1 Node.js – JavaScript auf dem Server

Node.js ist eine serverseitige Plattform, die auf Googles JavaScript-Laufzeitumgebung V8 basiert. Neben der reinen Laufzeitumgebung bringt node.js auch einige fest eingebaute Funktionen mit, die es besonders einfach machen, Serverapplikationen zu implementieren.

In diesem Tutorial wollen wir uns eine kleine Entwicklungsumgebung aufsetzen, mit der es möglich ist, die Beispiele aus dem Buch auszuführen und weiterzuentwickeln.

1.1 Entwicklungsumgebung Teil 1: node.js

Bevor wir weiter ins Detail gehen, sollten wir uns schnell eine Entwicklungsumgebung aufsetzen. Eigentlich gibt es im Internet schon hinreichend Dokumentation zu diesem Thema – weshalb an dieser Stelle nur das Nötigste erwähnt werden soll.

Auf *https://nodejs.org/* können Sie das Installationspaket für node.js herunterladen. Unter Windows finden Sie nach der Installation im Installationsverzeichnis eine spezielle Kommandozeile (*node Commandline*), die dafür sorgt, dass der Befehl `node` und der Paketmanager *npm* im Pfad verfügbar sind.

Jetzt brauchen wir nur noch einen Texteditor und dann können wir auch schon loslegen.

1.1.1 Der erste Webserver

Ein erstes Beispiel für die Implementierung eines Webservers ist auf der Seite nodejs.org unter „DOCS“ zu finden. Dieses haben wir übernommen und zum besseren Verständnis der folgenden Abschnitte ein wenig angepasst:

```
01 const http = require('http')
02
03 const hostname = '127.0.0.1'
04 const port = 3000
05
06 const app = function (request, response) {
07   response.statusCode = 200
08   response.setHeader('Content-Type', 'text/plain')
09   response.end('Hallo Welt!\n')
10 }
11
12 const server = http.createServer(app)
13
14 server.listen(port, hostname, () => {
15   console.log(`Server läuft auf: http://${hostname}:${port}/`)
16 })
```

Jetzt den Code-Schnipsel noch schnell in eine neue Datei kopiert, als *myserver.js* in einem von Ihnen bevorzugten Verzeichnis abgespeichert und mit `node myserver.js` gestartet – et voilà! Schon können wir das Ergebnis unserer Bemühungen begutachten, denn auf der Kommandozeile steht:

„Server läuft auf: http://localhost:3000/“. Wenn wir diese URL jetzt in unseren Browser eingeben, erscheint eine Seite mit dem bedeutsamen Text: „Hallo Welt!“

Zugegeben ist das jetzt auf den ersten Blick noch nicht besonders beeindruckend – aber immerhin kann man schon erahnen, nach welchen Prinzipien ein node.js-Server funktioniert: asynchron und ereignisgesteuert.

Zentrales Element von node.js ist die eingebaute Ereignisschleife. Diese wird automatisch gestartet, nachdem das Script, mit dem das node-Kommando aufgerufen wurde, ausgeführt ist.

Die Methode `createServer` des http-Objektes erzeugt ein Serverobjekt, das man mit einer Callback-Funktion parametrisiert. Diese Funktion bekommt vom Framework zwei Parameter (`request` und `response`) übergeben, die jeweils als Fassade für die Anfrage (Request) und die Antwort (Response) an den Klienten fungieren. Stark vereinfacht kann man sagen: Der Aufruf `server.listen(port, hostname, …)` in Zeile 14 registriert die dem Server vorher in Zeile 12 übergebene Callback-Funktion `app` in der zentralen `node`-Ereignisschleife für Ereignisse vom Typ „http-Aufruf“, die auf Port 3000 auftreten.

Innerhalb unserer Methode können wir dann jeden erfolgten Aufruf über die `request`-Fassade analysieren und dann unsere Antwort über die `response`-Fassade an die Klienten senden.

Wenn wir jetzt das im Beispiel als so bedeutungsschwanger genannte Objekt `app` in ein eigenes Modul auslagern könnten, dann wären wir schon einen kleinen Schritt weiter in Richtung eines eigenen Frameworks vorangekommen. Zum Glück unterstützt uns node.js auch dabei.

