Prototypen

Autor	Daniel Rötzer
Erstellungsdatum	22.11.16

Inhalt

Abbildungsverzeichnis....5
1 Vorbereitung....7
1.1 Installation....7
1.1.1 Node.js....7
1.1.2 RethinkDB....7
1.1.3 Paket- bzw. Modulinstallation....8
1.1.3.1 package.json....8
1.1.3.2 Node Package Manager....9
1.1.3.3 Vorteil....10
1.2 Server starten....10
1.3 Datenbank starten....10
1.4 Model View Controller – MVC....10
1.4.1 Ordnerstruktur....11
1.4.2 Bedeutungen....11
1.4.3 Kommunikation der einzelnen Bereiche....11
1.5 Typischer Aufbau von Node.js Projekten....12
1.5.1 configs....12
1.5.2 controllers, models & views....12
1.5.3 public....12
1.5.4 routes....13
1.5.5 tests....13
1.5.6 package.json....13
1.5.7 server.js....13
1.6 Verfügbarkeit der Prototypen....13
2 Prototypen....14
2.1 HelloWorld....14
2.1.1 Verwendete Pakete....14
2.1.2 MVC....14
2.1.3 Ordnerstruktur....14
2.1.4 Express einbinden....14
2.1.5 Browser Aufruf ermöglichen....14
2.1.6 Server starten....15
2.1.7 Hello World im Browser ausgeben....15
2.1.7.1 Ein weiteres Beispiel eines anderen Routing Pfades....15
2.2 Modules....15

2.2.1 Verwendete Pakete 15
2.2.2 MVC 16
2.2.3 Ordnerstruktur 16
2.2.4 Erstellung und Einbindung 16
2.2.5 Config.js 16
2.2.6 Anwendung der exportierten Funktionen und Zeichenketten 17
2.3 Formulare 17
2.3.1 BodyParser 18
2.3.1.1 Verwendete Pakete 18
2.3.1.2 MVC 18
2.3.1.3 Ordnerstruktur 18
2.3.1.4 Einbindung und Anwendung 18
2.3.1.5 Formulardaten empfangen und zurücksenden 19
2.3.1.6 Ausführung 19
2.3.2 Pug 20
2.3.2.1 Verwendete Pakete 20
2.3.2.2 MVC 20
2.3.2.3 Ordnerstruktur 20
2.3.2.4 Pug Syntax 20
2.3.2.5 Einbindung 21
2.3.2.6 index.pug 21
2.3.2.7 hello.pug 22
2.3.2.8 Im Browser aufrufen 22
2.3.2.9 Ausführung 23
2.4 Logging 23
2.4.1 Warum Logging anwenden? 23
2.4.2 Winston 24
2.4.2.1 Verwendete Pakete 24
2.4.2.2 MVC 24
2.4.2.3 Ordnerstruktur 24
2.4.2.4 Logging-Levels 24
2.4.2.5 Erstellung eines neuen Loggers 25
2.4.2.5.1 Definierbare Eigenschaften der Transporte: 25
2.4.2.5.2 Weitere Eigenschaften des Loggers 26
2.4.2.5.3 Export unseres Loggers 26
2.4.2.6 Einbindung und Anwendung in unserem Server 26

2.4.2.7 Ausführung 27
2.4.3 Morgan 28
2.4.3.1 Verwendete Pakete 28
2.4.3.2 MVC 29
2.4.3.3 Ordnerstruktur 29
2.4.3.4 Einbindung und Anwendung 29
2.4.3.5 Test der http-Protokollierung 29
2.5 RethinkDB 30
2.5.1 Verwendete Pakete 30
2.5.2 MVC 30
2.5.3 Ordnerstruktur 30
2.5.4 Erstellung und Einbindung 31
2.5.5 Datenbankabfragen 31
2.5.5.1 initDB 31
2.5.5.2 insertTestData 32
2.5.6 Server 33
2.5.7 Ausführung 34
2.6 Final 34
2.6.1 Verwendete Pakete 34
2.6.2 MVC 34
2.6.3 Ordnerstruktur 35
2.6.4 Config 36
2.6.5 Controller 37
2.6.6 Routes 37
2.6.7 Views 38
2.6.7.1 index.pug 38
2.6.7.2 store.pug 38
2.6.8 Models 39
2.6.9 Index.js 41
2.6.10 Ausführung 43

Abbildungsverzeichnis

Abbildung 1: Node.js Installer Download 7
Abbildung 2: RethinkDB Download 8
Abbildung 3: package.json Dateierstellung beginnen mit: npm init 8
Abbildung 4: Angaben der Projektinformationen und Endergebnis 9
Abbildung 5: Paketinstallation 9
Abbildung 6: Kommunikation der einzelnen MVC-Bereiche 11
Abbildung 7: Typischer Aufbau von Node.js Projekten 12
Abbildung 8: HelloWorld Ordnerstruktur 14
Abbildung 9: Express in den Server einbinden 14
Abbildung 10: Server „horcht“ auf localhost:3000 14
Abbildung 11: Server starten mit - node index.js 15
Abbildung 12: Antwort an die GET-Anfrage programmieren 15
Abbildung 13: Anderen Routing-Pfad definieren 15
Abbildung 14: Modules Ordnerstruktur 16
Abbildung 15: Config-Datei einbinden 16
Abbildung 16: Einfache Funktionen und Informationen exportieren 16
Abbildung 17: Exportierte Funktionen mithilfe der Shorthand-Methode 17
Abbildung 18: Anwendungsbeispiele unseres Moduls 17
Abbildung 19: Ausgaben in der Konsole nach Serverstart 17
Abbildung 20: BodyParser Ordnerstruktur 18
Abbildung 21: body-parser einbinden und anwenden 18
Abbildung 22: Ordner für die statischen Dateien festlegen 18
Abbildung 23: Inhalt von index.html 19
Abbildung 24: POST-Methode definieren 19
Abbildung 25: Formulareingabe 19
Abbildung 26: Ausgabe des übergebenen Wertes 19
Abbildung 27: Pug Ordnerstruktur 20
Abbildung 28: Einfache Pug Seite 21
Abbildung 29: Einfache HTML Seite 21
Abbildung 30: view engine auf pug setzen 21
Abbildung 31: Inhalt von index.pug 22
Abbildung 32: Inhalt von hello.pug 22
Abbildung 33: Antwort an die GET-Anfrage programmieren 22
Abbildung 34: Formulardaten mittels POST-Methode annehmen und danach ausgeben 23
Abbildung 35: Formulareingabe 23
Abbildung 36: Ausgabe des Übergebenen Wertes 23
Abbildung 37: Winston Ordnerstruktur 24
Abbildung 38: Winston einbinden 25
Abbildung 39: Logger erstellen 25
Abbildung 40: Weitere Einstellungen definieren 26
Abbildung 41: Logger exportieren 26
Abbildung 42: Logger einbinden 26
Abbildung 43: Logging Anwendungsbeispiele 26
Abbildung 44: Andere Schreibweise für Logs 27
Abbildung 45: Logging Ausgabe in der Konsole 27
Abbildung 46: Ändern des Log-Levels der Konsole 27
Abbildung 47: Neue Logging Ausgabe in der Konsole 27

Abbildung 48: Nicht abgefangener Fehler 28
Abbildung 49: Protokollierte Fehlermeldung 28
Abbildung 50: Morgan Ordnerstruktur 29
Abbildung 51: writeStream erstellen und Morgan anwenden 29
Abbildung 52: 3 definierte Routing-Pfade 30
Abbildung 53: Ausschnitt der Log-Datei 30
Abbildung 54: RethinkDB Ordnerstruktur 30
Abbildung 55: Datenbankinformationen 31
Abbildung 56: Benötigte Module laden 31
Abbildung 57: Export von 2 Datenbankabfragen 31
Abbildung 58: Sicherstellen, dass Datenbank und Table erstellt sind 32
Abbildung 59: Einfache Testdaten einfügen 33
Abbildung 60: Exportierte Datenbankabfragen mit Zeitverzögerung ausführen 33
Abbildung 61: Ergebnisse der ausgeführten Datenbankabfragen 34
Abbildung 62: Final Ordnerstruktur 35
Abbildung 63: Inhalt von config.js 36
Abbildung 64: Inhalt von winston.js 36
Abbildung 65: Inhalt von controller.js 37
Abbildung 66: Inhalt von router.js 38
Abbildung 67: Formular von index.pug 38
Abbildung 68: Rückmeldung an den Client 38
Abbildung 69: Benötigte Module laden 39
Abbildung 70: Export der 2 Funktionen 39
Abbildung 71:Programmierung von initDB 40
Abbildung 72: Programmierung von insertData 41
Abbildung 73: Teil 1 von index.js 41
Abbildung 74: Teil 2 von index.js 42
Abbildung 75: Rückmeldungen bei Serverstart 43
Abbildung 76: Aufruf der Startseite 43
Abbildung 77: Rückmeldung an den Client 43

1 Vorbereitung

1.1 Installation

Als erstes wird erklärt, welche Programme installiert werden müssen und wie diese Installiert werden. Alle Installationen werden für das Betriebssystem Windows beschrieben. Folgende Programme werden benötigt:

- Node.js
- RethinkDB
- Projektspezifische Pakete bzw. Module

1.1.1 Node.js

Um Node.js zu installieren, lädt man den Installer, von deren Homepage, herunter. Wichtig ist, dass die Version auf „Current“ gesetzt ist, um alle Features von Node.js zu erhalten und keine gekürzte Variante.

Außerdem benötigt die Datenbank „RethinkDB“ eine 64-bit Node.js Installation

Den Installer findet man hier: https://nodejs.org/en/download/current/

Folgend sieht man einen kleinen Ausschnitt der Homepage. Hier einfach den Windows Installer auswählen.

