 var mongoose = require('mongoose');
2
3 module.exports = function() {
4   var db = mongoose.connect('mongodb://localhost/mongodb');
5   require('../app/models/user.server.model');
6   return db;
7 }

express.js
1 module.exports = function() {
2   var app = express();
3   app.use(compress());
4   app.use(bodyParser.urlencoded({extended: true}));
5   app.use(bodyParser.json());
6   app.use(methodOverride());
7 }
```

- Node.js Modul für die Anbindung einer MongoDB an eine Express-Webanwendung
- Definition von Schemata als Grundlage für MVC-Modelle
- MongoDB ist eigentlich schemafrei, erlaubt jedoch die Definition von Schemata als Grundlage für Modelle im MVC-Pattern
- Modelle werden beim Start von MongoDB eingebunden und stehen dann der gesamten Webanwendung zur Verfügung

```
app/models
user.server.model.js
1 var mongoose = require('mongoose'),
2 Schema = mongoose.Schema;
3
4 var UserSchema = new Schema({
5   firstName: String,
6   lastName: String,
7   email: String,
8   username: String,
9   password: String
10 });
11
12 mongoose.model('User', UserSchema);
```

8.6 MONGOOSE

CRUD - EINFÜGEN VON DOKUMENTEN (I)

```
config  
  
express.js  
  
mongoose.js
```

```
1 var express=require('express'),  
2 compress = require('compression'),  
3 bodyParser = require('body-parser'),  
4 methodOverride = require('method-override');  
5 session = require('express-session');  
6  
7 module.exports = function() {  
8     var app = express();  
9     app.use(compress());  
10    app.use(bodyParser.urlencoded({extended: true}));  
11    app.use(bodyParser.json());  
12    app.use(methodOverride());  
13    app.use(session({secret: '1234567890QWERTY'}));  
14    app.set('views', './app/views');  
15    app.set('view engine', 'ejs');  
16    require('../app/routes/index.server.routes.js')(app);  
17    require('../app/routes/users.server.routes.js')(app);  
18    app.use(express.static('./public'));  
19    return app;  
20 };  
  
4     var db = mongoose.connect('mongodb://localhost/mongodb');  
5     require('../app/models/user.server.model');  
6     return db;  
7 }
```

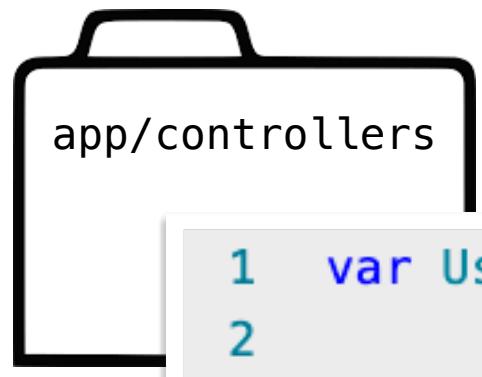
```
app/routes  
  
index.server.routes.js  
  
users.server.routes.js  
  
app/controllers  
  
index.server.controller.js  
  
users.server.controller.js
```

```
1 module.exports=function(app) {  
2     ...  
3     1 var users = require('../app/controllers/users.server.controller');  
4     2  
5     3 module.exports = function(app) {  
6         4     app.route('/users').post(users.create);  
7         5     };  
8 };
```

```
1 exports.render=function(req, res) {  
2     ...  
3     1 var User = require('mongoose').model('User');  
4     2  
5     3 exports.create = function(req, res, next) {  
6         4     var user = new User(req.body);  
7         5     user.save(function(err) {  
8             6         if (err) {  
9                 7             return next(err);  
10            8         } else {  
11                9                 res.json(user);  
12            10           }  
13        11       });  
14    12   };  
15 };
```

8.6 MONGOOSE

CRUD - EINFÜGEN VON DOKUMENTEN (II)



```
app/controllers
    users.server.controller.js

1 var User = require('mongoose').model('User');
2
3 exports.create = function(req, res, next) {
4     var user = new User(req.body);
5     user.save(function(err) {
6         if (err) {
7             return next(err);
8         } else {
9             res.json(user);
10        }
11    });
12};
```

Datenstruktur im Body der HTTP-
POST-Anfrage