1.1.2 Module in Node.js

Node.js implementiert den CommonJS Standard für Module. Eine Datei kann als Modul verwendet werden, wenn sie ein spezielles Objekt namens `module.exports` bereitstellt:

```
01 console.log('Ich bin ein Modul und werde immer auch ausgeführt!')
02
03 module.exports.eineMethode = function() {
04   console.log('ich bin eine Methode des Moduls')
05 }
06
07 exports.einWert = 42
```

Das Objekt `exports`, dem wir in Zeile 7 das Attribut `einWert` mit dem Inhalt „42“ zur Seite stellen, ist übrigens eine Referenz auf das Objekt `module.exports`.

Bevor wir weitermachen, speichern wir den Code in einer Datei in dem gewählten Verzeichnis unter dem Namen *test_modul.js*.

Die Funktion `require` importiert eine Moduldatei, führt sie aus und liefert das `module.exports`-Objekt als Ergebnis zurück:

```
01 const meinModul = require('./test_modul.js')
02
03 meinModul.eineMethode()
04 console.log(meinModul.einWert)
```

Speichern wir diesen Codeschnipsel ebenfalls in einer Datei namens *test_require.js*. Jetzt können wir unser Modul auf der Kommandozeile mit dem folgenden Befehl ausprobieren:

```
node test_require.js
```

Wenn wir alles richtig gemacht haben, ist die Ausgabe:

```
Ich bin ein Modul und werde immer auch ausgeführt!
ich bin eine Methode des Moduls
42
```

Nachdem wir das Konzept der Modularisierung in node.js verstanden haben, können wir jetzt anfangen, uns Gedanken zu machen, wie wir unsere Applikation am besten in Module zerschneiden.

Legen wir also eine weitere Datei an und kopieren die Zeilen 6 bis 10 aus der oben schon erzeugten Datei *myserver.js* hinein. Jetzt exportieren wir die Funktion `app` und speichern die neue Datei unter dem Namen *myapp.js*:

```
01 const app = function (request, response) {
02    response.statusCode = 200
03    response.setHeader('Content-Type', 'text/plain')
04    response.end('Hallo Welt!\n')
05 }
06
07 module.exports = app
```

Die Datei *myserver.js* ändern wir wie folgt ab:

```
01 const http = require('http')
02 const app = require('./myapp.js')
03
04 const hostname = '127.0.0.1'
05 const port = 3000
06
07 const server = http.createServer(app)
08
09 server.listen(port, hostname, () => {
10   console.log(`Server läuft auf: http://${hostname}:${port}/`)
11 })
```

Damit haben wir auch schon die „globale“ Konfiguration unseres Servers (myserver.js) von der eigentlichen Implementierung (myapp.js) getrennt. Jetzt könnten wir anfangen, unsere Applikation in weitere Module aufzuteilen. Aber wir verschieben das erst einmal auf später und kümmern uns stattdessen um die Datenhaltung.

1.2 Entwicklungsumgebung Teil 2: MongoDB

Es gibt für node.js eine ganze Menge von Treibern für die unterschiedlichsten Datenbanken. Für unsere Anwendung wollen wir uns eine Datenbank aussuchen, die möglichst intuitiv zu verwenden ist. MongoDB bietet sich deshalb an, weil wir nicht zwischen mehreren Sprachen springen möchten.

Daten werden in MongoDB in Form von Dokumenten gespeichert. Diese Dokumente werden – wie JavaScript-Objekte – als eine Menge von Eigenschaften beschrieben. Die Syntax ähnelt dabei stark der von JSON.

Herunterladen kann man MongoDB von der Seite *http://www.mongodb.org*. Wir benutzen hier den kostenlosen „Community Server“.

Nach der Installation finden Sie im Installationsordner alle nötigen Tools, die wir brauchen, um einen lokalen Datenbankservice für unsere Applikation aufzusetzen.

Spätestens jetzt wird es Zeit, dass wir uns ein Projektverzeichnis anlegen. Wir nennen unseres *nodeprojects* und legen es im Home-Directory unseres Benutzers an. Dort legen wir ein Unterverzeichnis namens *data* an und starten unseren Datenbankserver mit diesem Verzeichnis als *dbpath*:

```
DeepThought:nodeprojects Stefan$ ls -al
drwxr-xr-x  3 Stefan  users  102 22 Feb 11:11 .
drwxr-xr-x  9 Stefan  users  306 22 Feb 11:10 ..
drwxr-xr-x  2 Stefan  users   68 22 Feb 11:11 data
DeepThought:nodeprojects Stefan$ mongod --dbpath /Users/Stefan/nodeprojects/data
```

Die folgende Ausgabe sollte mit einer Zeile enden, die ungefähr so aussieht:

```
2018-05-21T14:15:13.650+0200 I NETWORK  [initandlisten] waiting for connections on
port 27017
```

Das sagt uns, dass die Datenbank läuft und auf Port 27017 lauscht. Mit dem Kommandozeilen-Client *mongo* können wir uns jetzt – von einer weiteren Kommandozeile aus – bei dieser Datenbank anmelden. Das wird später noch interessant, wenn wir nachsehen möchten, ob unsere Applikation auch das Richtige (oder überhaupt irgendetwas) gespeichert hat.

