

Rest-Implementierung: Express

Web-Engineering II

Prof. Dr. Sebastian von Klinski

Express

- Verbreitetes Web-Application-Framework für Node.js zur Umsetzung von Web-Anwendungen und Web-Services
- Die Umsetzung basiert auf einem einfachen Funktionsprinzip
 - Client-Anfragen werden auf Basis von Weiterleitungsregeln einem oder mehreren Request-Handlern zugewiesen.
 - Der Request-Handler bekommt das Request-Objekt und kann über das Response-Objekt die Antwort an den Client schicken.
 - Über Middleware-Funktionen kann die Beantwortung von Anfragen vorverarbeitet und modifiziert werden.
- Über integrierte Template-Engines können Ergebnisse in HTML oder andere Formate transformiert werden.
- HTTP wird als Message-Protokoll verwendet

Quelle: <https://de.wikipedia.org/wiki/Express.js>

Express

- Die Implementierung der Anwendung erfolgt in erster Linie durch die Umsetzung von Weiterleitungsroutinen
- Wesentliche Punkte sind dabei
 - Für welche URLs soll die Weiterleitungsroutine gelten?
 - Was muss vor der Bearbeitung der Anfrage überprüft oder ausgeführt werden?
 - Welche Antwort soll zurückgegeben werden?
- Ähnlich wie Node.js ist Express ein eher kleines Modul
- Es gibt jedoch viele Erweiterungen/ Handler, die die Umsetzung von Anwendung deutlich erleichtert und beschleunigt
 - Cookies, Sessions, User Logins, URL Parameters, POST Data, Security Headers, etc.

Quelle: <https://de.wikipedia.org/wiki/Express.js>

Beispiel für Web-Server

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

```
const app = express()
```

```
app.get('/', function (request, response) {  
  response.send('Hello World')  
})
```

```
const server = app.listen(3000, function () {  
  const host = server.address().address  
  const port = server.address().port  
  console.info('Example app listening at `http://${host}:${port}`')  
})
```

Route Handler

Port

Callback, nach dem Start des Servers

Standardadresse: <http://localhost:3000/>

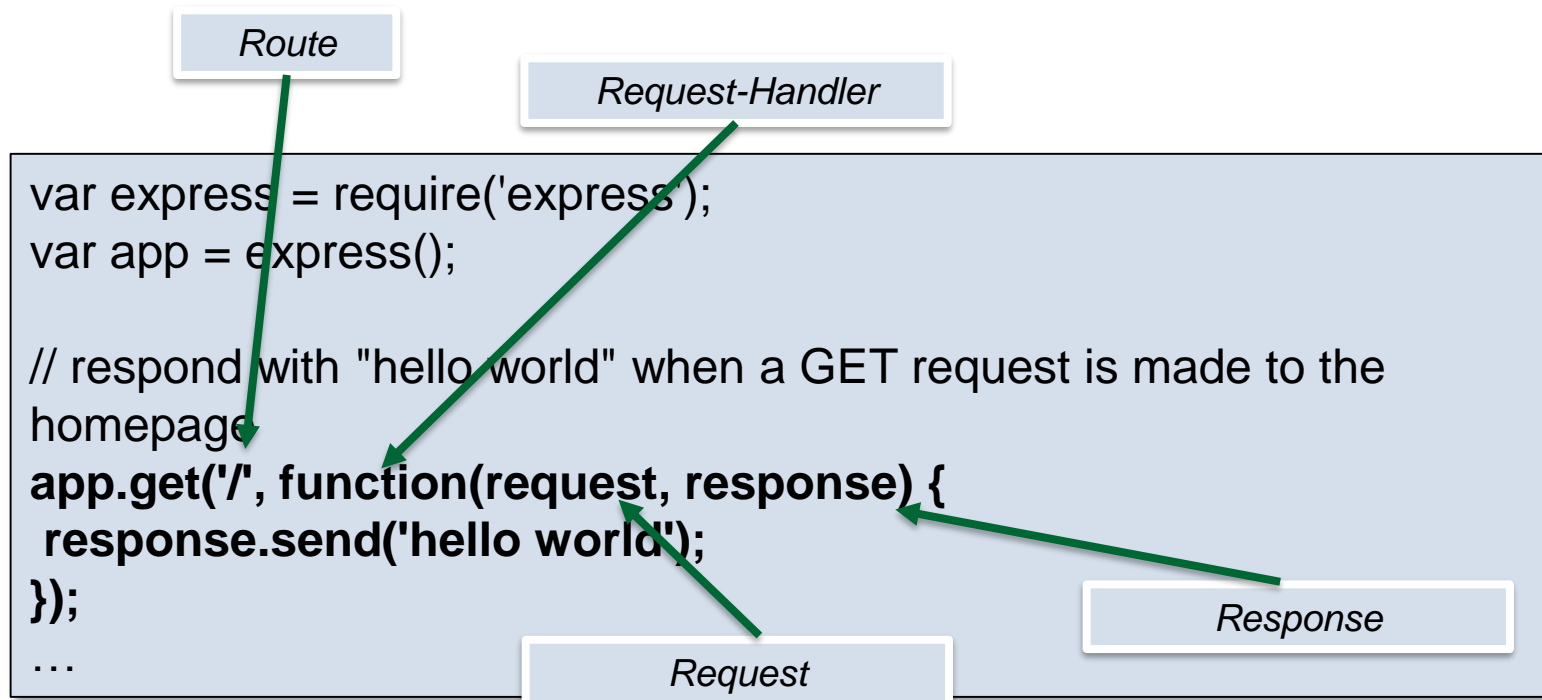
Weiterleitungen

- Es können Weiterleitungen für die gängigen HTTP-Methoden erstellt werden
 - GET: `app.get()`,
 - POST: `app.post()`,
 - DELETE: `app.delete()` und
 - PUT: `app.put()`
- Mit `all()` wird der Route-Handler für alle Anfragen mit dieser URL angewandt
- Eine Weiterleitung ist stets eine Kombination von Pfad und Callback-Methode (Route-Handler)

```
var app = require("express")();  
  
app.get("/", function(httpRequest, httpResponse, next){  
    httpResponse.send("Hello World!!!!");  
});  
  
app.listen(8080);
```

Weiterleitungen

- Der Route-Handler bekommt eine Referenz auf den HTTP-Request und die HTTP-Response



Quelle: <https://code.visualstudio.com/docs/nodejs/nodejs-tutorial>

Weiterleitungen

- Es kann für jede HTTP-Methode individuelle Request-Handler registriert werden

```
...  
app.get('/', function(req, res) {  
  res.send('hello world');  
});  
app.post('/', function (req, res) {  
  res.send('POST request to the homepage');  
});  
app.put('/', (req, res) => {  
  res.send('Received a PUT HTTP method');  
});  
app.delete('/', (req, res) => {  
  res.send('Received a DELETE HTTP method');  
});
```