```
{
  "firstName": "Max",
  "lastName": "Mustermann",
  "email": "max@mustermann.de",
  "username": "maxi",
  "password": "ghb344&!er"
}
```

- Controller holt sich zunächst eine Referenz auf das Modell
- Definition einer Callback-Funktion, die bei eingehenden POST-Anfragen auf dem Verzeichnis **/users** aufgerufen wird (siehe vorherige Folie)
- Controller erzeugt eine neue Instanz des Modells **User** (dessen Grundlage das Schema **UserSchema** bildet)
- Bei der Instanziierung werden die zu schreibenden Daten als JSON-Objekt übergeben
- Hier: JSON-Objekt ist im Body der HTTP-Anfrage enthalten
- Struktur der Daten im Body muss dem Schema entsprechen
- Asynchrones Speichern der Daten in der Datenbank durch Aufruf von **user.save**
- Übergabe einer Callback-Funktion
- Im Fehlerfall: Fehlermeldung wird der nächsten Middleware-Funktion übergeben
- Im Erfolgsfall: gelesene Daten werden zur Bestätigung an die Antwort angehängt



<http://mongoosejs.com/docs/guide.html>

8.6 MONGOOSE

CRUD - LESEN VON DOKUMENTEN (I)

app/controllers

```
users.server.controller.js
1 var User = ...
2
3 > exports.create = function(req, res, next) { ...
12 };
13
14 exports.list = function(req, res, next) {
15   User.find({}, function(err, users) {
16     if (err) {
17       return next(err);
18     } else {
19       res.json(users);
20     }
21   });
22};
```

app/routes

```
users.server.routes.js
1 var users = ...
2
3 module.exports = function(app) {
4   app.route('/users')
5     .post(users.create)
6     .get(users.list);
7};
```

- Lesen von Funktionen mittels **find**
- Signatur: **find(query, [fields], [options], [callback])** mit
 - **query**: Selektionskriterien
 - **[fields]**: Angabe der Felder des Dokumentes, die zurück gegeben werden sollen
 - **[options]**: weitere Konfiguration des Suchprozesses
 - **[callback]**: Optionale Callback-Funktion, die ausgeführt wird, sobald das Ergebnis der Anfrage vorliegt
- Beispiel liefert alle Dokumente der Kollektion



8.6 MONGOOSE

CRUD - LESEN VON DOKUMENTEN (II)

app/controllers

users.server.controller.js

```
1 var User = require('mongoose').model('User');
2
3 > exports.create = function(req, res, next) {
12 };
13
14 exports.list = function(req, res, next) {
15   User.find({}, 'username email', function(err, users) {...});
16 };
17
```

app/controllers

users.server.controller.js

```
1 var User = require('mongoose').model('User');
2
3 > exports.create = function(req, res, next) {
12 };
13
14 < exports.list = function(req, res, next) {
15   User.find({}, 'username email', {
16     skip: 10, limit: 10 },
17     function(err, users) {...});
18 };
```

EINSCHRÄNKUNG DER ZURÜCKGEGEBEN DOKUMENTFELDER

- Oberes Beispiel liefert lediglich die Felder **username** und **email** aller Dokumente zurück

VERWENDUNG VON SUCHOPTIONEN

- Unteres Beispiel überspringt die ersten zehn Dokumente (**skip**) und liefert die Felder **username** und **email** der nächsten zehn Dokumente zurück (**limit**)

8.6 MONGOOSE

CRUD - LESEN VON DOKUMENTEN (III)

```
app/controllers
    users.server.controller.js

1 var User = require('mongoose').model('User');
2
3 > exports.create = function(req, res, next) { ...
12 };
13
14 exports.list = function(req, res, next) {
15 };
16
17 < exports.userById = function(req, res, next, id) {
18   User.findOne({_id: id}, function(err, user) {
19     if (err) {
20       return next(err);
21     } else {
22       req.user = user;
23       next();
24     }
25   });
26 };
27
28 < exports.read = function(req, res) {
29   res.json(req.user);
30 };
```