Für den Moment wollen wir es dabei bewenden lassen, uns einmal anzumelden – wir werden erst die Installation unserer Entwicklungsumgebung abschließen.

1.3 Entwicklungsumgebung Teil 3: Paketmanagement mit npm und express.js

Module und Pakete sind eine der großen Stärken von node.js. Egal, welches Problem Sie haben, die Chancen stehen gut, dass es bereits ein Modul gibt, das zumindest bei der Lösung hilft.

Express ist eines der populärsten Frameworks für node.js. Express.js implementiert bereits große Teile der Infrastruktur eines Webservers – und hilft enorm bei der Implementierung der verbleibenden Teile.

In diesem Abschnitt wollen wir Express installieren und mit der Generierung unseres Applikationsrahmens die Installation unserer Entwicklungsumgebung abschließen.

Zusammen mit node.js wird ein Tool namens *npm* installiert, mit dem sich node.js-Pakete managen lassen. Dieses Tool benutzen wir jetzt, um die Bibliotheken zu installieren, die uns noch zur Fertigstellung unserer Umgebung fehlen. Wir möchten uns nicht zu lange an dieser Stelle aufhalten, schließlich geht es hier immer noch um objektorientierte Konzepte und nicht um die Infrastruktur von node.js. Deshalb werden wir im Folgenden die Installation nur oberflächlich betrachten.

In unserem Projektverzeichnis führen wir nacheinander folgende Befehle aus:

```
DeepThought:nodeprojects Stefan$ sudo npm install -g express

DeepThought:nodeprojects Stefan$ sudo npm install -g express-generator
```

'sudo' gilt natürlich nur, wenn wir uns in einer Unix-artigen Umgebung bewegen. Unter Windows muss die entsprechende Kommandozeile als Administrator ausgeführt werden, damit die Installation klappt.

Jetzt können wir das Kommando `express` benutzen, um einen Rahmen für unsere node.js-Applikation zu generieren:

```
DeepThought:nodeprojects Stefan$ express teamkalender --view=pug

   create : teamkalender\
   create : teamkalender\public\
   create : teamkalender\public\javascripts\
   create : teamkalender\public\images\
   create : teamkalender\public\stylesheets\
   create : teamkalender\public\stylesheets\style.css
   create : teamkalender\routes\
   create : teamkalender\routes\index.js
   create : teamkalender\routes\users.js
   create : teamkalender\views\
   create : teamkalender\views\error.pug
   create : teamkalender\views\index.pug
   create : teamkalender\views\layout.pug
   create : teamkalender\app.js
   create : teamkalender\package.json
   create : teamkalender\bin\
   create : teamkalender\bin\www

   change directory:
     > cd teamkalender

   install dependencies:
     > npm install

   run the app:
     > DEBUG=teamkalender:* npm start
```

Der Express-Generator hat uns unter *teamkalender* eine ganze Verzeichnisstruktur angelegt und mit allerhand Dateien gefüllt. Die erste, die wir uns ansehen – und verändern – möchten, heißt *package.json*. Sie hat (in etwa) folgenden Inhalt:

```
{
  "name": "teamkalender",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "cookie-parser": "~1.4.3",
    "debug": "~2.6.9",
```

```
    "express": "~4.16.0",
    "http-errors": "~1.6.2",
    "morgan": "~1.9.0",
    "pug": "2.0.0-beta11"
  }
}
```

Das JSON-Objekt in der Datei beschreibt unsere gerade generierte Applikation. Der Parameter `dependencies` ist dabei besonders wichtig: Hier werden alle node.js-Pakete aufgeführt, von denen unsere Applikation abhängt. Ändern wir die Datei ein wenig und fügen ein paar Abhängigkeiten hinzu:

```
{
  "name": "teamkalender",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "cookie-parser": "~1.4.3",
    "debug": "~2.6.9",
    "express": "~4.16.0",
    "express-session": "*",
    "http-errors": "~1.6.2",
    "morgan": "~1.9.0",
    "pug": "2.0.0-beta11",
    "moment": "*",
    "mongodb": "*",
    "mongoose": "*"
  }
}
```