Quelle: <https://code.visualstudio.com/docs/nodejs/nodejs-tutorial>

Weiterleitungen

- Mit den HTTP-Methoden können die unterschiedliche Serverarten umgesetzt werden
- Web-Server
 - GET und POST
- REST-Server
 - GET für Abrufen von Daten
 - POST für Anlegen von Daten
 - PUT für Update von existierenden Daten
 - DELETE zum Löschen von Daten
 - PATCH zum partiellen Aktualisieren von Daten (wenig gebräuchlich)

Quelle: <https://code.visualstudio.com/docs/nodejs/nodejs-tutorial>

Request-Objekt

- Über das Request-Objekt kann der Request-Handler Informationen zur Anfragen bekommen
- Zum Abrufen, Editieren und Löschen werden bei Rest-Services die IDs häufig über die URL mitgegeben
- Beispiel:
 - Aktualisieren von User wird über PUT mit der User-ID als Parameter umgesetzt
 - In den Request-Handler können die IDs aus den Request-Parametern übernommen werden

```
app.put('/users/:userId', (req, res) => {  
  return res.send(  
    `PUT HTTP method on user/${req.params.userId} resource`,  
  );  
});
```

Response-Objekt

- Über das Response-Objekt sendet der Request-Handler die Antwort an den Client
- Response-Objekt hat mehrere Methode, die genutzt werden können
 - `render`: umsetzen der Web-Seite mit der Template-Ending
 - `send`: direktes Senden der Antwort ohne ein Rendering

```
var express = require('express');
var router = express.Router();

// GET home page mit Seiten-Template. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

// Ohne Seiten-Template
router.get('/abc', function(req, res, next) {
  res.send('noch eine anfrage');
});
module.exports = router;
```

JSON-REST-Services

- JSON-Objekte werden direkt per „send“ zurückgesendet
- Beispiel: hart kodiert werden zwei JSON-Objekte zurückgegeben

```
let users = {
  1: {
    id: '1',
    username: 'Robin Wieruch',
  },
  2: {
    id: '2',
    username: 'Dave Davids',
  },
};

/* GET users listing. */
router.get('/', function(req, res, next) {
  res.send(Object.values(users));
});
```

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 75
ETag: W/"4b-bTKluFungUuVDp3crGpmyZ+VKIE"
Date: Fri, 28 Feb 2020 15:02:54 GMT
Connection: close
[
  {
    "id": "1",
    "username": "Robin Wieruch"
  },
  {
    "id": "2",
    "username": "Dave Davids"
  }
]
```

Quelle: <https://www.robinwieruch.de/node-express-server-rest-api>

Response-Objektmethoden

- Der Route-Handler kann die folgenden Methoden des Response-Objekts nutzen

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

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

Request Handler

- Zuweilen sollen Verkettungen von Request-Handlern angewandt werden
- Beispiel
 - Bevor auf einen geschützten Bereich zugegriffen werden kann, soll sich der Benutzer authentifizieren
 - Falls der Benutzer schon eingeloggt ist, leitet der Request-Handler die Anfrage einfach an den nächsten Request-Handler weiter
 - Falls das Login scheitert, wird er auf die öffentlichen Seiten weitergeleitet
- Zur Umsetzung solcher vorschalteten Prüfungen können Request-Handler in „Pipelines“ verkettet werden

```
app.get('/adminArea', authentication(request, response, next),  
showAdminArea(request, response))
```



*Verkettung von
Request-Handlern*

Verkettungen von Weiterleitungen

- Route-Handler können als Arrays von Route-Handlern verkettet werden

```
app.get('/example/b',
  function (req, res, next) {
    console.log('the response will be sent by the next function ...');
    next();
  },
  function (req, res) {
    res.send('Hello from B!');
  });
//Oder
var cb0 = function (req, res, next) {
  console.log('CB0');
  next();
}
...
app.get('/example/c', [cb0, cb1, cb2]);
```

Verkettungen von Weiterleitungen

- Es können mehrere Route-Handler für den gleichen Pfad registriert werden
- Die Route-Handler werden dann in der Reihenfolge ausgeführt, in der sie angelegt wurden.
- Sie bekommen neben Request und Response auch eine Referenz auf den nächsten Request-Handler (next)
- Bei Verkettungen von Weiterleiten muss beachtet werden, dass der Route-Handler entweder mit next() oder mit einer Antwort endet

```
var app = require("express")();  
app.get("/", function(httpRequest, httpResponse, next){  
    httpResponse.write("Hello");  
    next();  
});  
app.get("/", function(httpRequest, httpResponse, next){  
    httpResponse.write(" World !!!");  
    httpResponse.end();  
});  
app.listen(8080);
```

Express: `express.Router`

- Mit der Klasse `express.Router` lassen sich modulare Gruppen von Routenhandler erstellen.
- Eine Router-Instanz ist ein vollständiges Middleware- und Routingsystem.
- Aus diesem Grund wird diese Instanz oft auch als “Mini-App” bezeichnet.
- In der Regel werden Router für funktional zusammengehörige Bereiche angelegt
- Beispielsweise
 - Alle Methoden für das Verwalten von Usern
 - Alle Methoden die zu einer URL gehören (GET, POST, etc.)
- Die Router werden in der Regel in einer separaten Datei angelegt

Express: `express.Router`

- Anlegen von Router in einer separaten Datei

```
var express = require('express');  
var router = express.Router();  
  
// middleware that is specific to this router  
router.use(function timeLog(req, res, next) {  
  console.log('Time: ', Date.now());  
  next();  
});  
  
// define the home page route  
router.get('/', function(req, res) {  
  res.send('Birds home page');  
});  
  
// define the about route  
router.get('/about', function(req, res) {  
  res.send('About birds');  
});  
  
module.exports = router;
```

express.Router

- Router werden mit `app.use()` angelegt
- Laden des Routers aus der Datei `birds.js`
- Die Request-Handler des Routers können dann zu einer beliebigen URL gelinkt werden

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

Middleware

Middleware: Motivation

```
app.get("/dashboard", function(httpRequest, httpResponse, next){  
  logRequest();  
  if(checkLogin()){  
    {  
      httpResponse.send("This is the dashboard page");  
    }  
  }  
  else  
  {  
    httpResponse.send("You are not logged in!!!");  
  }  
});  
  
app.get("/profile", function(httpRequest, httpResponse, next){  
  logRequest();  
  if(checkLogin()){  
    {  
      httpResponse.send("This is the dashboard page");  
    }  
  }  
  else  
  {  
    httpResponse.send("You are not logged in!!!");  
  }  
});
```

*Wiederkehrende Log-
Funktionen und
Prüfungen*

Middleware: Motivation

```
app.get("/*", function(httpRequest, httpResponse, next){  
  logRequest();  
  next();  
})
```