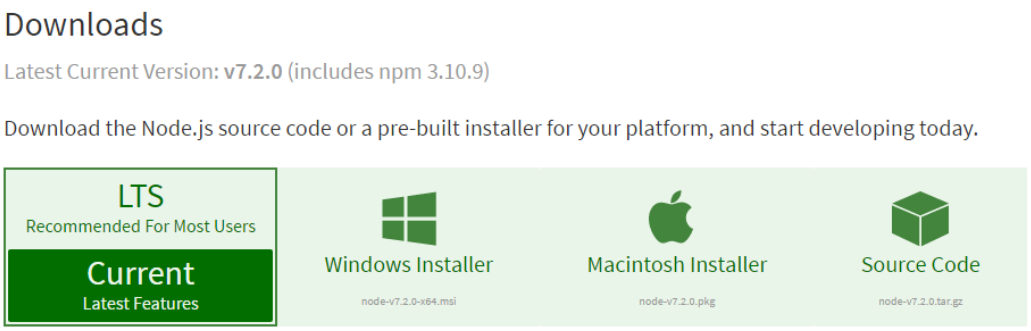

Abbildung 1: Node.js Installer Download

Ist der Installer fertig heruntergeladen, führt man diesen einfach aus und folgt den Anweisungen des Programmes.

1.1.2 RethinkDB

Als erstes lädt man sich die Datenbank von deren Homepage herunter:
https://www.rethinkdb.com/docs/install/windows/

Bevor man die Datenbank herunterlädt, findet man nochmals den Hinweis, dass RethinkDB ein 64-bit Windows benötigt, deshalb ist es auch optimal, wen die Node.js Installation 64-bit ist.

Die Datenbank erhält man in einem Zip-Archiv. Dieses wird einfach entpackt und man erhält die benötigte Datenbank, welche man einfach in einem beliebigen Ordner speichert. Am besten speichert man diese, wo sie leicht zugänglich ist.

Installiert werden muss nun nichts mehr, sondern man führt die Datenbank, die man in einem gewünschten Ordner gespeichert hat, einfach mit Doppelklick aus.

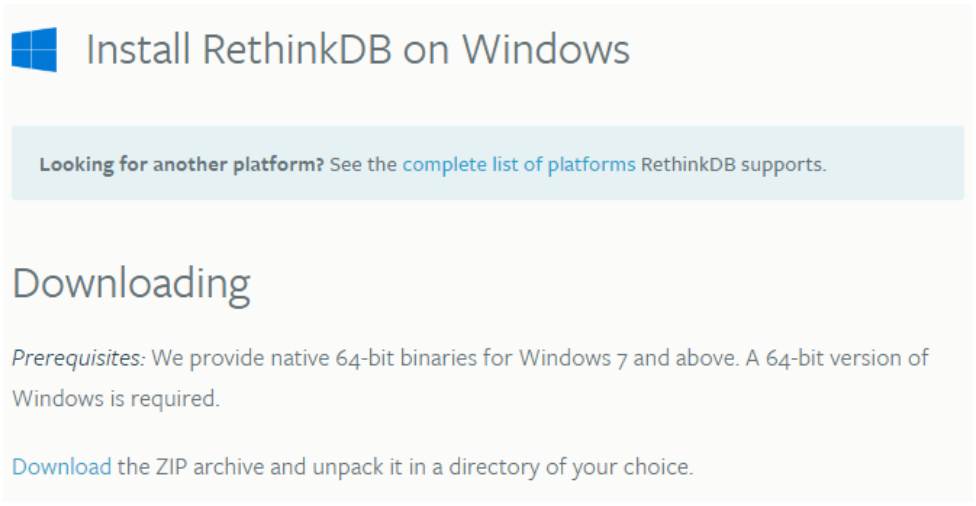

Abbildung 2: RethinkDB Download

1.1.3 Paket- bzw. Modulinstallation

Pakete und Module, sind exakt dasselbe. Der Begriff Modul ist in Node.js gängiger, aber unter Pakete kann man sich möglicherweise besser vorstellen, was damit gemeint ist.

Es gibt Unmengen an verschiedensten Pakten bzw. Modulen für Node.js, die einem das Programmieren erleichtern oder Funktionen hinzufügen. Diese werden mit dem – **Node Package Manager (NPM)** – installiert.

Bevor man Module installiert, erstellt man zuerst eine – **package.json** – Datei.

1.1.3.1 package.json

Jedes ordentliche Node.js Projekt, enthält diese Datei. Sie enthält verschiedenste Projektinformationen und die für dieses Projekt nötigen Module.

Um diese Datei einfach zu erstellen, öffnet man die Eingabeaufforderung. Hierzu führt man zuerst den Befehl – **WINDOWS-Taste + R** – aus und gibt in dem daraus resultierenden Fenster – **cmd** – ein.

In CMD navigiert man zu seinem Projektordner (kein Unterordner des Projekts). Nun führt man den Befehl „npm init“ aus.

```
C:\Users\Rötzer\Documents\GitHub\RVP-Repository>npm init
```

Abbildung 3: package.json Dateierstellung beginnen mit: npm init

Node führt dann die Erstellung durch und fragt nach bestimmten Projektinformationen, die direkt in CMD beantwortet werden. Wird nichts Spezifisches angegeben, so wird der Wert in der Klammer übernommen.

Nicht alle Werte müssen angegeben werden, die wichtigsten sind:

- name (= Projektname)
- entry point (= Dateiname, indem der Server programmiert ist)

Der nächste Screenshot zeigt die Abgefragten Projektinformationen und wie das File nachher aussehen würde:

```
Press ^C at any time to quit.
name: (RVP-Repository) rvp
version: (1.0.0)
description: Project description
entry point: (index.js)
test command:
git repository: (https://github.com/danielroetzer/RVP-Repository.git)
keywords:
author: DR
license: (ISC)
About to write to C:\Users\Rötzer\Documents\GitHub\RVP-Repository\package.json:

{
  "name": "rvp",
  "version": "1.0.0",
  "description": "Project description",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/danielroetzer/RVP-Repository.git"
  },
  "author": "DR",
  "license": "ISC",
  "bugs": {
    "url": "https://github.com/danielroetzer/RVP-Repository/issues"
  },
  "homepage": "https://github.com/danielroetzer/RVP-Repository#readme"
}
```

Abbildung 4: Angaben der Projektinformationen und Endergebnis

Zuletzt muss nur noch „Enter“ gedrückt werden und das File wird erstellt.

Nun sind die Vorbereitungen abgeschlossen.

1.1.3.2 Node Package Manager

Jetzt gelangen wir zur eigentlichen Paketinstallation. Hierzu navigiert man in der Eingabeaufforderung wieder zu seinem Projekt, dort wo sich das package.json File befindet und führt folgenden Befehl aus, indem man das Feld „Paketname“ mit dem gewünschten Paket ersetzt.

```
C:\Users\Rötzer\Documents\GitHub\RVP-Repository>npm install paketname --save
```

Abbildung 5: Paketinstallation

Befehl genauer erklärt: npm install [paketname] [--save]

Die Parameter in eckiger Klammer sind optional.

- Werden alle Parameter angegeben, so wird das gewünschte Paket installiert und ebenfalls im – package.json – Dokument vermerkt.
- Wird nur der Paketname ergänzt, so wird das gewünschte Paket installiert, ohne das es vermerkt wird.
- Werden alle optionalen Parameter weggelassen, so werden automatisch alle Vermerkten Pakete installiert.

Es ist auch mögliche mehrere, durch Lehrzeichen getrennte, Pakete mit einem Befehl zu installieren.

1.1.3.3 Vorteil

Angenommen ein Projekt wird weitergegeben oder wird in einem Versionsverwaltungssystem gespeichert. Weil die installierten Pakete vermerkt werden, muss kein einziges Paket kopiert oder im Versionssystem gespeichert werden. Wer das Programm ausführen will, installiert zuerst ganz einfach alle notwendigen Pakete durch den Befehl: **npm install**

1.2 Server starten

Server werden in JavaScript Dateien erstellt. Diese Serverdatei darf in keinem Unterordner liegen, sondern befindet sich direkt im Hauptverzeichnis, genauso wie die – **package.json** – Datei.

Einen erstellten Server starten ist das einfachste. Hierzu wieder in der Eingabeaufforderung zu der Serverdatei navigieren und anschließend den Befehl – **node servername.js** – ausführen. Hier setzt man einfach den Namen der eigenen Serverdatei ein und der Server startet.

Dies wird im Prototyp „**HelloWorld**“ nochmals gezeigt.

1.3 Datenbank starten

Um die Datenbank zu starten, wird einfach die heruntergeladene Ausführungsdatei von RethinkDB, mit Doppelklick gestartet.

1.4 Model View Controller – MVC

Das MVC Prinzip beschreibt, wie die Programmierung aufgeteilt wird, zwischen den 3 Bereichen:

- Model
- View
- Controller

Bei sehr kleinen Programmen ist dieses Prinzip nicht sehr nützlich, jedoch merkt man sehr früh, dass eine gewisse Unübersichtlichkeit und Unordnung entsteht. Diese Aufteilung, wird deswegen nicht bei allen Prototypen angewendet, aber bei jedem erklärt, in welchen Bereichen der Prototyp arbeitet.

1.4.1 Ordnerstruktur

Folgende übersichtliche Ordnerstruktur unterstützt das MVC Prinzip.

MVC-Projekt
- controllers
  - Controller Dateien
  - ..
- models
  - Model Dateien
  - …
- views
  - View Dateien
  - …
- server.js

1.4.2 Bedeutungen

M für Model: In diesem Bereich befinden sich sämtliche Abfragen, Funktionen, … zur Datenspeicherung und Datenauslesung.

V für View: Hier wird alles verwaltet bzw. erledigt, was der End-Nutzer auf seinem Bildschirm sehen kann.

C für Controller: Der Controller dient dazu, Model und View Bereich miteinander zu verbinden. Hier werden Nutzereingaben aus den Views angenommen und danach falls notwendig in den Models gespeichert oder bei Nutzeranfrage in den Views, Daten aus den Models geholt und dem Nutzer gezeigt. Der Controller fungiert also als Schnittstelle.