- `"express-session"` ist ein Paket, das uns ein einfaches Session-Handling über Cookies zur Verfügung stellt. Das brauchen wir, um uns später zu merken, welcher Benutzer gerade mit uns spricht.
- `"moment"` bietet uns eine etwas angenehmere Schnittstelle zum Arbeiten mit Datums- und Zeitobjekten – in JavaScript abgebildet durch „Date“-Objekte.
- `"mongodb"` ist – wie vermutet – der Datenbanktreiber, den wir benötigen, um uns mit unserer Applikation an der Datenbank anzumelden.
- `"mongoose"` wiederum kapselt `mongodb` mit einer Fassade, die es uns sehr einfach machen wird, Modelle zu definieren und zu implementieren.

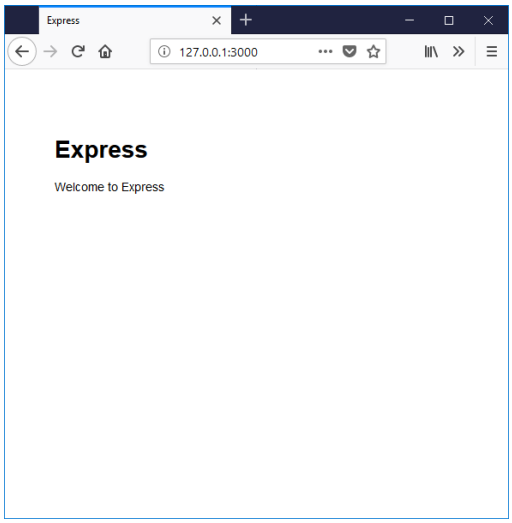
Jetzt schließen wir die Installation unserer Umgebung mit dem folgenden Befehl endgültig ab:

```
DeepThought:teamkalender Stefan$ npm install
```

Das Tool *npm* interpretiert die Datei *package.json*, die es im aktuellen Verzeichnis findet, und installiert die dort angegebenen Module inklusive aller Abhängigkeiten ins Unterverzeichnis *./node_modules*. Wenn das geklappt hat, ist die Installation der ersten Version unserer Applikation vollständig. Wir probieren sie am besten gleich aus:

```
DeepThought:teamkalender Stefan$ DEBUG=teamkalender:* npm start
```

Wenn wir jetzt in den Browser wechseln und dort in der Adresszeile „localhost:3000“ eingeben, sollten wir ungefähr folgendes Bild sehen:



Jetzt haben wir eine vollständige Entwicklungsumgebung und können uns endlich um unsere Applikation kümmern.

1.4 Der Benutzer – Vom Browser in die Datenbank

Unser erstes Anliegen ist es, einen Benutzer in der Datenbank anzulegen. Das Ziel für die folgenden Seiten ist also,

1. den Applikationsrahmen, den uns `express` generiert hat, an unsere Bedürfnisse anzupassen,
2. zu entscheiden, wie ein Benutzer in der Datenbank aussehen soll,
3. ein Modell für den Benutzer zu implementieren,
4. einen View nebst HTML-Template zu schreiben, mit dem sich über den Browser die entsprechenden Daten eingeben lassen,
5. die Applikation zu starten und das Benutzerobjekt in der Datenbank anzulegen.

1.4.1 Die Struktur einer Express-Applikation

Werfen wir zunächst einen Blick auf die Verzeichnisstruktur und die generierten Dateien:

```
teamkalender/
├── app.js
├── bin
│   └── www
├── node_modules/...
├── package.json
├── public
```

```
│   ├── images
│   ├── javascripts
│   └── stylesheets
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.pug
    ├── index.pug
    └── layout.pug
```

Da wir vorhaben, unsere Applikation nach dem MVC-Pattern zu organisieren, sollte sich dies auch in der Verzeichnisstruktur widerspiegeln. Legen wir also zusätzlich noch Verzeichnisse für unsere Modelle und die Controller an und nennen sie *datamodels* und *controller*s. Als weiteren Klassentyp haben wir im Buch in Kapitel „10.3 Das Framework“ die Aktion eingeführt. Die konkreten Klassen sollen ebenfalls ein eigenes Verzeichnis bekommen: *actions*.

Im Verzeichnis *views* befinden sich Dateien mit der Endung „*.pug“*. Hierbei handelt es sich um View-Templates in der Templatesprache Pug. Deshalb benennen wir das Verzeichnis um in *templates*. Jetzt fehlt natürlich noch ein Verzeichnis für unsere Views – das legen wir prompt an und nennen es wie gehabt *views*.