```
app.get("/*", function(httpRequest, httpResponse, next){  
  if(checkLogin())  
  {  
    next();  
  }  
  else  
  {  
    httpResponse.send("You are not logged in!!!");  
  }  
})
```

```
app.get("/dashboard", function(httpRequest, httpResponse, next){  
  httpResponse.send("This is the dashboard page");  
});
```

*Einmalige
Registrierung der Log-
Funktionen und der
Prüfungen für alle
URLs*

*Der eigentliche
Request-Handler weiß
nicht von den
Middleware-
Funktionen*

Middleware

- Middleware-Routinen setzen übergreifende Funktionen um, die vor der Ausführung des Route-Handlers zur Anwendung kommen
- Begriff Middleware aus meiner Sicht falsch verwendet
- Definition von Middleware

„Software für den Datenaustausch zwischen Anwendungsprogrammen, die unter verschiedenen Betriebssystemen oder in heterogenen Netzen arbeiten.“

- Middleware in Express

„Middlewarefunktionen sind Funktionen, die Zugriff auf das Anforderungsobjekt (req), das Antwortobjekt (res) und die nächste Middlewarefunktion im Anforderung/Antwort-Zyklus der Anwendung haben. Die nächste Middlewarefunktion wird im Allgemeinen durch die Variable next bezeichnet.“

Middleware

Middleware-Funktionen...

1. bekommen Anfragen vom Router
 2. geben diese in definierte Reihenfolge an die Request-Handler weiter und
 3. können auch selber Änderungen am Request vornehmen.
- Gängige Aufgaben von Middleware-Funktionen
 - Ausführen von Code
 - Überprüfung von Cookies, Authentifizierung, etc.
 - Vornehmen von Änderungen an der Anforderung und an Antwortobjekten
 - Anreichern von Daten (z.B. Session-Informationen)
 - Beenden des Anforderung/Antwort-Zyklus
 - Aufrufen der nächsten Middleware-Funktion im Stack
 - Zentrale Komponente zum Steuern des Arbeitsflusses beim Bearbeiten von Anfragen

Middleware

- Syntaktisch sind Middleware-Funktionen fast identisch wie Weiterleitungen
- Üblicherweise werden sie jedoch auf alle Requests für bestimmte Pfade angewandt
- Wichtige Vorteile
 - Middleware-Funktionen können jederzeit auch nachträglich hinzugefügt werden
 - Sie können schnell ein- und wieder ausgeschaltet werden
- Middleware-Funktionen werden in der Reihenfolge der Deklaration angewandt
- Wenn die Anfrage nicht über die Middleware-Funktion beendet wird, sollte in jedem Fall `next()` aufgerufen werden, da sonst die Anforderung in den Status „blockiert“ übergeht
- Für die Definition von Middleware-Routinen gibt es **`app.use()`**
- Theoretisch kann auch `app.all()` verwendet werden (leichter Unterschied)

Middleware

- App.use() und app.all() sind ähnlich, jedoch nicht identisch

```
app.use( "/product" , mymiddleware);  
// will match /product  
// will match /product/cool  
// will match /product/foo  
  
app.all( "/product" , handler);  
// will match /product  
// won't match /product/cool  <-- important  
// won't match /product/foo   <-- important  
  
app.all( "/product/*" , handler);  
// won't match /product      <-- Important  
// will match /product/cool  
// will match /product/foo
```

Quelle: <http://qanimate.com/express-js-middleware-tutorial/>

Middleware

- Gängige Middleware-Funktionen
 - Authentifizierung: wenn der User nicht eingeloggt ist, wird er auf eine öffentliche Seite geleitet, ansonsten auf die persönliche Seite
 - Logging: alle Anfragen gehen durch die Logging-Middleware, um beispielsweise Zugriffszeiten oder Abfragehäufigkeiten zu erfassen
 - Fehlerbehandlung: ein zentraler Fehler-Handler kann auf erwartete Fehler definierte Aktionen auslösen (z.B. Mail an Administrator senden)
- Beispiel:
 - Bei ersten Aufruf wird Cookie gesetzt
 - Bei allen weiteren Aufrufen wird Cookie ausgewertet
 - Verwendung des Moduls cookie-parser

```
var app = express();  
var cookieParser = require('cookie-parser');  
// load the cookie-parsing middleware  
app.use(cookieParser());
```

Express-Projekt anlegen

Projekt aufsetzen

- In Node.js gibt es für verschiedene Projektarten Projektgeneratoren
- Sie legen die Standardprojektstruktur an und erstellen gängige Dateien
- Für Express gibt es den express-generator
- Installation über

```
npm install -g express-generator
```

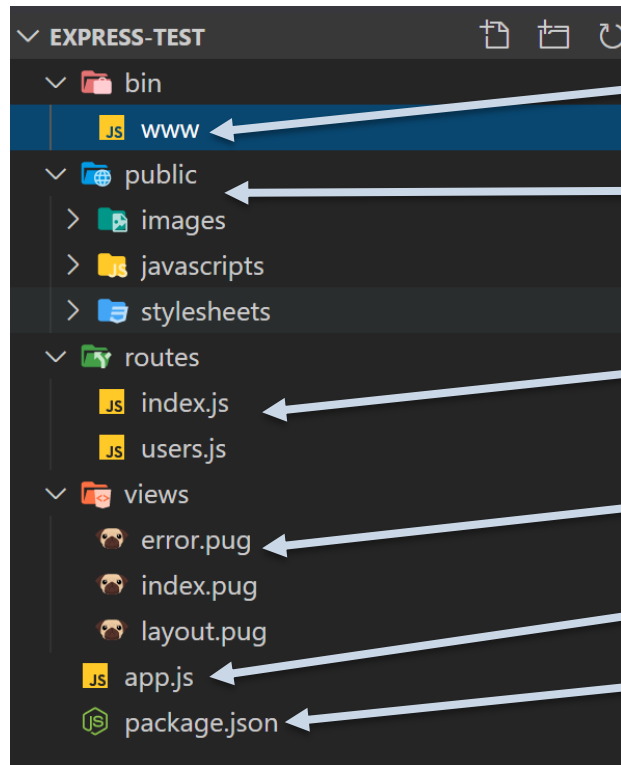
- install -g: installiert das Paket als globales Paket
- Es muss nicht mehr bei anderen Projekten installiert werden

Quelle: <https://code.visualstudio.com/docs/nodejs/nodejs-tutorial>

Projekt aufsetzen

Anlegen eines Projektes im Projektverzeichnis (-view pug kann weggelassen werden)