```
app/routes
    users.server.routes.js

1 var users = ...
2
3 module.exports = function(app) {
4   app.route('/users')
5     .post(users.create)
6     .get(users.list);
7
8   app.param('userId', users.userById);
9
10  app.route('users/:userId')
11    .get(users.read);
12};
```

- **findOne**: liefert das erste Dokument, welches die Selektionskriterien erfüllt
- Szenario RESTful API (siehe Kapitel 7): Aufruf eines Nutzerprofils durch **http://localhost:3000/users/2352**
- **app.param**: Middleware, welche Route-Parameter mit einem Callback verbindet
- Hier: Callback holt Nutzerprofil und stellt sie den im **req**-Objekt nachgelagerten Middleware-Funktionen zur Verfügung

8.6 MONGOOSE

CRUD - AKTUALISIEREN VON DOKUMENTEN

app/controllers

```
users.server.controller.js
1 var User = require('mongoose').model('User');
2
3 > exports.create = function(req, res, next) { ...
12 };
13 > exports.list = function(req, res, next) { ...
16 };
17 > exports.userById = function(req, res, next, id) { ...
26 };
27 > exports.read = function(req, res) { ...
29 };
30 exports.update = function(req, res, next) {
31   User.findByIdAndUpdate(req.user.id, req.body, function(err, user) {
32     if (err) {
33       return next(err);
34     } else {
35       res.json(user);
36     }
37   });
38 };
```

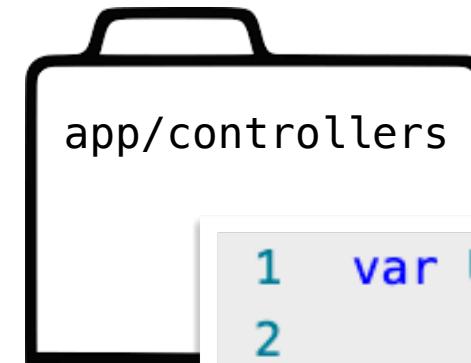
app/routes

```
users.server.routes.js
1 var users = require('../app/controllers/users.server.controller');
2
3 module.exports = function(app) {
4   app.route('/users')
5     .post(users.create)
6     .get(users.list);
7
8   app.param('userId', users.userById);
9
10  app.route('users/:userId')
11    .get(users.read)
12    .put(users.update);
13};
```

- **findByIdAndUpdate(query, [update], [options], [callback])**
 - **query**: Selektionskriterium
 - **[update]**: zu aktualisierende Felder
 - **[options]**: ggfs. Optionen
 - **[callback]**: Callback nach Beendigung der Operation
- Alternative Methoden: **update()**, **findOneAndUpdate()**

8.6 MONGOOSE

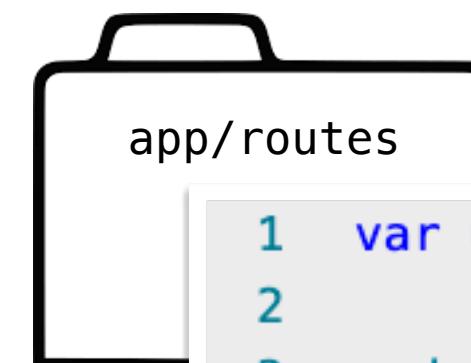
CRUD - ENTFERNEN VON DOKUMENTEN



app/controllers