1.4.3 Kommunikation der einzelnen Bereiche

Folgende Grafik zeigt, wie die einzelnen Bereiche untereinander kommunizieren sollten:

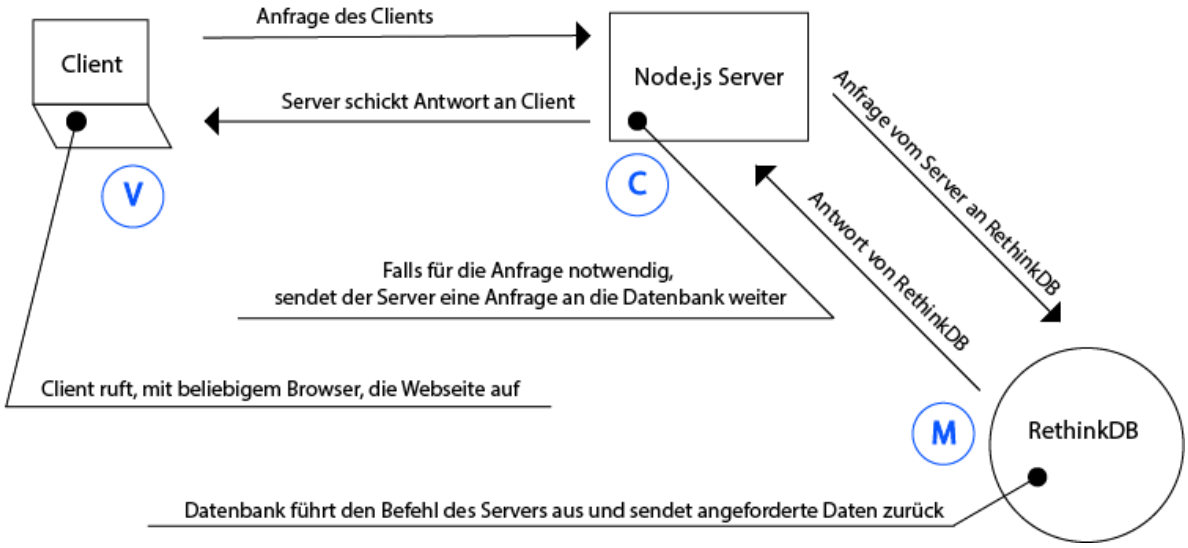

Abbildung 6: Kommunikation der einzelnen MVC-Bereiche

Beispielablauf: Der Client sendet ein Formular ab, mit seinem Namen und Alter. Diese Information soll in der Datenbank gespeichert werden und der Client erhält Rückmeldung, ob die Daten erfolgreich gespeichert wurden:

1. Am Client wird ein Formular mit 2 Feldern (Vorname und Alter) ausgefüllt und danach auf speichern gedrückt. Eine Anfrage wird vom Client an den Server gesendet.
2. Der Server bearbeitet die ankommende Nachricht und weil er Dateien in die Datenbank speichern muss, werden Vorname und Alter an RethinkDB übermittelt, mit dem Befehl, diese Daten zu speichern.
3. Die Datenbank nimmt die Anforderungen des Servers entgegen und führt diese aus. Am häufigsten genutzte Befehle an die Datenbank sind:
    - insert: Daten einfügen
    - delete: Bestimmte Daten löschen
    - update: Datensätze, die bereits in der Datenbank stehen, werden mit neuen Werten überschrieben
    - filter: Eine gefilterte Auslesung von Daten

    Nachdem die Anfrage vom Server fertiggestellt wurde, sendet die Datenbank eine Antwort an den Server zurück. Je nachdem, welcher Befehl ausgeführt wurde und ob dieser erfolgreich war, liefert die Datenbank andere Ergebnisse.
4. Der Server nimmt die Antwort der Datenbank entgegen. Falls die Speicherung der Daten erfolgreich war, liefert der Server eine dementsprechende Antwort an den Client. Wird ein Fehler zurückgeliefert, muss dem Benutzer dies ebenfalls mitgeteilt werden und im besten Fall, wird der Fehler in einer Log-Datei festgehalten, damit ein Programmierer den Fehler später beheben kann.

1.5 Typischer Aufbau von Node.js Projekten

MVC reicht noch nicht ausschließlich für große und saubere Node.js Projekte. Der nächste Screenshot zeigt den typischen Aufbau eines sauber strukturierten Projektes.

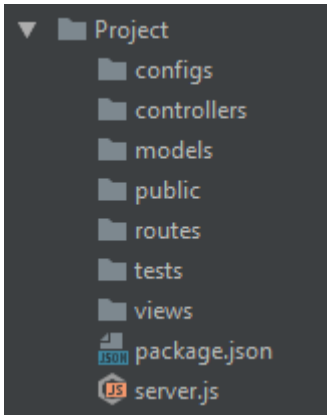

Abbildung 7: Typischer Aufbau von Node.js Projekten

1.5.1 configs

In diesem Ordner befinden sich Dokumente, die projektweit relevante Informationen enthalten (z.B.: Datenbankzugriffsdaten)

1.5.2 controllers, models & views

Diese Bereiche enthalten Dokumente nach dem MVC-Modell, welches oben genau erklärt wird.

1.5.3 public

Hier befinden sich alle statischen Dokumente die der Nutzer zum Aufruf der Webseite benötigt. Das sind Dateien wie CSS, JS, Favicon, usw. Diese Dateien werden als „öffentlich“ gesetzt, also für die Clients freigegeben.

1.5.4 routes

Hier befinden sich sämtliche Routingpfade, die der Client aufrufen kann (Wird im Prototyp „HelloWorld“ näher geklärt).

1.5.5 tests

In diesem Ordner lagert man Dateien für Durchführungen von Tests. Es gibt verschiedenste Testsysteme für Node.js Projekte und die wohl bekannteste Bibliothek dafür ist „**Mocha**“.

1.5.6 package.json

Diese Datei enthält verschiedenste Projektinformationen, sowie auch benötigte Module für die Ausführung des jeweiligen Projekts (Wird unter dem Punkt 1.1.3 Paket- bzw. Modulinstallation geklärt).

1.5.7 server.js

Das Hauptserverfile befindet sich direkt im Root-Verzeichnis. Die 3 beliebtesten Namen für dieses File sind:

5. app.js
6. index.js
7. server.js

Ziel des Hauptserverfiles ist es, nicht so viel Code wie möglich zu fassen, sondern sollte wenig Code beinhalten. Alles was sinnvoll ausgelagert werden könnte, sollte sich in externen Dateien befinden und das Hauptserverfile dient dann lediglich als Zusammenfassung aller Bestandteile des Projekts. Also wichtige Teilabschnitte werden gegliedert und in separaten Dateien ausprogrammiert, während im Hauptserverfile diese Abschnitte geladen und globale Einstellungen definiert werden.

1.6 Verfügbarkeit der Prototypen

Alle Prototypen sind unter diesem Link verfügbar:
https://github.com/danielroetzer/RVP-Repository/tree/master/Prototypes

Um die Prototypen zu testen, den gewünschten Prototyp zuerst herunterladen. Danach stellt man sicher, dass die nötigen Module für den jeweiligen Prototyp installiert sind, wie oben unter „Installation/Node Package Manager“ beschrieben wird.

2 Prototypen

2.1 HelloWorld

In diesem Programm wird erstmals, mithilfe von Express, ein Server erstellt und gestartet. Dieser Server soll lediglich im Browser „Hello World“ ausgeben.

2.1.1 Verwendete Pakete

Wie benötigte Pakete installiert werden, wird im Bereich „Installation/Node Package Manager“ genau beschrieben.

- Express

2.1.2 MVC

In diesem Prototyp würden wir uns nur in dem View-Teilbereich befinden, aber das Prinzip wird nicht angewendet, weil das Programm dazu viel zu kurz ist.

2.1.3 Ordnerstruktur

Folgende Abbildung zeigt die benötigten Dateien und Ordner für diesen Prototyp. Der Ordner „**node_modules“** wird bei Paketinstallationen immer automatisch erstellt.

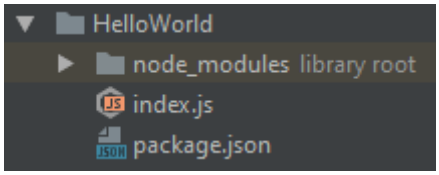

Abbildung 8: HelloWorld Ordnerstruktur

2.1.4 Express einbinden

Als erstes wird Express geladen und Initialisiert. Alle Funktionen von Express werden später mithilfe der Konstanten – **app** – aufgerufen.

```
//Calling express
const express = require('express');
const app = express();
```

Abbildung 9: Express in den Server einbinden

2.1.5 Browser Aufruf ermöglichen

Um den Server im Browser aufrufen zu können, befehlen wir ihm, auf einen bestimmten Port zu „horchen“.

```
//Setting server to listen to localhost:3000
//localhost = 127.0.0.1 is default
app.listen(3000, function () {
    //This text is displayed, when the server has been successfully started
    console.log('Hello World Server listening on localhost:3000!')
});
```

Abbildung 10: Server „horcht“ auf localhost:3000

2.1.6 Server starten

Um den Server zu starten, müssen wir in CMD den Befehl – **node servername.js** – ausführen. Klappt alles, wird unser vorgegebener Text ausgegeben, wie man folgend sehen kann.

```
C:\Users\Rötzer\Documents\GitHub\RVP-Repository\Prototypes\HelloWorld>node index.js
Hello World Server listening on localhost:3000!
```

Abbildung 11: Server starten mit - node index.js

Nun läuft der Server bereits, aber dennoch wird im Browser noch nichts angezeigt.

2.1.7 Hello World im Browser ausgeben

Die Seite wird mit der GET-Methode aufgerufen und diese muss zuerst definiert werden.

```
//reg = request, res = response
app.get('/', function (req, res) {
    //Display Hello World
    res.send('Hello World!')
});
```

Abbildung 12: Antwort an die GET-Anfrage programmieren

Startet man jetzt den Server und ruft ihn im Browser auf, wird – **Hello World!** – ausgegeben.

Das erste Element ist der Routing Pfad und wird in einfachen oder doppelten Hochkomma geschrieben. In der anonymen Funktion wird ein Text mithilfe des – **res** – Parameters gesendet.

Request = req: Dieses Objekt steht für die Anfrage des Clients. Hiermit können z.B.: gesendete Formulardaten verwendet werden (wird in späteren Prototypen gezeigt).

Response = res: Dieses Objekt steht für die Antwort an den Client. Einfacher Text oder ganze HTML Seiten (in den Prototypen Pug und BodyParser sichtbar) können hier gesendet werden.

2.1.7.1 Ein weiteres Beispiel eines anderen Routing Pfades

Man kann verschiedenste Routing Pfade definieren. Folgender Code zeigt, wie man unter Aufruf von – **localhost:3000/secret** – einen Text ausgeben kann.

```
app.get('/secret', function (req, res) {
    res.send('Shhhh! This is the secret place')
});
```

Abbildung 13: Anderen Routing-Pfad definieren

2.2 Modules

In diesem Beispiel wird gezeigt wie man eigene Module erstellt und anwendet. Diese können sehr nützlich sein und werden in jedem Node.js Projekt angewendet.