```
express [Projektverzeichnis] --view pug
```



JavaScript-Datei zum Starten des Servers

Ort für Stylesheets, Bilder, etc.

Routen für die jeweiligen URLs

Web-Seiten für Template Engine pug

App.js ist gestartete Anwendung

Konfigurationsdatei mit Abhängigkeiten

Quelle: <https://code.visualstudio.com/docs/nodejs/nodejs-tutorial>

Projekt aufsetzen

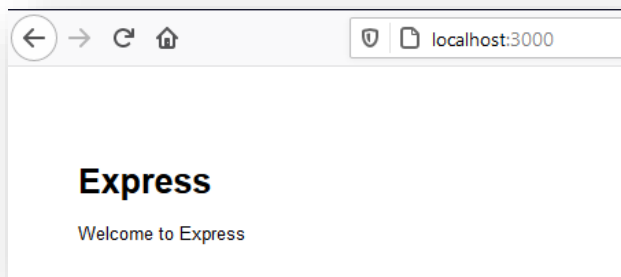
- Damit Projekt gestartet werden kann, müssen zunächst Abhängigkeiten aufgelöst werden (Node-Module geladen werden)

```
npm install
```

- Anschließend kann Express-Anwendung gestartet werden

```
npm start
```

- Der Server läuft auf Port 3000, wo eine Dummy-Seite angezeigt wird



Quelle: <https://code.visualstudio.com/docs/nodejs/nodejs-tutorial>

Projekt aufsetzen: Hinweis

- Nach Änderungen im Code müsste theoretisch jedes mal der Server neu gestartet werden
- Mit dem Modul nodemon kann das vermieden werden
 - Entwicklungsmodul für Node.js, das automatisch Änderungen am Code feststellt und dann Node.js neu startet
 - Für Entwicklung sehr hilfreich!
 - Installation: `npm install -g nodemon`
 - Anschließend wird die Anwendung nicht mehr mit „npm start“, sondern „nodemon“ gestartet

```
PS C:\Beuth-Temp\FirstSteps-0-Node> nodemon
[nodemon] 2.0.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node ./bin/www`
[nodemon] restarting due to changes...
[nodemon] starting `node ./bin/www`
GET / 200 237.336 ms - 188
GET /stylesheets/style.css 304 1.823 ms - -
□
```

package.json

- Zentrale Konfigurationsdatei für das Modul

```
{
  "name": "express-test",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "cookie-parser": "~1.4.4",
    "debug": "~2.6.9",
    "express": "~4.16.1",
    "http-errors": "~1.6.3",
    "morgan": "~1.9.1",
    "pug": "2.0.0-beta11"
  }
}
```

Modulbeschreibung

*Abhängigkeiten, werden automatisch
installiert, wenn „npm install“ aufgerufen wird*

pug

- Pug ist eine Template-Engine
- Sie erzeugt aus kurzen Template-Angaben HTML-Seiten
- Sie wird durch entsprechenden Router parametrisiert und Stylesheet formatiert

```
app.set('view engine', 'pug');    // Pug-Engine wird aktiviert
```

index.pug

```
extends layout
```

```
block content
```

```
h1= title
```

```
p Welcome to #{title}
```

index.js

```
var express = require('express');  
var router = express.Router();
```

```
/* GET home page. */
```

```
router.get('/', function(req, res, next) {  
  res.render('index', { title: 'Express' });  
});
```

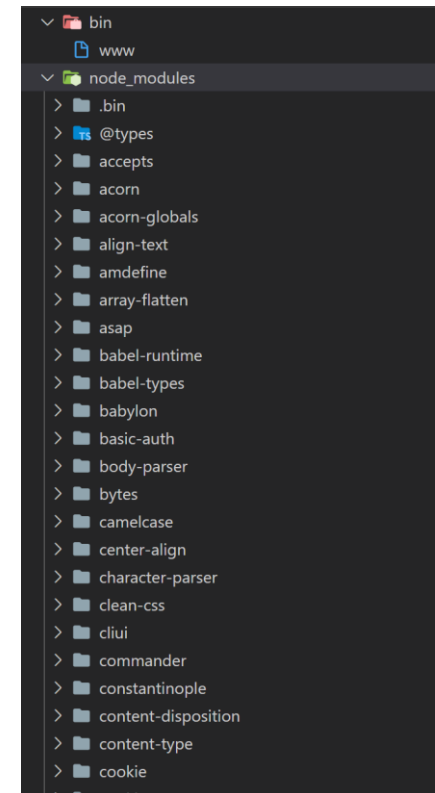
```
module.exports = router;
```

Projekt aufsetzen

- Vor dem Starten müssen zunächst alle Abhängigkeiten aufgelöst werden (Module aus dem Repository von npm kopieren)

```
npm install
```

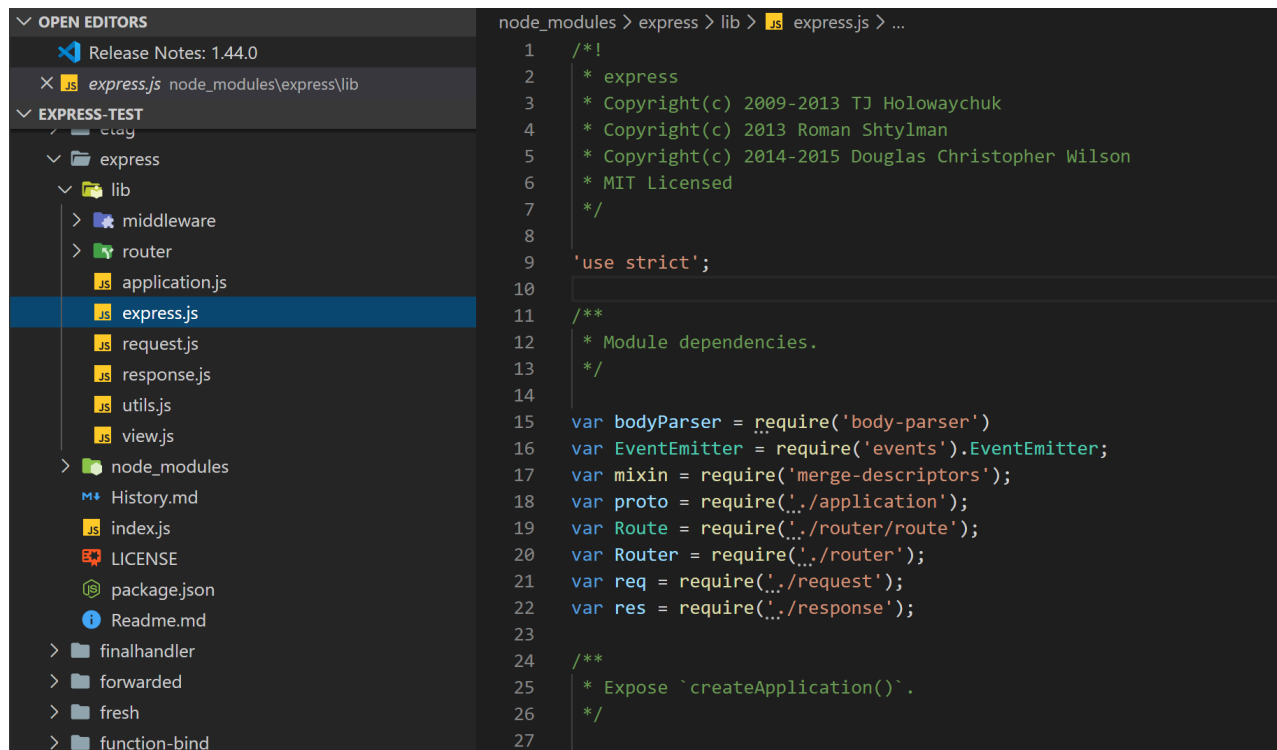
- Installierte Module werden in das Verzeichnis „node_modules“ kopiert
- Mehr Module, als im package.json aufgeführt sind, weil auch die Abhängigkeiten der Module aufgelöst und kopiert werden



Quelle: <https://code.visualstudio.com/docs/nodejs/nodejs-tutorial>

Projekt aufsetzen

- Module sind häufig mit Quelltext, Lizenzangaben und anderem versehen
- Es lohnt, sich mal reinzuschauen!



The screenshot shows the Visual Studio Code interface. On the left, the 'EXPRESSION-TEST' folder is expanded, showing the 'lib' subfolder. The 'express.js' file is selected and highlighted. The main editor area displays the source code of 'express.js'. The code includes a header with copyright information and a license, followed by a 'use strict' directive and a series of 'require' statements for various modules like 'body-parser', 'events', 'merge-descriptors', 'application', 'router', 'request', and 'response'.