```
users.server.controller.js
1 var User = ...
2
3 > exports.create = function(req, res, next) {
12 };
13 > exports.list = function(req, res, next) {
16 };
17 > exports.userById = function(req, res, next, id) {
26 };
27 > exports.read = function(req, res) {
29 };
30 > exports.update = function(req, res, next) {
38 };
39 < exports.delete = function(req, res, next) {
40   req.user.remove(function(err) {
41     if (err) {
42       return next(err);
43     } else {
44       res.json({message: 'Successfully deleted'});
45     }
46   });
47 };
```

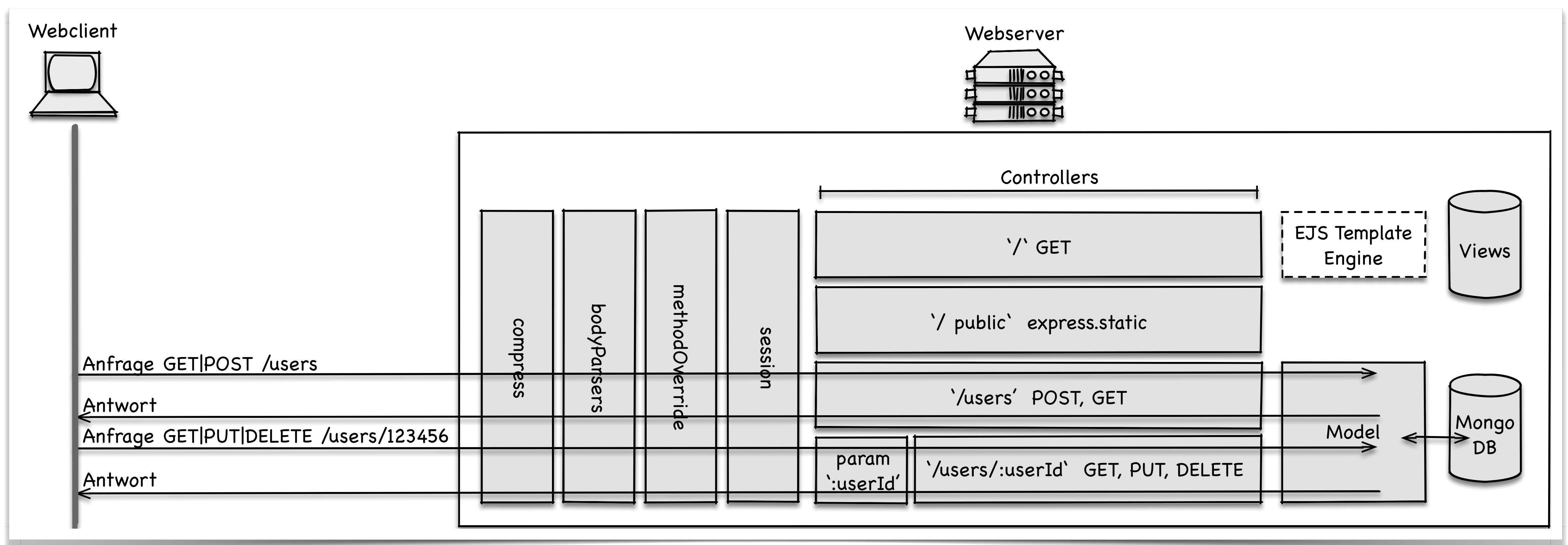
- Entfernen des Dokumentes auf dem remove aufgerufen wird
- Alternative Methoden zum Löschen: **findOneAndRemove()**, **findByIdAndRemove()**



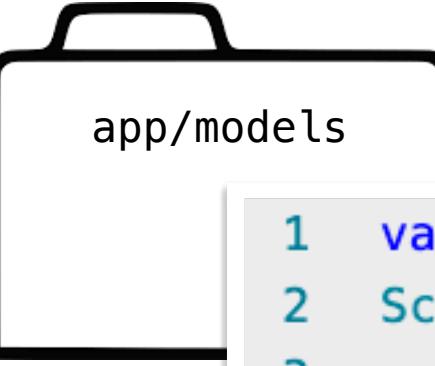
app/routes

```
users.server.routes.js
1 var users = ...
2
3 module.exports = function(app) {
4   app.route('/users')
5     .post(users.create)
6     .get(users.list);
7
8   app.param('userId', users.userById);
9
10  app.route('users/:userId')
11    .get(users.read)
12    .put(users.update)
13    .delete(users.delete);
14 };
```

8.6 MONGOOSE ZUSAMMENFASSUNG



8.6 MONGOOSE DEFAULT-WERTE



app/models

user.server.model.js