2.2.1 Verwendete Pakete

Keine Pakete verwendet.

2.2.2 MVC

Das MVC-Modell ist in diesem Prototyp nicht anwendbar, weil nur gezeigt wird, wie man eigene Module bzw. Pakete erstellen kann.

2.2.3 Ordnerstruktur

In „**config.js**“ befinden sich unsere selbst erstellten Module.

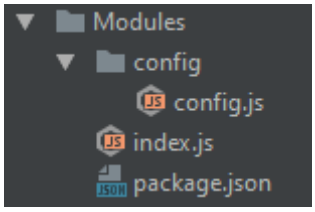

Abbildung 14: Modules Ordnerstruktur

2.2.4 Erstellung und Einbindung

Eigene Module erstellt man in separaten Files. Hierzu wird ein „**config.js**“ File im Ordner „**config**“ erstellt.

Um das File einzubinden, reicht diese Zeile, welche in der Server Datei gespeichert wird:

```
//Importing the modules from the config file
const config = require("./config/config");
```

Abbildung 15: Config-Datei einbinden

Alle exportierten Funktionen oder Werte sind nun in der Konstanten aufrufbar.

2.2.5 Config.js

Als Beispiel werden einfachste Funktionen und einfache Zeichenketten erstellt und exportiert.

```
//Exporting various modules in different ways
module.exports = {
    firstFunction : function () {
        return "Return of the first function";
    },secondFunction: function () {
        return "Return of the second function";
    },server: {
        "host":"localhost",
        "port":3000
    },values: {
        "name":"Daniel",
        "location":"Melk",
        "number":"06603448790"
    },
    //This method of calling the functions is possible with
    //the Javascript version: ECMAScript 6
    sayHello,
    sum
};
```

Abbildung 16: Einfache Funktionen und Informationen exportieren

Als erstes wird immer der Name der Funktion oder der Zeichenkette angeben. Mit diesen Namen, können alle Informationen später aufgerufen werden.

Eine neue Besonderheit der neuesten JavaScript Version ist, dass man nur den Namen angeben kann und später erst die Funktion definieren muss. Diese Schreibweise wird „Shorthand Methoden“ genannt. Dies sorgt für deutlich bessere Übersicht. Folgend werden die fehlenden 2 Funktionen abgebildet.

```
function sum(number1,number2){
    const result = number1 + number2;
    return "" + number1 + " + " + number2 + " = " + result;
}

function sayHello(){
    return "Hello, fellow user";
}
```

Abbildung 17: Exportierte Funktionen mithilfe der Shorthand-Methode

2.2.6 Anwendung der exportierten Funktionen und Zeichenketten

Unser exportiertes Modul ist bereits in unseren Server eingebunden. Folgend werden einfache Beispiele zur Anwendung unseres Moduls gezeigt.

```
//Call the defined functions
console.log(config.firstFunction());
console.log(config.secondFunction());
console.log(config.sayHello());
//Build sum
console.log(config.sum(3,5));

//Call the defined values
console.log(config.values);
console.log(config.values.name);
```

Abbildung 18: Anwendungsbeispiele unseres Moduls

Wird der Server gestartet, erhält man folgendes Ergebnis.

```
D:\GitHub\RVP-Repository\Prototypes\Modules>node index.js
Return of the first function
Return of the second function
Hello, fellow user
3 + 5 = 8
{ name: 'Daniel', location: 'Melk', number: '06603448790' }
Daniel
```

Abbildung 19: Ausgaben in der Konsole nach Serverstart

2.3 Formulare

Um nun auch einmal das „**Request-Objekt**“ anzuwenden, wurden 2 Prototypen erstellt, die sich mit Formularübergabe beschäftigen.

2.3.1 BodyParser

Mithilfe des „**body-parser**“ Paketes, können HTML Elemente in JavaScript Elemente umgewandelt werden. Dadurch können gesendete Formulardaten angenommen und bearbeitet werden. Das bedeutet: Die Anfragen des Clients werden so umgewandelt, dass diese im „**Request-Objekt**“ anwendbar und lesbar sind für den Server.

2.3.1.1 Verwendete Pakete

Die Verwendung des Express Pakets, wird im Prototyp „HelloWorld“ gezeigt.

- express
- body-parser

2.3.1.2 MVC

Hier wird zum ersten Mal ein Teilbereich, nämlich der View Bereich, ausgenutzt. Die anderen beiden Bereiche werden noch nicht benötigt.

2.3.1.3 Ordnerstruktur

In der HTML Datei befindet sich ein kleines Formular, das zur Datenübertragung genutzt wird.

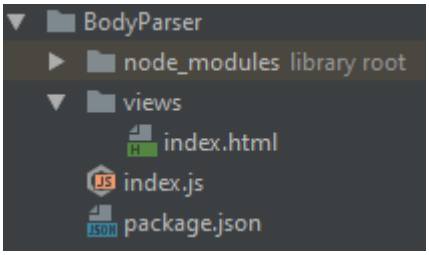

Abbildung 20: BodyParser Ordnerstruktur

2.3.1.4 Einbindung und Anwendung

Das Paket wird eingebunden, wie jedes andere. Wie gewohnt, ist Express in der Konstanten „**app**“ initialisiert und über diese wird mitgeteilt, dass das Modul angewendet werden soll.

```
//Load body-parser module and initialize it
const bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({extended: false}));
```

Abbildung 21: body-parser einbinden und anwenden

Als nächstes wird der Ordner festgelegt, indem sich die statischen HTML, CSS, usw. Dateien befinden.

```
//Defines the route, in which the static files like html,css, etc. are found
//Calling localhost:3000 will open the index.html in the view directory
app.use(express.static(__dirname + '/views'));
```

Abbildung 22: Ordner für die statischen Dateien festlegen

In dem Ordner – **views** – befindet sich ein einfaches HTML File mit einem Formular. Wichtig ist, dass dieses unter – **index.html** – gespeichert wird.

Folgend wird das Formular der HTML Datei abgebildet.

```
<form action="/" method="post">
    <input type="text" name="userName" placeholder="Enter your name" />
    <br><br>
    <button type="submit">Submit</button>
</form>
```

Abbildung 23: Inhalt von index.html

Äußerst wichtig ist der Pfad, der in – **action** – angegeben wird. Hier können verschiedenste Routing Pfade gewählt werden, nur muss man später den richtigen anwenden.

2.3.1.5 Formulardaten empfangen und zurücksenden

Die Daten werden mittels POST-Methode gesendet, also muss die zugehörige Express Funktion erstellt werden.

```
app.post('/',function (req,res) {
    //Get variable from the form
    const userName = req.body.userName;
    //Create a little html, with the variable included
    const html = 'Hello: ' + userName + '.<br>' +
                 '<a href="/">Try again.</a>';

    //Send it
    res.send(html);
});
```

Abbildung 24: POST-Methode definieren

Der erste Paramater ist der Pfad, der beim Formular unter – **action** – angegeben wurde. Jetzt kommt zum ersten Mal das „**Request-Objekt**“ zum Einsatz. Mithilfe dessen, können die Formulardaten angenommen werden. Anschließend wird ein wenig HTML erzeugt und zuletzt erneut an den Client gesendet.

Wie man hier bereits sieht, ist die Aufbereitung der Antwort, indem man HTML hier in JavaScript schreibt, sehr unschön. Ablösung hierfür sorgt „**Pug**“ (später erklärt).

2.3.1.6 Ausführung

Folgende 2 Screenshots zeigen den ersten Aufruf des Servers und nebenbei den zurückgesendeten Text, nach Formulareinsendung.

Abbildung 25: Formulareingabe

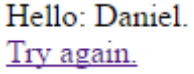

Abbildung 26: Ausgabe des übergebenen Wertes

2.3.2 Pug

Mithilfe von „**Pug**“ können Formulardaten optimal wieder an den Client gesendet werden. Im Prototyp „**Body-Parser**“, wurde HTML im Server erzeugt, was natürlich nicht optimal ist und das ist jetzt nicht mehr nötig. Jedoch hat „**Pug**“ nicht nur diese Funktion, es verändert die gesamte Schreibweise von HTML Seiten und fügt weitere Funktionen wie Schleifen hinzu.

Es ist aber trotzdem immer noch das „**body-parser**“ Modul nötig, damit der Server die Anfragen des Clients bearbeiten kann.

Hinweis: Jade ist exakt das gleiche wie Pug, jedoch wurde es, aus rechtlichen Gründen, von Jade zu Pug unbenannt.

2.3.2.1 Verwendete Pakete

Die Verwendung des Body-Parser Pakets, wird im Prototyp „BodyParser“ gezeigt und Express wird im Prototyp „HelloWorld“ gezeigt.

- express
- body-parser
- pug

2.3.2.2 MVC

Genauso wie beim vorherigen Prototyp, wird hier auch nur der View-Teilbereich angewendet.

2.3.2.3 Ordnerstruktur

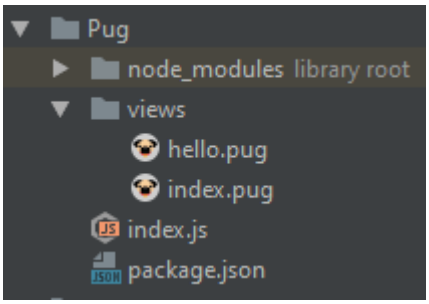

Abbildung 27: Pug Ordnerstruktur

2.3.2.4 Pug Syntax

Pug ist eine Template Engine mit hoher Performance. Implementiert ist dieses Template in JavaScript für Node.js und Browser. Vom Prinzip ist Pug das gleiche wie HTML, nur ist die Syntax anders und das schlichte HTML wird um einige Funktionen erweitert.