```
node_modules > express > lib > express.js > ...
1  /*!
2   * express
3   * Copyright(c) 2009-2013 TJ Holowaychuk
4   * Copyright(c) 2013 Roman Shtylman
5   * Copyright(c) 2014-2015 Douglas Christopher Wilson
6   * MIT Licensed
7   */
8
9  'use strict';
10
11 /**
12  * Module dependencies.
13  */
14
15 var bodyParser = require('body-parser')
16 var EventEmitter = require('events').EventEmitter;
17 var mixin = require('merge-descriptors');
18 var proto = require('./application');
19 var Route = require('./router/route');
20 var Router = require('./router');
21 var req = require('./request');
22 var res = require('./response');
23
24 /**
25  * Expose `createApplication()`.
26  */
27
```

Quelle: <https://code.visualstudio.com/docs/nodejs/nodejs-tutorial>

Projekt aufsetzen

- Für den Start des Projektes kann npm start aufgerufen werden

```
npm start
```

- Npm kann Skripte ausführen, die im package.json definiert sind
- In diesem Fall wird „node ./bin/www“ ausgeführt

```
{
  "name": "express-test",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    ...
  }
}
```

Quelle: <https://code.visualstudio.com/docs/nodejs/nodejs-tutorial>

Projekt aufsetzen

- Startskript für Standard-Express-Anwendung

```
// Start-Script in bin/www
...
var port = normalizePort(process.env.PORT || '3000');
app.set('port', port);
var server = http.createServer(app);
server.listen(port);
...
```

- Standardadresse: <http://localhost:3000/>
- Kann aber im Skript geändert werden

Quelle: <https://code.visualstudio.com/docs/nodejs/nodejs-tutorial>

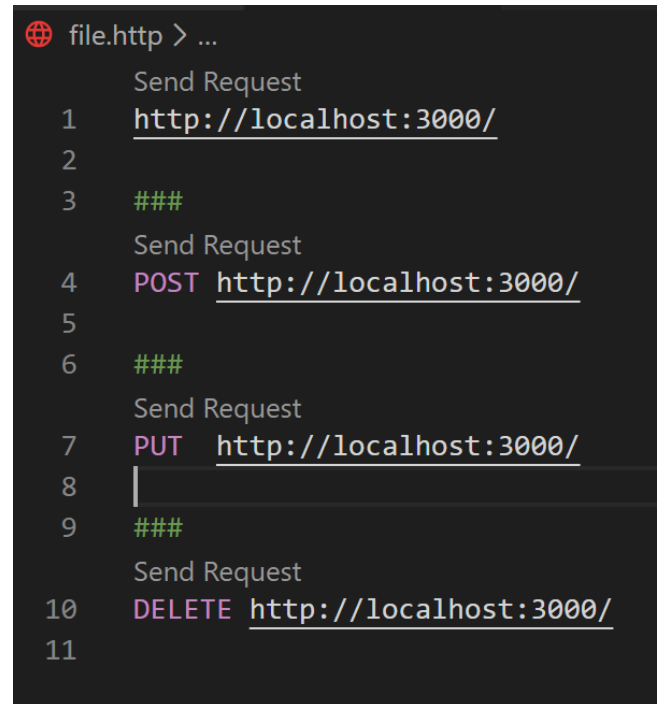
Testen der Anwendung

Testen von Routen

- Für das Testen von Routen wird ein Request-Client benötigt
 - GET-Requests können per Browser getestet werden
 - Für alle anderen wird ein Client benötigt, der die entsprechenden Requests erzeugt
- Möglicher Test-Client: Visual Studio Code Extension „Rest-Client“
 - Über das Extension-Fenster von Visual Studio Code installieren
 - Nach der Installation im Root-Verzeichnis eine Datei file.http anlegen
 - URLs eintragen
 - Anschließend per Mouse-Klick ausführen

Visual Studio Code: REST-Client

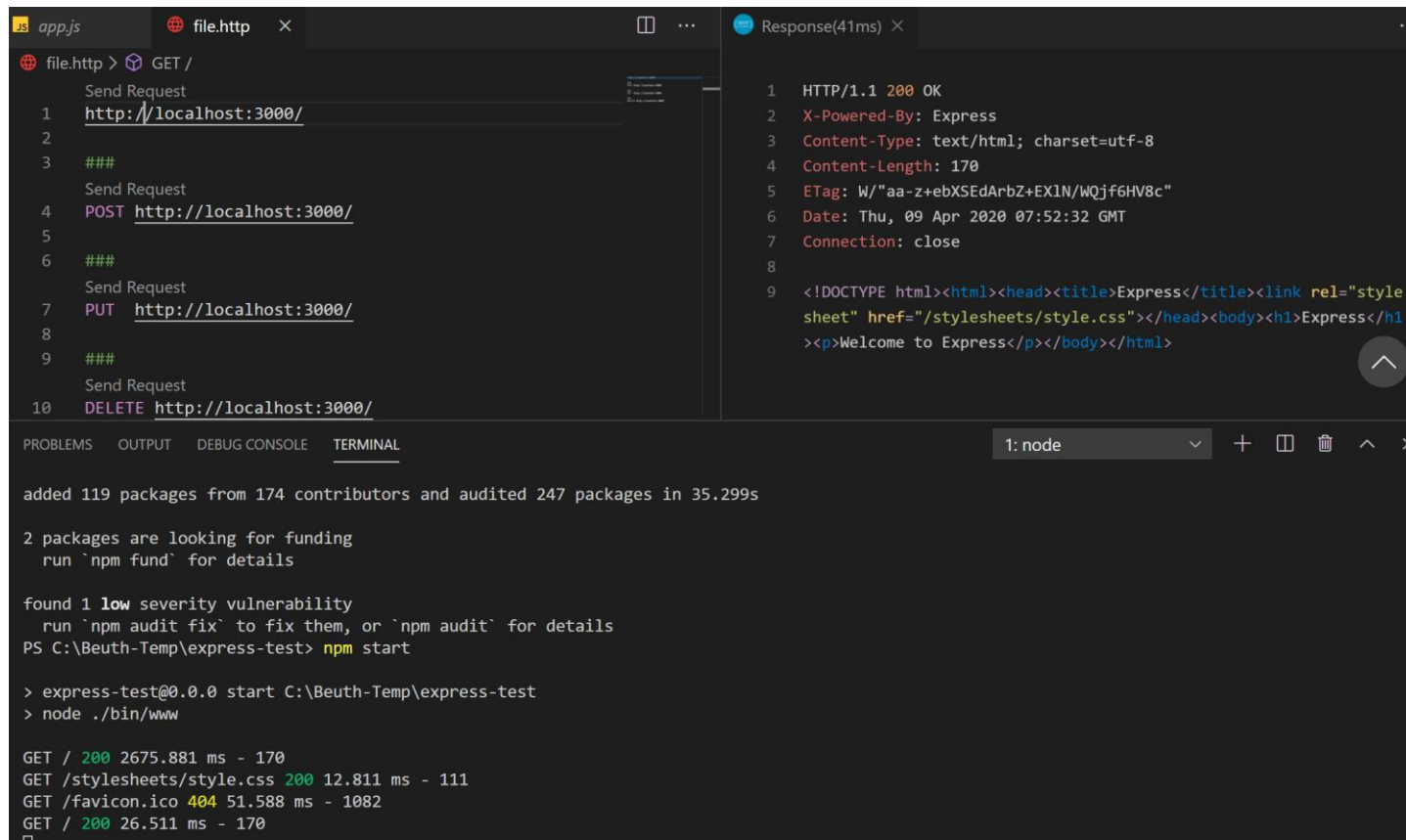
- REST-Client-Extension legt automatisch über dem Request den Button „Send Request“ an
- Ausführen des Requests entweder über den Button oder über das Kontext-Menü „Send Request“
- Es können auch Sequenzen von Requests über den Button ausgeführt werden
- Unterschiedliche Requests durch ### trennen