Ein weiteres Verzeichnis werden wir unserem Framework widmen – und nennen es dementsprechend auch *framework*.

Die Dateien im Verzeichnis *routes* stellen in einer „originalen“ Express-Umgebung die Verknüpfung der URIs mit unseren Diensten dar. Die Namen *index.js* und *users.js* sind allerdings nur als Vorschläge zu verstehen. Wir löschen zunächst die Datei *users.js*. Die Datei *index.js* lassen wir erst einmal bestehen, da sie uns in den folgenden Beispielen helfen wird, die Struktur einer Express-Applikation besser zu verstehen. Später wird das Verzeichnis komplett gelöscht.

Widmen wir uns jetzt der Datei *app.js*. Diese Datei ist der Dreh- und Angelpunkt einer Express-Applikation. Hier laufen alle Fäden zusammen. Deshalb müssen wir auch gleich ein paar kleine, aber wichtige Änderungen vornehmen, damit die Datei zu unserer neuen Verzeichnisstruktur passt. Werfen wir erst einen Blick in die Datei – die zu ändernden Zeilen sind fett markiert:

```
01 var createError = require('http-errors')
02 var express = require('express')
03 var path = require('path')
04 var cookieParser = require('cookie-parser')
05 var logger = require('morgan')
06
07 var indexRouter = require('./routes/index')
08 var usersRouter = require('./routes/users')
09
10 var app = express()
11
12 // view engine setup
13 app.set('views', path.join(__dirname, 'templates'))
```

```
14 app.set('view engine', 'pug')
15
16 app.use(logger('dev'))
17 app.use(express.json())
18 app.use(express.urlencoded({ extended: false }))
19 app.use(cookieParser())
20 app.use(express.static(path.join(__dirname, 'public')))
21
22 app.use('/', indexRouter)
23 app.use('/users', usersRouter)
24
25 // catch 404 and forward to error handler
26 app.use(function (req, res, next) {
27   next(createError(404))
28 })
29
30 // error handler
31 app.use(function (err, req, res, next) {
32   // set locals, only providing error in development
33   res.locals.message = err.message
34   res.locals.error = req.app.get('env') === 'development' ? err : {}
35
36   // render the error page
37   res.status(err.status || 500)
38   res.render('error')
39 })
40
41 module.exports = app
```

Die vorzunehmenden Änderungen in aller Kürze:

In den Zeilen 7 und 8 werden die routen als Module geladen.

```
07 var indexRouter = require('./routes/index')
08 var usersRouter = require('./routes/users')
```

Die Datei *users.js* haben wir ja gerade gelöscht, weshalb wir das hier jetzt nachziehen müssen – wir können die Zeile löschen oder einfach auskommentieren:

```
07 var indexRouter = require('./routes/index')
08 // var usersRouter = require('./routes/users')
```

In den Zeilen 22 und 23 werden eben diese Module benutzt, um dem Express-Framework mitzuteilen, wo Anfragen bearbeitet werden sollen. Das müssen wir auch anpassen:

```
22 app.use('/', indexRouter)
23 // app.use('/users', usersRouter)
```

In Zeile 13 schließlich wird dem Applikationsobjekt gesagt, wo es die View-Templates findet. Da wir das Verzeichnis umbenannt haben, tun wir das auch hier:

```
14 app.set('views', path.join(__dirname, 'templates'));
```

Jetzt müssen wir uns nur noch mit der Datenbank verbinden. Das machen wir sinnvollerweise auch hier – und als Allererstes! Also fügen wir in *app.js vor* Zeile 1 noch folgende Zeilen ein:

```
01 var mongoose = require('mongoose')
02
03 mongoose.Promise = global.Promise
04
05 mongoose.connect('mongodb://localhost/teamkalender', {
06   useMongoClient: true
07 })
```

1.4.2 Dokumente in der Datenbank

Daten werden in MongoDB in Datenbanken gespeichert. In diesen Datenbanken befinden sich Collections in denen wiederum die Daten in Form sogenannter Dokumente abgelegt werden. Ein Dokument ist dabei für MongoDB nur eine Ansammlung von Eigenschaften, die auch selbst wieder Dokumente sein können – Unterdokumente.

Wie schon angedeutet liefert mongoose uns eine Abstraktionsschicht für den Datenbanktreiber, sodass wir uns vollständig auf unsere Modelle konzentrieren können.