In HTML ist es typisch, dass jeder „Start-Tag“ auch ein „Ende-Tag“ besitzt. Hingegen in Pug wird nur das „Start-Tag“ geschrieben und die fehlenden „Ende-Tags“ werden beim Umwandeln zu HTML ergänzt. Das funktioniert, weil Pug „**whitespace sensitive**“ ist, was bedeutet, dass Pug die Zusammenhänge mit Tabulator Einrückungen erkennt. Die folgenden 2 Screenshots stellen dies dar.

```
doctype html
html(lang='en')
    head
        title Pug vs HTML
        link(href='bootstrap/css/bootstrap.min.css', rel='stylesheet', type='text/css')
    body
        h1.header This shows the differences between Pug and HTML.
        #container.
            Div-Container with id='container'
        .container.
            Div-Container with class='container'
```

Abbildung 28: Einfache Pug Seite

Wird zu:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Pug vs HTML</title>
        <link href="styles.css" rel="stylesheet" type="text/css">
    </head>
    <body>
        <h1 class="header">
            This shows the differences between Pug and HTML.
        </h1>
        <div id="container">
            Div-Container with id='container'
        </div>
        <div class="container">
            Div-Container with class='container'
        </div>
    </body>
</html>
```

Abbildung 29: Einfache HTML Seite

2.3.2.5 Einbindung

Die Einbindung unterscheidet sich von allen anderen. Es muss die „**view engine**“ auf „**pug**“ umgestellt werden, sodass die erstellten Seiten richtig angezeigt werden.

```
//Load pug
app.set('view engine', 'pug');
```

Abbildung 30: view engine auf pug setzen

Die Pug Dateien müssen im Ordner – **views** – erstellt werden, ansonsten kommt es zu Fehlern. Die Dateibenennung hingegen, ist nicht so wichtig, nur die Dateiendung muss – **.pug** – sein.

2.3.2.6 index.pug

Folgende Zeilen zeigen den Inhalt der erstellten Datei, welche als Startseite dient. Die beiden Variablen „**title**“ und „**message**“ werden durch mitgesendete Werte ersetzt.

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

        form(action='/', method='POST')
            input(type='text', name='userName', placeholder='Enter your name')
            br
            br
            input(type='submit', value='Submit')
```

Abbildung 31: Inhalt von index.pug

2.3.2.7 hello.pug

Diese Seite wird aufgerufen, nachdem man die Formulardaten mit Buttonklick gesendet hat. Merke wie 2 Verschiedene Arten der Variablenzuweisung genutzt wird. Die zweite ist sehr nützlich in Fließtext, da dieser so nicht unterbrochen werden muss.

```
html
    head
        title= title
    body
        p.
            Hello: #{userName}
        a(href='/') Try again
```

Abbildung 32: Inhalt von hello.pug

2.3.2.8 Im Browser aufrufen

Um das Beispiel im Browser aufzurufen, hilft uns erneut Express. Dieses Mal jedoch, wird – **.render** – anstatt – **.send** – verwendet. Der erste Wert hierbei, ist die Pug Datei, die dargestellt werden soll. Außerdem werden hier die Parameter „**title**“ und „**message**“ übergeben.

```
//Render the index file and send some arguments
app.get('/', function (req, res) {
    res.render('index', {
        title: 'Pug is great!',
        message: 'Hello there! I am using pug'
    });
});
```

Abbildung 33: Antwort an die GET-Anfrage programmieren

Nun muss noch die POST-Methode definiert werden, die die Formulardaten annimmt und eine neue Seite öffnet.

```
//Retrieve submitted value and send it back
app.post('/',function (req,res) {
    res.render('hello', {
        title: 'Pug is great!',
        userName: req.body.userName
    })
});
```

Abbildung 34: Formulardaten mittels POST-Methode annehmen und danach ausgeben

2.3.2.9 Ausführung

Das Ergebnis sollte nun das gleiche sein wie beim Prototyp „**Body-Parser**“. Lediglich die Überschrift unterscheidet die 2 Prototypen nun am Ende. Dies zeigen folgende 2 Screenshots.

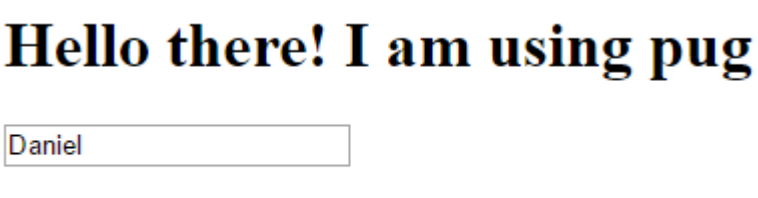

Abbildung 35: Formulareingabe

Abbildung 36: Ausgabe des Übergebenen Wertes

2.4 Logging

Als Logging bezeichnet man die Protokollierung verschiedenster Informationen bzw. Daten. Angefangen von Konsolenausgaben, wenn der Server gestartet wird, bis hin zu Fehlern und Serverabstürzen oder sogar Nutzerzugriffe.

2.4.1 Warum Logging anwenden?

Entwickelte Systeme müssen laufend überwacht, kontrolliert und verbessert werden. Aber wie erfährt der zuständige Programmierer nach der Entwicklungszeit, wie sein System läuft, wenn es tausende Benutzer gleichzeitig in Verwendung haben? Hierzu dient die Protokollierung wichtiger Ereignisse in den Log-Files, vor allem Fehlermeldungen sind ausschlaggebend.

In der Entwicklungszeit, sitzt der Programmierer ständig vor seinem System, testet dieses und erhält direkt die Ausgaben im Programmierprogramm oder in einer Konsole. Aber nach Veröffentlichung seines Systems, funktioniert dies nicht mehr so, denn er kann sein Produkt nicht täglich 24 Stunden überwachen und gleichzeitig noch schnell ausbessern. Deshalb ist die Protokollierung äußerst wichtig, um später sein Produkt weiter überwachen und verbessern zu können.

Würde man alle ungefilterten Meldungen direkt im Browser ausgeben, so würden die meisten alltäglichen Benutzer nichts mit dieser Information anfangen. Außerdem sind Fehlermeldungen eine potentiell angreifbare Stelle für Hacker, weil diese Personen durch diese Meldungen, einfache Schwachstellen im System entdecken können.

2.4.2 Winston

Winston ist eine universelle Protokollierungsbibliothek, mit der Möglichkeit, mehrere Transportmethoden anzuwenden. Das bedeutet, dass bestimmte Dinge nur in der Konsole ausgegeben werden können und andere in einem externen File oder sogar in einer Datenbank gespeichert werden können, und das Gleichzeitig.

2.4.2.1 Verwendete Pakete

- winston

2.4.2.2 MVC

Das Prinzip wird nicht angewendet, weil in diesem Prototyp kein Bereich einsetzbar ist. Es wird lediglich die Logging Funktionen von Winston gezeigt und erklärt.

2.4.2.3 Ordnerstruktur

In „**winston.js**“ befindet sich der erstellte Logger und im Ordner „**logs**“ alle externen Log-Dateien.

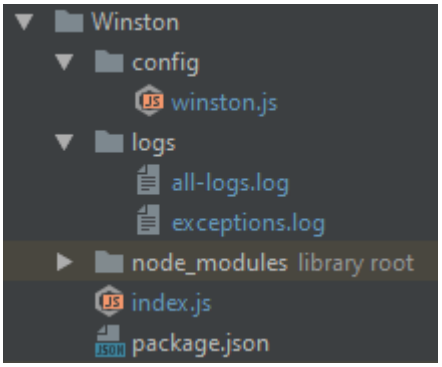

Abbildung 37: Winston Ordnerstruktur

2.4.2.4 Logging-Levels

Die Levels (Ebenen) bei Winston, geben Informationen über die Art des Protokolleintrages. Folgende 6 vordefinierte Levels sind vorhanden (Reihenfolge ist bedeutend):

- silly
- debug
- verbose
- info
- warn
- error

Der Log-Level, den man bei den Transporten definiert, gibt an welche Logs geschrieben werden. Verwendet man den Level „verbose“, dann werden alle Levels von „verbose“ bis hinunter zu „error“ geschrieben. Verwendet man „info“, so werden alle Levels beginnend von „info“ geschrieben. Also alle höheren Levels, inkludieren die Levels darunter (aus der oberen Aufzählung ablesbar).

2.4.2.5 Erstellung eines neuen Loggers

Da Logger ständig über das gesamte Projekt verteilt verwendet werden, Lagern wir diesen in den **config** – Ordner in die Datei **winston.js** aus.

Wie gewohnt, wird als erstes das Winston-Modul eingebunden.

```
//Load Winston logger
//=====================================
const winston = require('winston');
```

Abbildung 38: Winston einbinden

Danach erstellen wir auf Basis des Winston Moduls, einen neuen Logger:

```
//Initialize our own logger, based on winston
//=====================================================
const logger = new winston.Logger({
    transports: [
        new winston.transports.File({
            level: 'info',
            filename: './logs/all-logs.log',
            json: true,
            timestamp: new Date(),
            maxsize: 5242880, //5MB
            colorize: false
        }),
        new winston.transports.Console({
            level: 'silly',
            handleExceptions: true,
            json: false,
            timestamp: new Date(),
            colorize: true
        })
    ],
    exceptionHandlers: [
        new winston.transports.File({
            filename: 'logs/exceptions.log',
            json: false
        })
    ]
});
```

Abbildung 39: Logger erstellen

In der Eigenschaft **transports**, befinden sich die verschiedenen Transporte. Der erste definiert die Ausgabe in ein externes File und der zweite die Ausgabe in der Konsole. Darunter befindet sich die Exception-Handlers, welche aufgerufen werden, wenn sogenannte „**uncaughtExceptions**“ (=unbehandelte Fehler) auftreten.

2.4.2.5.1 Definierbare Eigenschaften der Transporte:

- **level**: Gibt den Log-Level des Transportes an
- **filename**: Dateiname, falls in externes File geschrieben wird
- **json**: Ist dieser Wert auf true gesetzt, so wird die Ausgabe in JSON formatiert
- **timestamp**: Zeitstempel hinzufügen

- **maxsize**: Wird diese Angabe überschritten, so wird automatisch ein neues File erstellt
- **maxfiles**: Diese Anzahl gibt an wie viele Dateien dieses Transportes erlaubt sind. Wird diese Zahl überschritten, so wird automatisch die älteste Datei gelöscht
- **colorize**: Bei true wird die Ausgabe farblich gestaltet

2.4.2.5.2 Weitere Eigenschaften des Loggers

Zusätzliche Eigenschaften können außerhalb der Initialisierung angewendet werden mithilfe der erstellten Konstanten „**logger“**.