```
file.http > ...  
Send Request  
1 http://localhost:3000/  
2  
3 ###  
Send Request  
4 POST http://localhost:3000/  
5  
6 ###  
Send Request  
7 PUT http://localhost:3000/  
8  
9 ###  
Send Request  
10 DELETE http://localhost:3000/  
11
```


Visual Studio Code: REST-Client

- Ergebnis des Requests wird im separaten Fenster angezeigt



```
file.http > GET /
Send Request
1 http://localhost:3000/
2
3 ###
Send Request
4 POST http://localhost:3000/
5
6 ###
Send Request
7 PUT http://localhost:3000/
8
9 ###
Send Request
10 DELETE http://localhost:3000/

Response(41ms) X
1 HTTP/1.1 200 OK
2 X-Powered-By: Express
3 Content-Type: text/html; charset=utf-8
4 Content-Length: 170
5 ETag: W/"aa-z+ebXSEdArbZ+EX1N/WQjf6HV8c"
6 Date: Thu, 09 Apr 2020 07:52:32 GMT
7 Connection: close
8
9 <!DOCTYPE html><html><head><title>Express</title><link rel="style
sheet" href="/stylesheets/style.css"></head><body><h1>Express</h1
><p>Welcome to Express</p></body></html>

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: node
added 119 packages from 174 contributors and audited 247 packages in 35.299s
2 packages are looking for funding
  run `npm fund` for details
found 1 low severity vulnerability
  run `npm audit fix` to fix them, or `npm audit` for details
PS C:\Beuth-Temp\express-test> npm start
> express-test@0.0.0 start C:\Beuth-Temp\express-test
> node ./bin/www
GET / 200 2675.881 ms - 170
GET /stylesheets/style.css 200 12.811 ms - 111
GET /favicon.ico 404 51.588 ms - 1082
GET / 200 26.511 ms - 170
```

Testen von REST-Services

- Wenn über REST-Services das Anlegen und Ändern von Daten möglich ist, müssen auch Daten mitgeschickt werden
- Diese können wie bei einem normalen HTTP-Request an die URL angehängen werden
- Beispiel: Anlegen eines Users mit User-ID und User-Name

```
POST http://localhost:3000/users
Content-Type: application/json

{
  "userID": "admin",
  "userName": "Thomas Schreiber"
}
```

Verwendung von Routern

Router zur Modularisierung

- Verwendung von Routern erlaubt Modularisierung der Request-Handler
- Registrieren der Router und ihrer Request-Handler wird durch `app.use` vorgenommen

```
var indexRouter = require('./routes/index');  
...  
app.use('/', indexRouter);
```

- Das Registrieren der Router-Skripte wird in der Regel in der zentralen `app.js` vorgenommen
- Bei großen Projekten ist direkte Verlinken der Skriptdateien problematisch
 - Viele Entwickler müssen ggfls. immer wieder an der gleichen Datei arbeiten → Versionskonflikte
 - In der `app.js` ist die Projektstruktur fest „verdrahtet“
- Vorgänge beim Starten der zentralen Anwendung sollten möglichst modular ausgelagert werden

Registrieren der Routen in app.js

- Beispiel: die Skriptdateien index.js und users.js werden direkt in app.js verlinkt
- Bei großen Projekten mit vielen Routen keine gute Lösung

```
...  
var indexRouter = require('./routes/index');  
var usersRouter = require('./routes/users');  
...  
app.use('/', indexRouter);  
app.use('/users', usersRouter);  
...
```

Laden der Router

- Die Details zur Umsetzung der verschiedenen Routes sollte in einer separaten Datei (z.B. all-routes.js) ausgelagert werden
 - all-routes.js definiert die möglichen Routen
 - App.js lädt all-routes.js und registriert die notwendigen Routen

app.js

```
...  
var allRoutes = require('./routes/all-routes');  
...  
app.use('/', allRoutes.indexRoute);  
app.use('/users', allRoutes.usersRoute);  
...
```

- Änderungen in app.js müssen nur vorgenommen werden, wenn neue Routen hinzukommen oder entfernt werden sollen
- Es kann auch schneller zwischen Router-Gruppen gewechselt werden

Laden der Router

- Registrieren der individuelle Routen und Request-Handler

all-routes.js