```
1 var mongoose = require('mongoose'),
2 Schema = mongoose.Schema;
3
4 var UserSchema = new Schema({
5   firstName: String,
6   lastName: String,
7   email: String,
8   username: String,
9   password: String,
10  created: {
11    type: Date,
12    default: Date.now
13  }
14});
15
16 mongoose.model('User', UserSchema);
```

- SchemaTypes: Parameter zur Spezifikation von Eigenschaften eines Dokumentfeldes
- Ermöglichen Manipulation von Daten
 - Automatisches Einfügen von Default-Werten
 - Validierung
 - Modifikation
 - Formatierung
- Vordefinierte SchemaTypes oder Erstellung eigener SchemaTypes

DEFINITION VON DEFAULT-WERTEN

- Automatische Belegung von Feldern mit Vorgabewerten beim Erstellen eines Dokumentes wenn kein expliziter Wert für das Feld angegeben wurde
- Festlegung des Default-Wertes mittels **default**-Option im SchemaType

8.6 MONGOOSE

MAßGESCHNEIDERTE MODIFIERS (I)

app/models

user.server.model.js

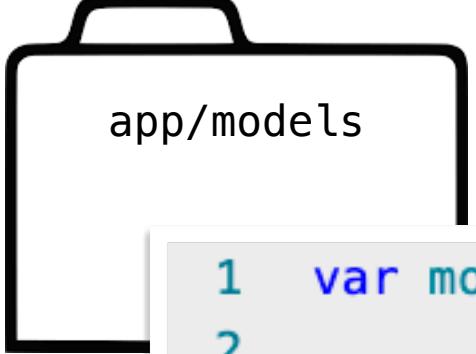
```
1 var mongoose = require('mongoose'), Schema = mongoose.Schema;
2
3 var UserSchema = new Schema({
4   firstName: String,
5   lastName: String,
6   email: String,
7   username: String,
8   password: String,
9   website: {
10     type: String,
11     set: function(url) {
12       if (!url) {
13         return url;
14       } else {
15         if (url.indexOf('http://') !== 0 && url.indexOf('https://') !== 0) {
16           url = 'http://' + url;
17         }
18         return url;
19       }
20     }
21   },
22   created: ...
23 });
24
25 mongoose.model('User', UserSchema);
```

MAßGESCHNEIDERTE SETTER-MODIFIER

- Implementierung eigener Modifier zur Manipulation von Daten vor dem Schreiben in die Datenbank
- Beispiel: Überprüft ob durch Nutzer eingegebene URL einer Website das Präfix **http://** oder **https://** enthält und ergänzt **http://** falls nicht



8.6 MONGOOSE MAßGESCHNEIDERTE MODIFIERS (II)



app/models

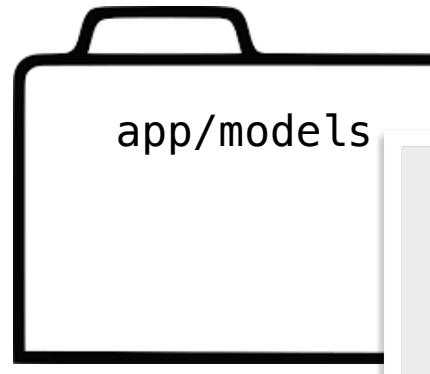
user.server.model.js

```
1 var mongoose = require('mongoose'), Schema = mongoose.Schema;
2
3 var UserSchema = new Schema({
4   firstName: String,
5   lastName: String,
6   email: String,
7   username: String,
8   password: String,
9 >   website: { ...
10    },
11   creditCardNumber: {
12     type: String,
13     get: function(cc) {
14       return '*****-****-' + cc.slice(cc.length-4, cc.length);
15     }
16   },
17   created: { ...
18    }
19  });
20  UserSchema.set('toJSON', {getters: true});
21  mongoose.model('User', UserSchema);
```

MAßGESCHNEIDERTE GETTER-MODIFIER

- Implementierung eigener Modifier zur Manipulation von Daten nach dem Lesen aus der Datenbank und vor der Auslieferung an den Client
- Beispiel: Maskiert aus Sicherheitsgründen die ersten 12 Stellen einer Kreditkartennummer vor Auslieferung des Nutzerprofils
- Beachte: Aufruf der Methode **UserSchema.set()** im Beispiel erzwingt bei der Konvertierung in das JSON-Format (z.B. mittels **res.json()**) die Verwendung der getter-Methode (wird andernfalls ignoriert)

8.6 MONGOOSE VALIDIERUNG (I)



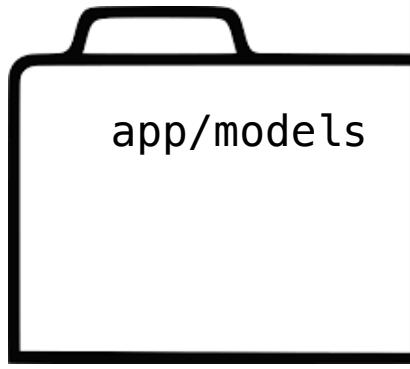
app/models