Der folgende Screenshot zeigt, wie die Fehlermeldungsausgabe eingeschaltet wird und was passiert. Außerdem wird eingestellt, das unser Programm geschlossen wird, falls eine „**uncaughtException**“ auftritt.

```
//Emit errors
logger.emitErrs = true;

//Exit after logging an uncaughtException (default=true)
logger.exitOnError = true;
```

Abbildung 40: Weitere Einstellungen definieren

2.4.2.5.3 Export unseres Loggers

Zuletzt muss unser erstellter Logger noch exportiert werden, damit er Projektweit eingebunden und angewendet werden kann.

```
module.exports = logger;
```

Abbildung 41: Logger exportieren

2.4.2.6 Einbindung und Anwendung in unserem Server

Um unseren erstellten Logger anwenden zu können, muss er zuerst eingebunden werden.

```
//Load the defined logger from the configs
//=========================================================
const logger = require('./config/winston');
```

Abbildung 42: Logger einbinden

Nun geben schreiben wir verschiedenste Logs, mit allen möglichen Log-Levels.

```
//Logging different sample messages
//=========================================================
logger.silly('Logging level = silly');
logger.debug('Logging level = debug');
logger.verbose('Logging level = verbose');
logger.info('Logging level = info');
logger.warn('Logging level = warn');
logger.error('Logging level = error');
```

Abbildung 43: Logging Anwendungsbeispiele

Die Anwendung kann auch mit anderer Schreibweise erfolgen und dennoch wird dasselbe Ergebnis erreicht.

```
logger.log('silly','Logging level = silly');
logger.log('debug','Logging level = debug');
logger.log('verbose','Logging level = verbose');
logger.log('info','Logging level = info');
logger.log('warn','Logging level = warn');
logger.log('error','Logging level = error');
```

Abbildung 44: Andere Schreibweise für Logs

2.4.2.7 Ausführung

Bei Serverstart, werden nun die erstellten Logs ausgeführt. Der Log-Level, der bei den Transporten definiert wurde, bestimmt jetzt, wo dies Logs überall geschrieben werden.

Für die Ausgabe in der Konsole, wurde der Level „**silly**“ angegeben. Daher werden alle Logs in der Konsole angezeigt, die wir vorhin programmierten.

```
C:\Users\Rötzer\Documents\GitHub\RVP-Repository\Prototypes\Winston>node index.js
2016-11-30T09:22:49.359Z - silly: Logging level = silly
2016-11-30T09:22:49.363Z - debug: Logging level = debug
2016-11-30T09:22:49.366Z - verbose: Logging level = verbose
2016-11-30T09:22:49.370Z - info: Logging level = info
2016-11-30T09:22:49.370Z - warn: Logging level = warn
2016-11-30T09:22:49.371Z - error: Logging level = error
```

Abbildung 45: Logging Ausgabe in der Konsole

Wird der Log-Level also heruntergesetzt, werden nicht mehr alle Logs angezeigt. Folgend wird der Level von „**silly**“ auf „**info**“ gesetzt.

```
new winston.transports.Console({
    level: 'info',
    handleExceptions: true,
    json: false,
    timestamp: new Date(),
    colorize: true
})
```

Abbildung 46: Ändern des Log-Levels der Konsole

Bei erneuten Serverstart, werden nur noch die Logs ab „info“ ausgegeben, obwohl an der Ausgabe der Logs selbst nichts geändert wurde.

```
C:\Users\Rötzer\Documents\GitHub\RVP-Repository\Prototypes\Winston>node index.js
2016-11-30T09:43:21.229Z - info: Logging level = info
2016-11-30T09:43:21.229Z - warn: Logging level = warn
2016-11-30T09:43:21.229Z - error: Logging level = error
```

Abbildung 47: Neue Logging Ausgabe in der Konsole

Dieses Prinzip wird bei allen definierten Transporten angewendet, so kann man verschiedene Ausgaben für bestimmte Ereignisse, zu bestimmten Zeitpunkten erzeugen. Normalerweise will man nämlich nicht alles in externe Files auslagern, sondern nur die wichtigeren Ereignisse.

Der nächste Screenshot zeigt, dass in unserem Server absichtlich ein einfacher Fehler geschrieben wird. Dieser Fehler ist eine „**uncaught Exception**“, also ein nicht abgefangener Fehler. Einer Konstanten darf nur ein einziges Mal ein Wert zugewiesen werden und danach nicht mehr verändert werden.

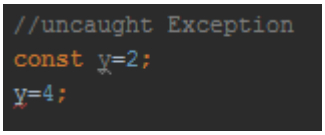

Abbildung 48: Nicht abgefangener Fehler

Beim Serverstart, stürzt das Programm ab und der Fehler wird in der definierten Datei „**exceptions.log**“ protokolliert.

Folgender Screenshot zeigt einen kleinen Abschnitt, der protokollierten Fehlermeldung.

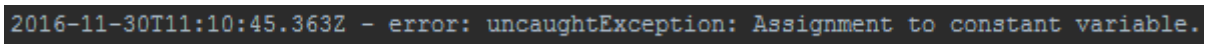

Abbildung 49: Protokollierte Fehlermeldung

Mit solchen Einträgen, kann ein Programmierer gut Fehlerbehebungen vornehmen.

2.4.3 Morgan

Dieses Modul dient auch zur Protokollierung, aber in einer anderen Art als Winston. Morgan registriert und protokolliert alle http-Zugriffe auf den Server, also alles, was Benutzer auf der Webseite im Browser anklicken.

Mit dieser Information kann man feststellen, welche Seiten bzw. Artikel einer Homepage am meisten besucht werden und welche eher unbeliebter bei den Nutzern sind.

Weitere Informationen über den Benutzer, die protokolliert werden:

- Datum + Zeitpunkt des Zugriffs
- Verwendeter Browser
- Browserversion
- Installiertes Betriebssystem seines Computers
- Verwendete http-Methode (POST oder GET)

2.4.3.1 Verwendete Pakete

Die Verwendung des Express Pakets, wird im Prototyp „HelloWorld“ gezeigt.

- express
- morgan
- fs

Das Paket „**fs**“ steht für File System und ist nicht herunterzuladen, sondern wird bereits in den Standardpaketen von Node.js mitgeliefert und wird benötigt, um Daten in ein externes File zu schreiben. Bei Winston ist dies nicht nötig, denn dieses Modul kann das eigenständig.

2.4.3.2 MVC

In diesem Prototyp könnte der View-Teilbereich angewendet werden, da Morgan zuständig ist um Nutzerzugriffe am Client zu protokollieren, jedoch ist das Project noch zu klein um einen Nutzen daraus zu erlangen.

2.4.3.3 Ordnerstruktur

Die Log-Einträge werden in die „**access.log**“ Datei ausgelagert.

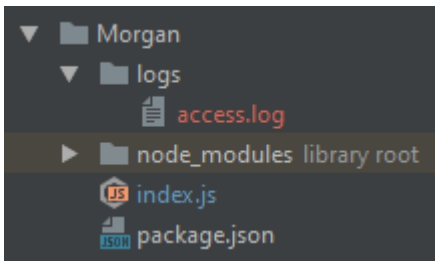

Abbildung 50: Morgan Ordnerstruktur

2.4.3.4 Einbindung und Anwendung

Dieser Logger wird nicht ausgelagert in eine Konfigurationsdatei, wie Winston, sondern direkt in der Hauptserverdatei eingebunden und initialisiert. Das hat den Grund, weil Morgan nicht Systemweit verwendbar sein muss, sondern nur einmal initialisiert wird.

Die Pakete werden so wie die meisten eingebunden.

```
//Load morgan and fs
const morgan = require('morgan');
const fs = require('fs');
```

Anschließend wird der Schreibvorgang in ein spezifisches File definiert und danach wird Morgan mit dem Schreibvorgang, zu Express zugewiesen.

```
//Create write stream, in append mode
const accessLogStream = fs.createWriteStream(__dirname + '/logs/access.log', {flags: 'a'});
//Get express to use morgan
app.use(morgan('combined', {stream:accessLogStream}));
```

Abbildung 51: writeStream erstellen und Morgan anwenden

Beim Erstellen des Schreibvorganges unter „createWriteStream“, sind 2 Parameter vorhanden.

Der erste gibt den Pfadnamen zur externen Datei an, wobei „**__dirname**“ für den absoluten Pfad steht, zu dem noch der spezifische Ort hinzugefügt wird.

Der zweite Parameter bestimmt, dass jeder Text, der in die Log-Datei geschrieben wird, angehängt wird. Somit löscht nicht jeder Log-Eintrag den vorherigen, sondern weitere Einträge werden hinzugefügt.

2.4.3.5 Test der http-Protokollierung

Um die Funktion des Morgan Moduls nun testen zu können, werden verschiedene Routing-Pfade mit Express erstellt. Folgendes Bild zeigt drei verschiedene Pfade, in denen jeweils der Link zu einem anderen definierten Pfad, an den Browser gesendet wird.

```
app.get('/', function (req, res) {
    res.send('<a href="/page1">Go to /page1</a>')
});

app.get('/page1', function (req, res) {
    res.send('<a href="/page2">Go to /page2</a>')
});

app.get('/page2', function (req, res) {
    res.send('<a href="/">Go to /</a>')
});
```

Abbildung 52: 3 definierte Routing-Pfade

Zuletzt startet man den Server und ruft im Browser unseren Server auf. Mithilfe der Links, kann man jetzt schnell zwischen den verschiedenen Pfaden wechseln und dies wird alles protokolliert.

Folgend wird ein kleiner Ausschnitt der resultierenden Log-Datei abgebildet. Hier kann man das Datum und den Zeitpunkt des Aufrufs ablesen, das die GET-Methode verwendet wurde und die gelb hinterlegten stellen zeigen die Pfade, die aufgerufen wurden.

```
[30/Nov/2016:11:57:42 +0000] "GET / HTTP/1.1" 304 -
[30/Nov/2016:11:57:43 +0000] "GET /page1 HTTP/1.1"
[30/Nov/2016:11:57:45 +0000] "GET /page2 HTTP/1.1"
```

Abbildung 53: Ausschnitt der Log-Datei

2.5 RethinkDB

In diesem Beispiel werden eine Datenbank und eine zugehörige Tabelle erzeugt. Danach werden Testdaten eingefügt.

2.5.1 Verwendete Pakete

- async
- rethinkdb

2.5.2 MVC

Der Model-Teilbereich wird für die Datenbankabfragen genutzt. Die anderen beiden Bereiche werden noch nicht benötigt.

2.5.3 Ordnerstruktur

Allgemeine Informationen zur Datenbankanbindung werden in der Konfigurationsdatei gespeichert. Die Datenbankabfragen werden in „**db.js**“ ausgelagert.