```
var indexRoute = require('./index');
var usersRoute = require('./users');

module.exports = {
  indexRoute,
  usersRoute,
};
```

index.js

```
var express = require('express');
var router = express.Router();

router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});
module.exports = router;
```

Erster REST-Service

Ein erster REST-Service

- Bei einem REST-Service wird in der Regel
 - Anfrage an den Web-Server geschickt
 - Eine Antwort häufig mit Json-Body zurückgeschickt
- Viele andere Varianten auch möglich
- Schritte
 - Modul für Route anlegen
 - Route in der app.js registrieren

Ein erster REST-Service

- Express laden
- Von Express den Router holen
- Dummy-Daten definieren
- Beim Router die Route registrieren
- Bei der Callback-Funktion mit `res.json()` die Daten als Json-Antwort zurückgeben
- Den Router exportieren, da von anderem Modul geladen werden kann

```
1  var express = require('express');
2  var router = express.Router();
3
4  const users = [
5    "user1" : {
6      "name" : "mahesh",
7      "password" : "password1",
8      "profession" : "teacher",
9      "id": 1
10   },
11
12   "user2" : {
13     "name" : "suresh",
14     "password" : "password2",
15     "profession" : "librarian",
16     "id": 2
17   },
18
19   "user3" : {
20     "name" : "ramesh",
21     "password" : "password3",
22     "profession" : "clerk",
23     "id": 3
24   }
25 ]
26
27 router.get('/', function(req, res, next) {
28   res.json(users)
29 });
30
31 module.exports = router;
32
33
```

Ein erster REST-Service

- Zum Registrieren der Route:
 - Laden des Moduls mit der Route
 - Registrieren der Route unter einem eigenen Pfad

app.js

```
...  
var restTestRouter = require('./routes/restServiceTest');  
...  
app.use('/', allRoutes.indexRoute);  
app.use('/users', allRoutes.usersRoute);  
app.use('/restTest', restTestRouter);  
...
```

Automatisiertes Testing mit Moca und Chai

Testing von Node.js

- Mit Rest-Client können nur einzelne Abfragen geprüft werden
- Um ganze Sequenzen zu prüfen, wird Testing-Framework benötigt
- Beispiel für Test-Sequenz
 - Einloggen und Token erhalten
 - Token zum Anlegen von User verwenden
 - User verändern
 - User löschen
- Damit diese Sequenz ausgeführt werden kann, müssen jeweils die Ergebnisse des vorangegangenen Schritts genutzt werden (Token, User)
- Im Umfeld von Node.js ist Mocha eines der am meisten verbreiteten Testing-Frameworks
- Chai ist eine Expectation-Bibliothek, um assert-Ausdrücke zu formulieren

Mocha und Chai

- Installation: `npm install --save mocha chai`
- Um Tests von der Konsole zu starten, wird test-Skript konfiguriert

package.json

```
...  
"scripts": {  
  "test": "mocha --reporter spec"  
},  
...
```

Moca: Testdefinition

- Moca führt JavaScript-Dateien im Testverzeichnis aus
- `describe()` definiert Testmodule, können hierarchisch strukturiert werden
- Mit `it()` werden konkrete Tests eingeleitet, in ihnen dürfen nur noch assert-Statements sein

```
describe("Color Code Converter", function() {  
  describe("RGB to Hex conversion", function() {  
    it("converts the basic colors", function() {  
  
      });  
    });  
  
  describe("Hex to RGB conversion", function() {  
    it("converts the basic colors", function() {  
  
      });  
    });  
  });  
});
```

Moca: Testdefinition

- Prüfungen über chai-expect-Statements

```
var expect = require("chai").expect;
var converter = require("../app/converter");

describe("Color Code Converter", function() {
  describe("RGB to Hex conversion", function() {
    it("converts the basic colors", function() {
      var redHex = converter.rgbToHex(255, 0, 0);
      var greenHex = converter.rgbToHex(0, 255, 0);
      var blueHex = converter.rgbToHex(0, 0, 255);

      expect(redHex).to.equal("ff0000");
      expect(greenHex).to.equal("00ff00");
      expect(blueHex).to.equal("0000ff");
    });
  });
});
```


Supertest

- Um einen REST-Server zu testen, muss für die Testausführung der REST-Server gestartet werden
- Hierzu kann das Modul „supertest“ verwendet werden
- Installation: `npm install supertest --save-dev`
- Beispiel: Testen von Homepage

```
var expect = require("chai").expect;  
const request = require('supertest');  
const app = require('../app');  
  
describe("Test the basic application", function () {  
  describe("Test the root page", function () {  
    it("Test the home page", function () {  
      request(app).get('/').expect(200);  
    })  
  });  
});
```

Testing mit Moca, Chai und Supertest

- Beim automatisierten Testen einer REST-API müssen in der Regel die folgenden Aspekte umgesetzt werden
 - Es sollen viele, thematisch gruppierte Tests durchgeführt werden. In der Regel werden für jeden End-Point Test-Module umgesetzt
 - Die Tests sollen zuweilen in einer zentralen Datei ein- und ausgeschaltet werden
 - Es sollten alle Schichten vom Controller bis zur Datenbank getestet werden

Testing mit Mocha, Chai und Supertest: Struktur

- Zum zentralen Ein- und Ausschalten sowie dem einfachen Ergänzen von weiteren Tests sollte im Verzeichnis „test“ ein zentrales Test-Modul sein
- Das Test-Modul sollte die anderen Module aus einem Unterverzeichnis laden

```
const server = require('../bin/www')
const db = require('../services/database/db')

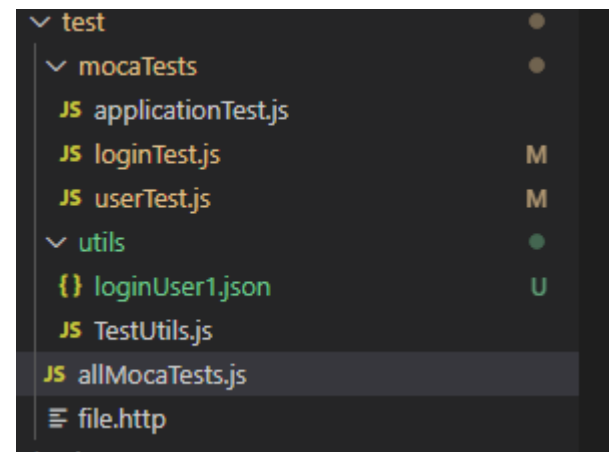
describe.skip('Execute Application Tests', function () {
  require('./mochaTests/applicationTest')
})

describe.skip('Login Tests', function () {
  require('./mochaTests/loginTest')
})

describe('User Service Tests', function () {
  require('./mochaTests/userTest')
})

after(() => {
  console.log("Shut down application")
  db.close();
  server.close();
})
```

- Am Ende der Tests sollte der Server und die Datenbankanbindung beendet werden
- Das kann über den after()-Block geschehen

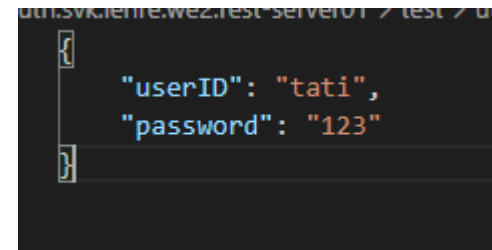


Testing mit Mocha, Chai und Supertest: Test-Util

- Wiederkehrende Funktionen wie das Einloggen eines speziellen Users oder das Parsen von Antworten kann in ein Utility-Modul umgesetzt werden