```
user.server.model.js
1 var mongoose = require('mongoose'), Schema = mongoose.Schema;
2
3 var UserSchema = new Schema({
4   firstName: String,
5   lastName: String,
6   email: {
7     type: String,
8     match: /.+\@.+\..+/
9   },
10  username: {
11    type: String,
12    trim: true,
13    unique: true,
14    required: true
15  },
16  password: String,
17  website: ...,
18  creditCardNumber: ...,
19  created: ...
20 });
21 UserSchema.set('toJSON', {getters: true});
22 mongoose.model('User', UserSchema);
```

VALIDIERUNG

- Überprüfung von Daten auf verschiedene Kriterien vor dem Schreiben in die Datenbank
- **unique**: überprüft und erfordert Eindeutigkeit des Wertes im betreffenden Feld der jeweiligen Kollektion
- **required**: erfordert Wertbelegung des Feldes als Voraussetzung für das Speichern eines neuen Dokumentes
- Beim Fehlschlagen der Validierung wird ein Fehler an die aufrufende Callback-Funktion zurückgegeben

8.6 MONGOOSE VALIDIERUNG (II)



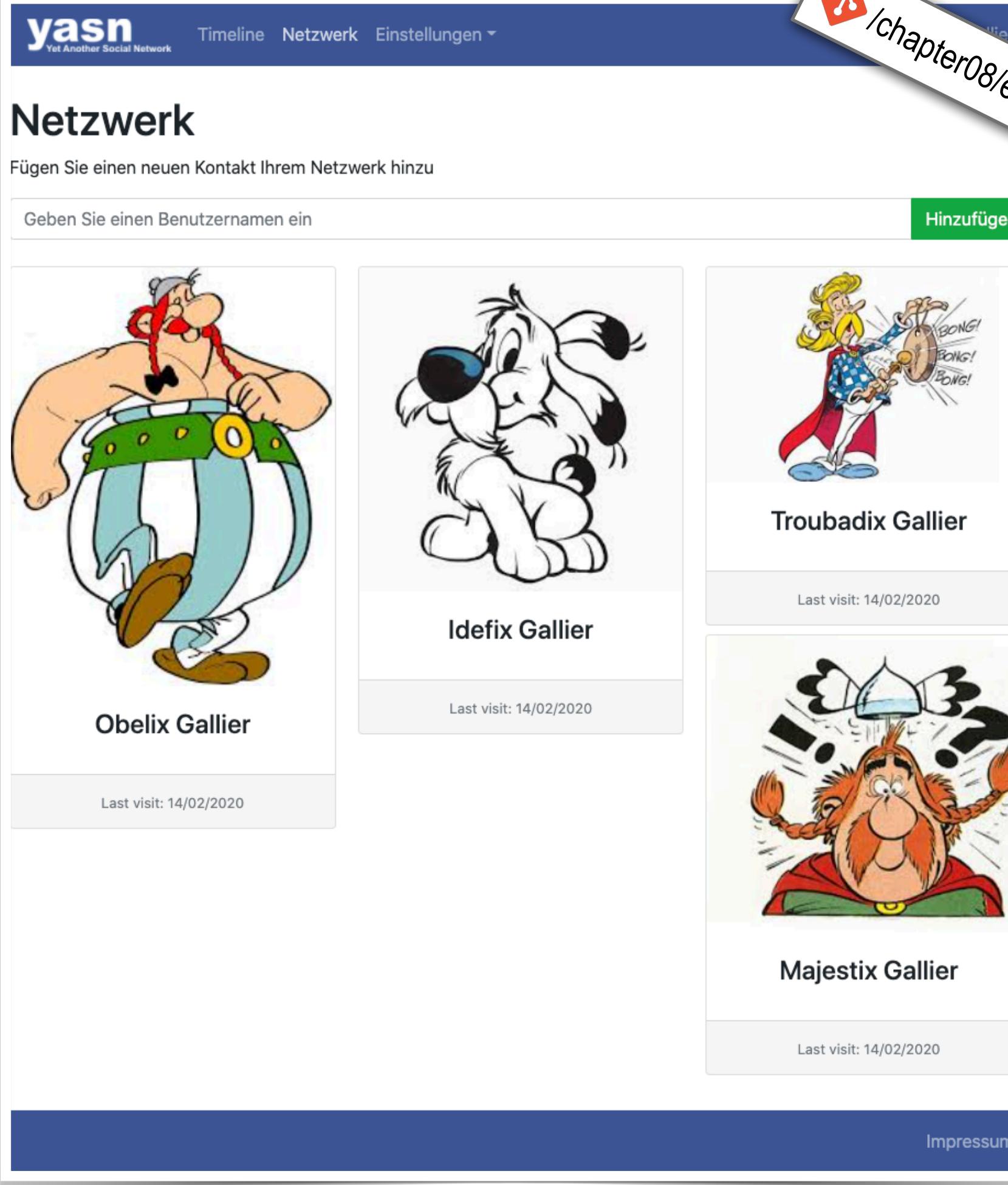
```
1 var mongoose = require('mongoose'), Schema = mongoose.Schema;
2
3 var UserSchema = new Schema({
4   firstName: String,
5   lastName: String,
6   email: ...,
7   },
8   username: ...,
9   },
10  },
11  password: {
12    type: String,
13    validate: [
14      function(password) {
15        return password.length >= 6;
16      },
17      'Password should be longer'
18    ]
19  },
20  website: ...,
21  },
22  creditCardNumber: ...,
23  },
24  created: ...,
25  }
26 });
27 UserSchema.set('toJSON', {getters: true});