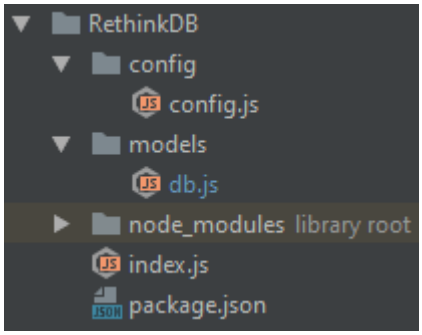

Abbildung 54: RethinkDB Ordnerstruktur

2.5.4 Erstellung und Einbindung

Im „**config.js**“ File im Ordner „**config**“ befinden sich notwendige Informationen zur Datenbankanbindung.

```
module.exports = {
    rethinkdb: {
        host: 'localhost',
        port: 28015,
        authKey: '',
        db: 'rethinkdb_ex'
    }
};
```

Abbildung 55: Datenbankinformationen

Die Datenbankabfragen werden in der Datei „**db.js**“ im Ordner „**models**“ erstellt.

In dieser JavaScript Datei wird unsere config.js Datei eingebunden und die 2 benötigten Module.

```
//Load config
const config = require('../config/config');

//Load the async module
const async = require('async');

//Load the RethinkDB module
const r = require('rethinkdb');
```

Abbildung 56: Benötigte Module laden

2.5.5 Datenbankabfragen

Nach dem einbinden der Module, werden in der Datei – **db.js** – 2 Funktionen definiert und exportiert. Wie die Namen verraten, Initialisiert die erste Funktion die Datenbank und die zweite fügt einfache Testdaten hinzu.

```
//Exporting the database query functions
module.exports = {
    initDB,
    insertTestData
};
```

Abbildung 57: Export von 2 Datenbankabfragen

2.5.5.1 initDB

Zweck dieser Funktion ist es, zu prüfen ob die Datenbank bereits vorhanden ist und ob der Table bereits vorhanden ist. Ist dies nicht der Fall, wird beides erstellt.

Hier kommt die Funktion – **async.waterfall** – zum Einsatz. In dieser werden nacheinander, beliebig viele anonyme Funktionen ausgeführt. Am Ende jeder Funktion wird die Variable, in der die Datenbankverbindung gespeichert ist, mithilfe der – **callback** – Funktion an die nächste Funktion weitergegeben. Tritt in irgendeiner Weise, in den anonymen Funktionen, ein Fehler auf, so wird an dieser Stelle abgebrochen und die Fehlermeldung ausgegeben.

```
//Initialize DB -> Called one time on server start
//async.waterfall([function(){},function(){},...) executes all defined function in a row
//The connection variable must be given to the other functions with callback()
function initDB(){
    async.waterfall([
        function (callback) {
            r.connect(config.rethinkdb, function (err, connection) {
                if (err){
                    console.log('Failed to connect to the database');
                    //throw err;
                }
                callback(null,connection);
            });
        },function (connection, callback) {
            r.dbCreate('rethinkdb_prototype').run(connection, function(err, result){
                if(err) {
                    console.log("Database already created");
                } else {
                    console.log("Created new database: rethinkdb_prototype");
                    console.log(JSON.stringify(result, null, 2));
                }

                callback(null, connection);
            });
        },function (connection, callback) {
            r.db('rethinkdb_prototype').tableCreate('protoTable').run(connection, function(err, result) {
                if (err) {
                    console.log("Table already created");
                }else{
                    console.log("Created new table: protoTable");
                    console.log(JSON.stringify(result, null, 2));
                }

                callback(null,'###### Database is ready ######');
            });
        }
    ],function (err, status) {
        if (err) throw err;
        else console.log(status);
    });
}
```

Abbildung 58: Sicherstellen, dass Datenbank und Table erstellt sind

Aufgaben der 3 anonymen Funktionen:

1. Verbindung zur Datenbank herstellen und weitergeben
2. Datenbank erstellen, falls diese nicht vorhanden ist
3. Table erstellen, falls dieser nicht vorhanden ist

2.5.5.2 insertTestData

Diese Funktion fügt einen String und eine Zahl der Datenbank hinzu. Vom Prinzip her ist diese Funktion dasselbe wie die Vorherige. Als erstes muss wieder die Datenbankverbindung hergestellt werden und wird dann an die nächsten Funktionen weitergegeben.

```
function insertTestData(){
    async.waterfall([
        function (callback) {
            r.connect(config.rethinkdb, function (err, connection) {
                if (err){
                    console.log('Failed to connect to the database');
                    //throw err;
                }
                callback(null,connection);
            });
        },function (connection, callback) {
            r.db('rethinkdb_prototype').table('protoTable').insert({
                name: 'Daniel',
                age: 20,
            }).run(connection, function(err, result) {
                if (err) throw err;
                console.log(JSON.stringify(result, null, 2));
            });
            callback(null, '++ test data successfully added')
        }
    ],function (err, status) {
        if (err) throw err;
        else console.log(status);
    });
}
```

Abbildung 59: Einfache Testdaten einfügen

Aufgaben der 2 anonymen Funktionen:

1. Verbindung zur Datenbank herstellen und diese weitergeben
2. Testdaten einfügen
    a. name: Daniel
    b. age: 20

2.5.6 Server

Das Server File ist fast leer, lediglich die Datenbankabfragen werden geladen und die Funktionen ausgeführt. Aufgrund des asynchronen Models, kann es sein, dass versucht wird die Testdaten einzufügen, bevor die Datenbank oder der Table erstellt wurde. Deswegen lässt man die zweite Funktion ein wenig warten, was in diesem Fall 2 Sekunden sind.

```
//Load Database file
const db = require('./models/db.js');

db.initDB();
setTimeout(db.insertTestData, 2000);
```

Abbildung 60: Exportierte Datenbankabfragen mit Zeitverzögerung ausführen

Natürlich ist dies nicht optimal, weil das System dadurch für 2 Sekunden blockiert wird, aber das wird nur beim Serverstart ausgeführt und solange der Server weiterläuft, wird das System nicht weiter blockiert.

2.5.7 Ausführung

Hinweis: Vergiss nicht RethinkDB zu starten, wie in der Vorbereitung erklärt wird.

Der nächste Screenshot zeigt die Ausgabe in der Konsole nach zweiten Serveraufruf. Also die Datenbank und der zugehörige Table wurden bereits erstellt, weshalb diese nicht mehr erzeugt werden. Danach sieht man, dass die Testdaten erfolgreich eingetragen wurden.

```
C:\Users\Rötzer\Documents\GitHub\RVP-Repository\Prototypes\RethinkDB>node index.js
Database already created
Table already created
###### Database is ready ######
++ test data successfully added
{
  "deleted": 0,
  "errors": 0,
  "generated_keys": [
    "98dafc5a-e2a2-4829-8a14-07442838612c"
  ],
  "inserted": 1,
  "replaced": 0,
  "skipped": 0,
  "unchanged": 0
}
```

Abbildung 61: Ergebnisse der ausgeführten Datenbankabfragen

2.6 Final

In diesem letzten finalen Prototyp, werden alle vorherigen Prototypen kombiniert. Als Ergebnis kann man im Browser ein Formular absenden und die Daten werden in die Datenbank gespeichert.

2.6.1 Verwendete Pakete

Alle verwendeten Module werden in vorhergehende Prototypen angewendet und erklärt.

- async
- body-parser
- express
- morgan
- pug
- rethinkdb
- winston

2.6.2 MVC

Hier wird das erste Mal das MVC-Prinzip vollständig ausgeschöpft und effektiv angewendet. In vorherigen Prototypen wird MVC immer nur Teilweise oder gar nicht angewendet, weil die Programme viel zu kurz sind. Aber jetzt ergibt sich bereits ein umfangreicheres Projekt, indem alles Vorherige kombiniert wird.

2.6.3 Ordnerstruktur

Dieser letzte Prototyp ist schon etwas umfangreicher, daher ergibt sich auch schon eine breitere Ordnerstruktur. Folgend wird nochmals der Inhalt der einzelnen Ordner bzw. Dateien, kurz geschildert.

- **config:** Projektweit relevante Informationen wie
  - Datenbankanbindung
  - Server Host + Server Port
  - Winston Logger
- **controller:** Beinhaltet die Funktionen, die Models und Views miteinander verbinden.
- **logs:** Alle Logs, die in externe Files geschrieben werden, befinden sich hier.
- **models:** Datenbankabfragen und Datenbankanbindung
- **node_modules:** Installierte Module
- **views:** Hier befinden sich die Webseiten, die der Benutzer sieht.
- **index.js:** Hauptserverfile
- **routes:** Enthält die Routingpfade, die der Benutzer im Browser aufrufen kann und ruft Funktionen aus dem Controller auf.

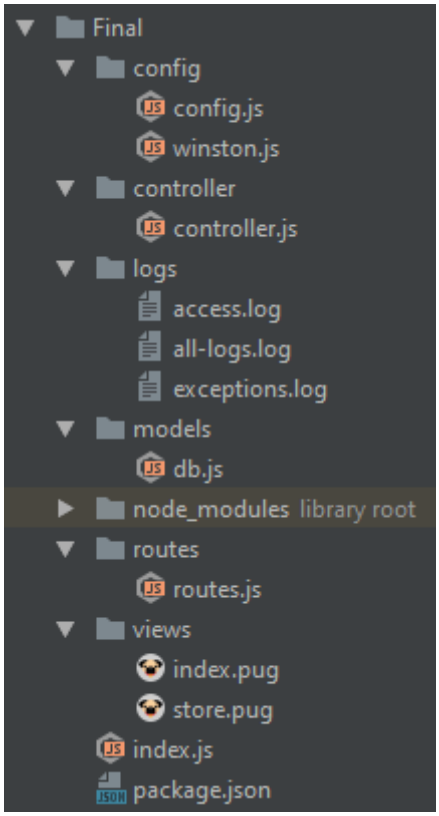

Abbildung 62: Final Ordnerstruktur

2.6.4 Config

In „**config.js**“ befinden sich die Datenbankanbindungsinformationen, sowie Server Host und Server Port.

```
//Export project wide relevant information
module.exports = {
    rethinkdb: {
        host: 'localhost',
        port: 28015,
        authKey: '',
        db: 'rethinkdb_ex'
    },
    server: {
        host: "localhost",
        port: 3000
    }
};
```