```
function parseJsonBody(response) {  
  return JSON.parse(JSON.stringify(response.body));  
}  
  
module.exports = {  
  parseJsonBody  
}
```

- Für spezielle Anfragen kann auch der erwartete Json-Content in Dateien abgelegt werden, um ihn einfach an den Server zu senden



```
{  
  "userID": "tati",  
  "password": "123"  
}
```

Testing mit Mocha, Chai und Supertest

- Einige Tests können die korrekte Funktionsweise des Service-Layers testen

```
describe("Test Login mit Json-Body über Session-Service", function () {  
  it("Login with Json-Data via service", function () {  
    user = { ...require('../utils/loginUser1.json') }  
    sessionService.createSessionToken(user, function (err, token, user) {  
      expect(err).to.be.a('null');  
      expect(token).to.be.a('string');  
      expect(user).to.be.an('object');  
      if (user) {  
        expect(user.userID).to.equal("tati");  
      }  
    })  
  });  
});
```

Testing mit Mocha, Chai und Supertest

- Andere Tests können als richtige HTTP-Requests an die Express-Anwendung gesandt werden
- Tests sollten immer mit `end()` abgeschlossen werden, da sonst nicht der Server beendet werden kann

```
describe("Test Login mit Json-Body über Route", function () {  
  it("Login mit Json-Daten via HTTP-Request", function () {  
    user = { ...require('../utils/loginUser1.json') }  
    request(app).post('/login')  
      .send(user)  
      .expect(function (err, res) {  
        var jsonContent = testUtils.parseJsonBody(res);  
        expect(jsonContent => {  
          expect(jsonContent.userID).to.equal(user.userID);  
        });  
      }).end(function (err, res) {  
      });  
    });  
  });  
});
```

Testing mit Moca, Chai und Supertest

- Ein Test bildet häufig eine ganze Sequenz von Schritten ab
 1. Einloggen als User
 2. Abrufen von Daten
 3. Neuen User Anlegen
 4. User-Daten versenden
 5. User wieder löschen
 6. ...
- Die Tests können über „npm run test“ ausgeführt werden
- In Gitlab können die Tests automatisch ausgeführt werden, wenn der Code neu eingecheckt wird
 - Hier wird eine Pipeline angelegt
 - Gitlab kann dann den Befehl „npm run test“ ausführen
 - Der Testbericht wird automatisch erzeugt und im Git angelegt

Logging in Node.js

Logging in Node.js

- Logging besser als Konsole-Ausgaben, da ...
 - Log-Level eingestellt werden kann
 - Log-Nachrichten in Dateien geschrieben werden können
 - Log-Nachrichten angereichert werden können
 - Etc.
- Für Node.js-Anwendungen gibt es zahlreiche Logging-Frameworks/-Bibliotheken
- Einer der populärsten ist „Winston“

Logging mit Winston

- Winston wird häufig für Web-Anwendung auf Node.js verwendet
- Kann auch in Kombination mit „Morgan“ verwendet werden (bei Express-Anwendungen standardmäßig eingebaute Logging-Middleware)
- Anpassungen in app.js von Express-Anwendungen
 - „logger“ nach „morgan“ umbenennen
 - Log-Format von „dev“ nach „combined“ ändern

```
...  
var morgan = require('morgan');  
...  
app.use(morgan('combined'));  
...
```

Logging mit Winston

- Installation: `npm install winston`
- Detaillierte Beschreibung für das Aufsetzen von Winston und die Einbindung im Projekt: <https://www.digitalocean.com/community/tutorials/how-to-use-winston-to-log-node-js-applications>
- Einstellungen für den Logger
 - `level` – Log-Level, die protokolliert werden sollen
 - `filename` – Namen der Datei, in die die Log-Nachrichten geschrieben werden sollen
 - `handleExceptions` – Fangen und Loggen von unbehandelten Exceptions
 - `json` – Speichern die Log-Nachrichten im Json-Format
 - `maxsize` – Maximale Dateigröße, bis eine neue Log-Datei angefangen wird
 - `maxFiles` – Maximale Anzahl von Log-Dateien
 - `colorize` – Farbige Darstellung der Log-Nachrichten auf der Konsole

Logging mit Winston

- Log-Level
 - 0: error
 - 1: warn
 - 2: info
 - 3: verbose
 - 4: debug
 - 5: silly
- Es wird immer alle Nachrichten geloggt, die höher in der Priorität sind als der eingestellte Level (vom Wert her kleiner)

Logging mit Winston

- Die Konfiguration wird über eine JavaScript-Datei vorgenommen, die im config-Verzeichnis abgelegt sein sollte

```
var appRoot = require('app-root-path');
var winston = require('winston');

var options = {
  file: {
    level: 'info',
    filename: `${appRoot}/logs/app.log`,
    handleExceptions: true,
    json: true,
    maxsize: 5242880, // 5MB
    maxFiles: 5,
    colorize: false,
  },
  console: {
    level: 'debug',
  },
  ...
```

Quelle: <https://www.digitalocean.com/community/tutorials/how-to-use-winston-to-log-node-js-applications>

Logging mit Winston

- Auch die Logger-Instanz sollte in dieser JavaScript-Datei angelegt werden

```
...  
  
var logger = winston.createLogger({  
  transports: [  
    new winston.transports.File(options.file),  
    new winston.transports.Console(options.console)  
  ],  
  exitOnError: false, // do not exit on handled exceptions  
});  
  
...
```

Logging mit Winston

- Auch die Log-Nachrichten von „Morgan“ können zusätzlich zur Konsole in die Datei weitergeleitet werden
- Zum Schluss muss der Logger noch exportiert werden

```
...  
  
logger.stream = {  
  write: function(message, encoding) {  
    logger.info(message);  
  },  
};  
  
...  
module.exports = logger;
```

Logging mit Winston

- Winston muss beim Start der Anwendung (app.js) geladen werden
- Anschließend muss noch der Morgan-Logger mit Winston verbunden werden

App.js

```
...  
var winston = require('./config/winston');  
...  
app.use(morgan('combined', { stream: winston.stream }));  
...
```

Logging mit Winston

- Zum Loggen wird...
 - Logger importiert
 - Log-Befehl eingefügt
 - In dem Konfigurationsmodul von Winston kann jederzeit der Log-Level geändert werden

App.js

```
...  
var logger = require('../config/winston');  
...  
logger.debug("Login-Request")  
...
```