28 mongoose.model('User', UserSchema);
```

MAßGESCHNEIDERTE VALIDATOREN

- Implementierung eigener Funktionen zur Überprüfung von Werten mittels **validate**

8.6 MONGOOSE

YASN - FINALES BEISPIEL



The screenshot shows the YASN (Yet Another Social Network) interface. At the top, there's a navigation bar with 'yasn' logo, 'Timeline', 'Netzwerk', and 'Einstellungen'. A red banner across the top says 'Ichapter08/example05/'. Below the banner, the page title is 'Netzwerk' with the sub-instruction 'Fügen Sie einen neuen Kontakt Ihrem Netzwerk hinzu'. There's a text input field 'Geben Sie einen Benutzernamen ein' and a green 'Hinzufügen' button. Below this, four user profiles are listed: 'Obelix Gallier' (with a pic of Obelix), 'Idefix Gallier' (with a pic of Idefix), 'Troubadix Gallier' (with a pic of a character in a hat), and 'Majestix Gallier' (with a pic of a character in a crown). Each profile has a 'Last visit: 14/02/2020' message below it. At the bottom right of the screenshot, there's a link 'Impressum'.

```
1 var mongoose=require('mongoose'), Schema=mongoose.Schema;
2
3
4 var userSchema=new Schema({
5   title: String,
6   firstname: String,
7   lastname: String,
8   email: String,
9   sex: String,
10  dateOfBrith: Date,
11  username: String,
12  password: String,
13  thumbnail: String,
14  contacts: [
15    {
16      firstname: String,
17      lastname: String,
18      username: String,
19      thumbnail: String
20    }
21  ];
22 module.exports=mongoose.model('User', userSchema);
```



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