Abbildung 63: Inhalt von config.js

Der Winston Logger wird in „**winston.js**“ erstellt.

```
const winston = require('winston');

//Initialize our own logger, based on winston
//=======================================================
const logger = new winston.Logger({
    transports: [
        new winston.transports.File({
            level: 'info',
            filename: './logs/all-logs.log',
            json: true,
            timestamp: new Date(),
            maxsize: 5242880, //5MB
            colorize: false
        }),
        new winston.transports.Console({
            level: 'silly',
            handleExceptions: true,
            json: false,
            timestamp: new Date(),
            colorize: true
        })
    ],
    exceptionHandlers: [
        new winston.transports.File({
            filename: 'logs/exceptions.log',
            json: false
        })
    ]
});

//Emit errors
logger.emitErrs = true;

module.exports = logger;
```

Abbildung 64: Inhalt von winston.js

2.6.5 Controller

Der Controller regelt die Verbindung zwischen den Nutzereingaben und der Datenspeicherung in der Datenbank. Die dafür programmierten Funktionen werden exportiert, sodass diese in den „**routes**“ (Erklärung folgt im Punkt 2.6.6 Routes) aufrufbar sind.

```
const db = require('../models/db');
module.exports = {
    indexAction,
    storeAction
};

//Call index page
function indexAction (req, res) {
    res.render('index', {
        title: 'Pug is great!'
    });
}

//Retrieve the submitted data and store them into the database
function storeAction (req, res) {
    //Get data
    const name = req.body.name;
    const age = req.body.age;

    //Store data
    db.insertData(name,age);

    //Show results
    res.render('store', {
        title: 'Pug is great!',
        name: name,
        age: age
    });
}
```

Abbildung 65: Inhalt von controller.js

2.6.6 Routes

In den „**routes**“ stehen die definierten Routingpfade für Express. Diese wurden bisher immer im Hauptserverfile definiert und ausgeführt. In größeren Projekten ist es aber üblich auch diese auszulagern.

Normalerweise könnte alles was sich im routes-Ordner befindet, im Controller stehen. Aber Ziel ist es den Code relativ übersichtlich zu schreiben und die einzelnen Dateien nicht zu überladen, vor allem nicht das Hauptserverfile.

Daher ergibt sich folgender Ansatz: Die nötigen Routingpfade werden in separaten Dateien lediglich initialisiert und die Ausführung der zugeteilten Funktionen, findet in den Controllern statt. Das bedeutet, dass Funktionen in den Controllern geschrieben werden und diese werden dann den Routingpfaden zugeordnet, wie folgende Abbildung zeigt.

```
const controller = require('./controller/controller');

//Defining routes and which function they execute from the controller
module.exports = function (app) {
    app.get('/', controller.indexAction);
    app.post('/store', controller.storeAction);
};
```

Abbildung 66: Inhalt von router.js

In der ersten Zeile wird der erstellte Controller eingebunden, wodurch seine beinhaltenden Funktionen mithilfe der zugeteilten Konstanten „**controller**“, aufrufbar sind.

2.6.7 Views

In diesem Bereich befinden sich 2 Dateien, die der Client zu sehen bekommt.

2.6.7.1 index.pug

Das ist die Startseite, die der Nutzer als erstes sieht. In dieser Startseite, befindet sich lediglich ein kleines Formular.

```
html
    head
        title!= title
    body
        form(method='POST' action='/store')
            div.form-group
                label(for='name') Name:
                input#name.form-control(type='text', placeholder='first name' name='name')
            div.form-group
                label(for='age') Age:
                input#age.form-control(type='text', placeholder='0-999' name='age')
            button.btn.btn-primary(type='submit') Save
```

Abbildung 67: Formular von index.pug

2.6.7.2 store.pug

Nachdem das Formular von der Startseite abgesendet wurde und die Daten in der Datenbank gespeichert wurden, dient diese Seite als Rückmeldung an den Client.

```
html
    head
        title!= title
    body
        span Name: 
            name!= name
            br
        span Age: 
            age!= age
            br

        p.
            Stored your submitted Data!

        //Load form again
        a(href='/') Go back
```

Abbildung 68: Rückmeldung an den Client

2.6.8 Models

In „db.js“ befinden sich 2 Funktionen die exportiert werden. Die erste Funktion stellt sicher, dass die benötigte Datenbank und zugehörige Tabelle erstellt sind, weshalb diese immer bei Serverstart aufgerufen wird. Die zweite Funktion dient dazu, die aus dem Formular übermittelten Daten zu speichern.

Als erstes müssen die benötigten Module geladen werden.

```
const config = require('../config/config');

//Load the async module
const async = require('async');

//Load the RethinkDB module
const r = require('rethinkdb');

//Load logger
const logger = require('../config/winston');
```

Abbildung 69: Benötigte Module laden

Danach wird angegeben, dass die 2 oben genannten Funktionen, exportiert werden sollen.

```
//Export the database query functions
module.exports = {
    initDB,
    insertData
};
```

Abbildung 70: Export der 2 Funktionen

Folgend werden die 2 Funktionen „**initDB**“ und „**insertData**“ ausprogrammiert.

```
function initDB(){
    async.waterfall([
        function (callback) {
            r.connect(config.rethinkdb, function (err, conn) {
                if (err){
                    logger.log('error','Failed to connect to the database \n' + err);
                }else{
                    logger.log('verbose','Database connection successful');
                }
                callback(null,conn);
            });
        },function (connection, callback) {
            r.dbCreate('finalPrototype').run(connection, function(err, result){
                if(err) {
                    logger.log('verbose','Database already created');
                } else {
                    logger.log('verbose','Created new database: finalPrototype');
                    logger.log('verbose',JSON.stringify(result, null, 2));
                }
                callback(null, connection);
            });
        },function (connection, callback) {
            r.db('finalPrototype').tableCreate('finalTable').run(connection, function(err, result) {
                if (err) {
                    logger.log('verbose','Table already created');
                }else{
                    logger.log('verbose','Created new table: finalTable');
                    logger.log('verbose',JSON.stringify(result, null, 2));
                }
                callback(null,'Database is ready');
            });
        }
    ],function (err, status) {
        if (err) logger.log('error',err);
        else logger.log('info',status);
    });
}
```

Abbildung 71:Programmierung von initDB

```
function insertData(name,age){
    logger.log('info','Inserting data into the database...');
    async.waterfall([
        function (callback) {
            r.connect(config.rethinkdb, function (err, conn) {
                if (err){
                    logger.log('error','Failed to connect to the database \n' + err);
                }else{
                    logger.log('verbose','Database connection successful');
                }
                callback(null,conn);
            });
        },function (connection, callback) {
            r.db('finalPrototype').table('finalTable').insert({
                name: name,
                age: age
            }).run(connection, function(err, result) {
                if (err) {
                    logger.log('error',err);
                }else{
                    logger.log('verbose','Data successfully inserted');
                    logger.log('verbose',JSON.stringify(result, null, 2));
                }
            });
            callback(null, 'Data successfully added')
        }

    ],function (err, status) {
        if (err) logger.log('error',err);
        else logger.log('info',status);
    });
}
```

Abbildung 72: Programmierung von insertData

2.6.9 Index.js

Im Hauptserverfile, werden nun alle Teilbereiche zusammengefügt.

```
//Load our winston logger
//=========================================================
const logger = require('./config/winston');

//Load config
//=========================================================
const config = require('./config/config.js');


//Load and initialize express
//=========================================================
const express = require('express');
const app = express();
logger.log('info','configuring express...');
```

Abbildung 73: Teil 1 von index.js

```
//Load and initialize the body parser module
//=======================================================
const bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({extended: false}));


//Load morgan and fs modules, write the logs to a file
//=======================================================
const morgan = require('morgan');
const fs = require('fs');

const accessLogStream = fs.createWriteStream(__dirname + '/logs/access.log', {flags: 'a'});

app.use(morgan('combined', {stream:accessLogStream}));

logger.log('verbose','http logger loaded');


//Set view engine to pug
//=======================================================
app.set('view engine', 'pug');
logger.log('verbose','view engine set to pug');


//Load the router
//=======================================================
require('./routes/routes')(app);
logger.log('verbose','routing paths set');


//Listener
//=======================================================
app.listen(config.server.port, function () {
    logger.log('info','express configured');
    logger.log('info','listening on port: ' + config.server.port);
});


//Load Database file and load the first setup
//=======================================================
const db = require('./models/db.js');

logger.log('info','loading database setup...');
db.initDB();
```

Abbildung 74: Teil 2 von index.js

2.6.10 Ausführung

Beim Serverstart erhält man nun Rückmeldungen, von den einzelnen geladenen Teilbereichen.

```
C:\Users\Rötzer\Documents\GitHub\RVP-Repository\Prototypes\Final>node index.js
2016-12-12T08:25:44.617Z - info: configuring express...
2016-12-12T08:25:44.664Z - verbose: http logger loaded
2016-12-12T08:25:44.664Z - verbose: view engine set to pug
2016-12-12T08:25:44.773Z - verbose: routing paths set
2016-12-12T08:25:44.789Z - info: loading database setup...
2016-12-12T08:25:44.789Z - info: express configured
2016-12-12T08:25:44.789Z - info: listening on port: 3000
2016-12-12T08:25:45.430Z - verbose: Database connection successful
2016-12-12T08:25:45.586Z - verbose: Database already created
2016-12-12T08:25:45.586Z - verbose: Table already created
2016-12-12T08:25:45.586Z - info: Database is ready
```

Abbildung 75: Rückmeldungen bei Serverstart

Nun kann der Client den Server unter „**localhost:3000**“ aufrufen und erhält als Resultat, die Startseite mit dem erstellten Formular. Nachdem Absenden des Formulars, werden die eingegebenen Daten gespeichert und der Client erhält eine Rückmeldung.

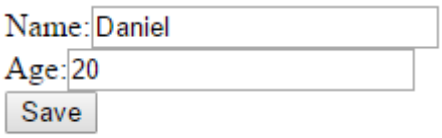

Abbildung 76: Aufruf der Startseite

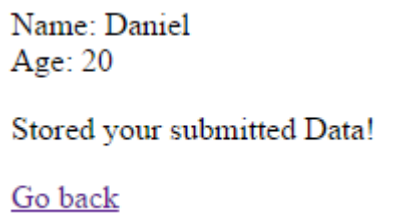

Abbildung 77: Rückmeldung an den Client