Zuerst wollen wir uns um die Beschreibung unseres Benutzer-Modells kümmern. Wir erzeugen im noch leeren Verzeichnis *datamodels* eine neue Datei namens *benutzer.js* – und füllen sie mit Leben:

Dazu brauchen wir zunächst das Modul `mongoose`, das uns ein Objekt namens `Schema` zur Verfügung stellt:

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
```

Eine Instanz von `mongoose.Schema` wird gleich die vollständige Beschreibung unseres Modells enthalten. Diese Beschreibung zerfällt für uns in drei Bestandteile, die wiederum durch drei Objekte repräsentiert werden:

- die Beschreibung der Eigenschaften des Dokuments
- Informationen über die Collection, in der das Dokument gespeichert werden soll
- die Methoden des Modells – die wir für den Moment hinten anstellen wollen

Definieren wir also zunächst die Eigenschaften eines Benutzer-Objekts:

```
01 var SchemaObj = {
02    username : {
03         type: String,
04         required: '{PATH} ist ein Pflichtfeld!',
05         unique: true
06     },
07
08     vorname: String,
09     nachname: String,
10
11     pwd_hash: String,
```

```
12      pwd_salt: String,
13
14      abwesenheiten: [{
15          von: Date,
16          bis: Date,
17          fix: Boolean
18      }],
19
20      rollen: [{type: Schema.Types.ObjectId, ref: 'Rolle'}]
21  };
22
```

Als Nächstes entscheiden wir uns für einen Namen für die Collection:

```
23  var CollectionObj = {
24      collection: 'Benutzer'
25  }
26
```

Jetzt müssen wir unser neues Dokument bei mongoose anmelden. Solange die Applikation läuft, merkt sich mongoose alle angemeldeten Modelle. Deshalb schreiben wir eine Funktion, die wir in der Datei *app.js* nur einmal – nämlich beim Start der Applikation – aufrufen müssen. Damit das klappt, exportieren wir die Funktion:

```
27  exports.init = function() {
28      var mongooseSchema = new Schema(SchemaObj, CollectionObj);
29      return mongoose.model(CollectionObj.collection, mongooseSchema);
30  }
```

Mongoose bietet uns natürlich auch eine Möglichkeit, um an ein angemeldetes Modell heranzukommen. Das werden wir häufiger brauchen – und exportieren es deshalb ebenfalls als Funktion unseres Benutzer-Moduls.

```
31  exports.get = function() {
32      return mongoose.model(CollectionObj.collection);
33  };
34
```

In der Datei *app.js* fügen wir gleich nach der Zeile, die uns mit der Datenbank verbindet, den folgenden Code ein:

```
05  mongoose.connect('mongodb://localhost/teamkalender', {
06    useMongoClient: true
07  })
08  var Benutzer = require('../datamodels/benutzer');
09  Benutzer.init();
    ...
```

Damit haben wir eine erste Version unseres Benutzer-Modells implementiert – probieren wir es gleich aus!

1.4.3 Die Route als Einstieg für einen http-Request

In Express sind sogenannte Routen der Ausgangspunkt für die Ausführung der Geschäftslogik. Routen werden über das Router-Objekt des Express-Frameworks in das Applikations-Framework eingehängt. Dieses Objekt hat für jedes HTTP-Verb eine Funktion, die zwei Parameter erhält:

1. Die Route, die wir uns der Einfachheit halber erst einmal als den Teil einer URL vorstellen, der nach dem Hostnamen kommt. Ein Beispiel wäre *http://localhost:3000/admin*, wobei „/admin“ die Route darstellt.
2. eine Callback-Funktion, die aufgerufen wird, wenn der Server nach der angegebenen Route gefragt wird

Ein Beispiel bringt hier mehr als alle Erklärungen. Hier kommt die Implementierung für unsere erste Route (*/admin*) in der Datei *routes/index.js*:

```
01 var express = require('express')
02 var router = express.Router()
03
04 var Benutzer = require('../datamodels/benutzer')
05
06 router.get('/admin', (request, resonse, next) => {
07     var BenutzerModell = Benutzer.getMongooseModel();
08     BenutzerModell.find(function(err, alle_benutzer) {
09         if(err) {
10             return next(err);
11         }
12
13         response.render('admin', {
14             title: 'Administration',
15             benutzerliste: alle_benutzer,
16         })
17     })
18 })
19
20 module.exports = router;
```

Gehen wir das Beispiel im Detail durch.

In **Zeile 4** holen wir uns unser vorhin geschriebenes Benutzermodell.

In **Zeile 6** tragen wir die Route für eine HTTP „GET“-Anfrage ein, indem wir `router.get(...)` mit unserer Route */admin* und einer Callback-Funktion aufrufen. Die Funktion bekommt zunächst die beiden schon aus unserem node.js-Beispiel bekannten Parameter `request` und `response`. Außerdem bekommt unsere Funktion ein weiteres Funktionsobjekt geliefert, das wir aufrufen können, wenn wir die Anfrage nicht abschließend bearbeiten können, weil beispielsweise ein Fehler aufgetreten ist.

Unsere Funktion macht an sich nicht viel. In **Zeile 8** wird das Benutzer-Modell benutzt, um alle Benutzer aus der Datenbank zu lesen. Das Lesen findet asynchron statt, weshalb wir auch hier eine Callback-Funktion mitgeben müssen, die vom System aufgerufen wird, sobald die Daten – oder ein Fehlerobjekt – von der Datenbank geliefert wurden.

Wenn alles gut gegangen ist, wird in den **Zeilen 13 bis 16** das Ergebnis in Form eines Views an den Klienten ausgegeben. Hierzu wird auf dem Ergebnis-Objekt (`response`) die Funktion `render` aufgerufen. Diese Funktion erhält als Parameter ein Template aus dem in *app.js* konfigurierten Verzeichnis für View-Templates, sowie ein Datenobjekt, das im View-Template beim Aufbau der HTMLSeite verfügbar sein soll.

In **Zeile 20** wird am Ende noch das `router`-Objekt exportiert, damit es in der Datei *app.j*s per `require` verfügbar wird.

Das war jetzt so einfach, dass wir gleich noch eine Route in die Datei einfügen – natürlich *vor* Zeile 20:

```
20 router.post('/admin/benutzer', function(request, response, next) {
21     var BenutzerModell = Benutzer.getMongooseModel();
22     var NewUser = new BenutzerModell({
23         username: request.body.username,
24         vorname: request.body.vorname,
25         nachname: request.body.nachname,
26         pw_hash: '',
27         pw_salt: '',
28         abwesenheiten: []
29     })
30
31     NewUser.save(function(err) {
32         if(err) {
33             return next(err)
34         }
35         response.redirect('back')
36     })
37 })
38
39 module.exports = router
```

Wie wir in der neuen **Zeile 20** sehen können, wird diese jetzt nicht mit der Funktion `get` verknüpft, sondern mit `post`. Das bedeutet, dass unsere zweite Callback-Funktion nur aufgerufen wird, wenn der Klient mit dem HTTP-Verb POST anfragt.

Die **Zeilen 22 und 31** zeigen, was wir mit unserem Mongoose-Modell noch so alles anfangen können: Zuerst erzeugen wir in **Zeile 22** ein neues Benutzer-Objekt, indem wir unsere Fassade als Konstruktor-Funktion verwenden und dieser ein Objekt mit den zur Erzeugung eines neuen Dokuments benötigten Daten-Elementen mitgeben. In **Zeile 31** speichern wir den so erzeugten neuen Benutzer in der Datenbank. Das ist wieder eine asynchrone Funktion, sodass wir eine Callback-Funktion mitgeben müssen. Wenn kein Fehler aufgetreten ist, benutzen wir das `response`-Objekt in **Zeile 35** nicht dazu, eine HTML-Seite an den Klienten zu schicken, sondern um den Aufruf umzuleiten – zurück auf die Seite, von der der Aufruf ursprünglich kam.

Jetzt haben wir zwei Routen. Die erste zeigt uns die Administrations-Seite an, die zweite fügt, wenn alles gut geht, einen neuen Benutzer in die Datenbank ein und leitet dann um – zurück auf die erste Route.

Jetzt fehlt zu unserem Glück nur noch ein Template, mit dem wir die HTML-Seite beschreiben, die der Benutzer über seinen Browser schließlich zu Gesicht bekommt.

1.4.4 Ein View-Template mit Pug

Pug ist eine relativ mächtige Template-Engine, die sehr gut mit Express harmoniert. Wer im Detail wissen möchte, was diese Engine so alles kann, der kann auf *http://pugjs.org* eine umfangreiche Dokumentation finden. Hier wollen wir wie üblich nur das nötigste an einem Beispiel erläutern.

Ein Tipp gleich vorweg: Pug ist **sehr** empfindlich, wenn es darum geht, wie weit eine Zeile eingerückt ist und ob dafür Tabs oder Spaces verwendet wurden. Leider wird dann oft nicht das angezeigt, was man sich erhofft – und man neigt dazu, an der falschen Stelle zu suchen. Spaces zählen hilft! Das ist aber auch schon der einzige Nachteil.

Genug der Vorrede: Hier ist das Beispiel – und damit das letzte Puzzleteil auf dem Weg zu unserer ersten Seite.

Alle Layouts der Applikation sind im Verzeichnis *templates* untergebracht. Unser Seitenlayout besteht aus zwei Teilen: dem Basislayout (*layout.pug*) und dem eigentlichen Layout, welches das Basislayout erweitert. Werfen wir zunächst einen Blick auf das Basislayout:

```
01 doctype html
02 html
03     head
04         title= title
05         link(rel='stylesheet', href='/stylesheets/style.css')
06         block styles
07         block scripte
08     body
09         .page-header
10             block header
11         .page-menu
12             block navigation
13                 .auswahl
14                     != user_menue
15         .page-content
16             block content
```

Eines der Hauptanliegen von Pug ist, den Code für die Seitenvorlagen so knapp und übersichtlich wie möglich zu halten. Die Entwickler haben sich aus diesem Grund dazu entschlossen, die Verschachtelung der HTML-Elemente durch die Tiefe ihrer Einrückung auszudrücken – so wie in der Sprache Python.

Die wichtigsten Elemente der Sprache sind die Befehle `extends` und `block`. Wie die Elemente zusammenarbeiten, sehen Sie, wenn Sie sich die zweite Template-Datei (*admin.pug*) ansehen:

```
01 extends layout
02
03 block header
04   h1 #{title}
05
```

```
06 block content
07   .AdminBenutzerDaten
08     h2 Eingetragene Benutzer
09     form#UserAdd(name='benutzer-neu', method='post', action='/admin/benutzer')
10     table.UserTable
11       thead
12         th Benutzername
13         th Vorname
14         th Nachname
15         th Passwort
16         th Aktion
17       tbody
18         tr
19           td
20             input#txtUserName(type='text', placeholder='username',
name='username')
21           td
22             input#txtVName(type='text', placeholder='Vorname', name='vorname')
23           td
24             input#txtNName(type='text', placeholder='Nachname', name='nachname')
25           td
26             input#pwInitPW(type='password', placeholder='initiales Passwort',
name='passwort')
27           td
28             button#btnSubmitUser(type='Submit') Benutzer Anlegen
29         each user in benutzerliste
30           tr
31             td #{user.username}
32             td #{user.vorname}
33             td #{user.nachname}
34             td ***
35             td
36               a(href='#') bearbeiten
37               |  
38               a(href='#') l&ouml;schen
```

Mit dem Befehl `extends` (Zeile 1 im Listing oben) leiten Sie Ihr Layout von einem Basislayout ab. Das Basislayout wiederum ist mit dem Befehl block in – wie der Name schon sagt – Blöcke unterteilt, die Sie im abgeleiteten Layout überschreiben kann. Der Inhalt des Blocks (der in diesem Fall *content* heißt) wird dabei durch den Inhalt des „abgeleiteten" Layouts ersetzt.

In **Zeile 9** definieren wir das Formular, in das wir die Daten für unseren neuen Benutzer eingeben können. Der Parameter `action` enthält dabei unsere oben beschriebene zweite Route (zum Anlegen eines neuen Benutzers) und im Parameter `method` wird das http-Verb angegeben, mit dem wir die Route vorher beim Express-Router angemeldet haben.

Ab Zeile 29 erzeugen wir zu guter Letzt mit einer Schleife über das Array „benutzerliste“, das dem Template mit dem Befehl `render` übergeben wird, für jeden aus der Datenbank gelesenen Benutzer eine Zeile in der HTML-Tabelle.

Die beiden Links, die wir in den **Zeilen 36 und 38** erzeugen, haben erst einmal noch keine Funktion – wir überlassen es Ihnen, die Anwendung an dieser Stelle nach eigenem Gusto zu erweitern.

1.5 Fazit

Jetzt haben wir also eine erste lauffähige Webanwendung, anhand derer man einiges an Grundlagenforschung betreiben kann. Natürlich können wir in diesem Rahmen nur an der Oberfläche kratzen – Pug, MongoDB, Express und vor allem Node.js können noch viel mehr.

Für tiefer gehende Informationen und einen umfassenderen Einstieg in das Thema Webentwicklung seien an dieser Stelle noch die hervorragenden Bücher aus dem Rheinwerk Verlag empfohlen.

Wir wünschen allen Lesern viel Spaß beim Experimentieren und Weiterentwickeln!