

**Prototypen \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |
| --- | --- |
| Autor | Daniel Rötzer |
| Erstellungsdatum | 22.11.16 |

Inhalt

[Abbildungsverzeichnis 4](#_Toc468445460)

[1 Vorbereitung 6](#_Toc468445461)

[1.1 Installation 6](#_Toc468445462)

[1.1.1 Node.js 6](#_Toc468445463)

[1.1.2 RethinkDB 6](#_Toc468445464)

[1.1.3 Paket/Modulinstallation 7](#_Toc468445465)

[1.1.3.1 package.json 7](#_Toc468445466)

[1.1.3.2 Node Package Manager 8](#_Toc468445467)

[1.1.3.3 Vorteil 9](#_Toc468445468)

[1.2 Server starten 9](#_Toc468445469)

[1.3 Datenbank starten 9](#_Toc468445470)

[1.4 Model View Controller – MVC 9](#_Toc468445471)

[1.4.1 Ordnerstruktur 10](#_Toc468445472)

[1.4.2 Bedeutungen 10](#_Toc468445473)

[1.5 Verfügbarkeit der Prototypen 10](#_Toc468445474)

[2 Prototypen 11](#_Toc468445475)

[2.1 HelloWorld 11](#_Toc468445476)

[2.1.1 Verwendete Pakete 11](#_Toc468445477)

[2.1.2 MVC 11](#_Toc468445478)

[2.1.3 Ordnerstruktur 11](#_Toc468445479)

[2.1.4 Express einbinden 11](#_Toc468445480)

[2.1.5 Browser Aufruf ermöglichen 11](#_Toc468445481)

[2.1.6 Server starten 12](#_Toc468445482)

[2.1.7 Hello World im Browser ausgeben 12](#_Toc468445483)

[2.1.7.1 Ein weiteres Beispiel eines anderen Routing Pfades 12](#_Toc468445484)

[2.2 Modules 12](#_Toc468445485)

[2.2.1 Verwendete Pakete 13](#_Toc468445486)

[2.2.2 MVC 13](#_Toc468445487)

[2.2.3 Ordnerstruktur 13](#_Toc468445488)

[2.2.4 Erstellung und Einbindung 13](#_Toc468445489)

[2.2.5 Config.js 13](#_Toc468445490)

[2.2.6 Anwendung der exportierten Funktionen und Zeichenketten 14](#_Toc468445491)

[2.3 Formulare 14](#_Toc468445492)

[2.3.1 BodyParser 15](#_Toc468445493)

[2.3.1.1 Verwendete Pakete 15](#_Toc468445494)

[2.3.1.2 MVC 15](#_Toc468445495)

[2.3.1.3 Ordnerstruktur 15](#_Toc468445496)

[2.3.1.4 Einbindung und Anwendung 15](#_Toc468445497)

[2.3.1.5 Formulardaten empfangen und zurücksenden 16](#_Toc468445498)

[2.3.1.6 Ausführung 16](#_Toc468445499)

[2.3.2 Pug 17](#_Toc468445500)

[2.3.2.1 Verwendete Pakete 17](#_Toc468445501)

[2.3.2.2 MVC 17](#_Toc468445502)

[2.3.2.3 Ordnerstruktur 17](#_Toc468445503)

[2.3.2.4 Pug Syntax 17](#_Toc468445504)

[2.3.2.5 Einbindung 18](#_Toc468445505)

[2.3.2.6 index.pug 18](#_Toc468445506)

[2.3.2.7 hello.pug 19](#_Toc468445507)

[2.3.2.8 Im Browser aufrufen 19](#_Toc468445508)

[2.3.2.9 Ausführung 20](#_Toc468445509)

[2.4 Logging 20](#_Toc468445510)

[2.4.1 Warum Logging anwenden? 20](#_Toc468445511)

[2.4.2 Winston 21](#_Toc468445512)

[2.4.2.1 Verwendete Pakete 21](#_Toc468445513)

[2.4.2.2 MVC 21](#_Toc468445514)

[2.4.2.3 Ordnerstruktur 21](#_Toc468445515)

[2.4.2.4 Logging-Levels 21](#_Toc468445516)

[2.4.2.5 Erstellung eines neuen Loggers 22](#_Toc468445517)

[2.4.2.5.1 Definierbare Eigenschaften der Transporte: 22](#_Toc468445518)

[2.4.2.5.2 Weitere Eigenschaften des Loggers 23](#_Toc468445519)

[2.4.2.5.3 Export unseres Loggers 23](#_Toc468445520)

[2.4.2.6 Einbindung und Anwendung in unserem Server 23](#_Toc468445521)

[2.4.2.7 Ausführung 24](#_Toc468445522)

[2.4.3 Morgan 25](#_Toc468445523)

[2.4.3.1 Verwendete Pakete 25](#_Toc468445524)

[2.4.3.2 MVC 26](#_Toc468445525)

[2.4.3.3 Ordnerstruktur 26](#_Toc468445526)

[2.4.3.4 Einbindung und Anwendung 26](#_Toc468445527)

[2.4.3.5 Test der http-Protokollierung 26](#_Toc468445528)

[2.5 RethinkDB 27](#_Toc468445529)

[2.5.1 Verwendete Pakete 27](#_Toc468445530)

[2.5.2 MVC 27](#_Toc468445531)

[2.5.3 Ordnerstruktur 27](#_Toc468445532)

[2.5.4 Erstellung und Einbindung 28](#_Toc468445533)

[2.5.5 Datenbankabfragen 28](#_Toc468445534)

[2.5.5.1 initDB 28](#_Toc468445535)

[2.5.5.2 insertTestData 29](#_Toc468445536)

[2.5.6 Server 30](#_Toc468445537)

[2.5.7 Ausführung 31](#_Toc468445538)

[2.6 Final 31](#_Toc468445539)

[2.6.1 Verwendete Pakete 31](#_Toc468445540)

[2.6.2 MVC 31](#_Toc468445541)

[2.6.3 Ordnerstruktur 31](#_Toc468445542)

# Abbildungsverzeichnis

[Abbildung 1: Node.js Installer Download 6](#_Toc468445543)

[Abbildung 2: RethinkDB Download 7](#_Toc468445544)

[Abbildung 3: package.json Dateierstellung beginnen mit: npm init 7](#_Toc468445545)

[Abbildung 4: Angaben der Projektinformationen und Endergebnis 8](#_Toc468445546)

[Abbildung 5: Paketinstallation 8](#_Toc468445547)

[Abbildung 6: HelloWorld Ordnerstruktur 11](#_Toc468445548)

[Abbildung 7: Express in den Server einbinden 11](#_Toc468445549)

[Abbildung 8: Server „horcht“ auf localhost:3000 11](#_Toc468445550)

[Abbildung 9: Server starten mit - node index.js 12](#_Toc468445551)

[Abbildung 10: Antwort an die GET-Anfrage programmieren 12](#_Toc468445552)

[Abbildung 11: Anderen Routing-Pfad definieren 12](#_Toc468445553)

[Abbildung 12: Modules Ordnerstruktur 13](#_Toc468445554)

[Abbildung 13: Config-Datei einbinden 13](#_Toc468445555)

[Abbildung 14: Einfache Funktionen und Informationen exportieren 13](#_Toc468445556)

[Abbildung 15: Exportierte Funktionen mithilfe der Shorthand-Methode 14](#_Toc468445557)

[Abbildung 16: Anwendungsbeispiele unseres Moduls 14](#_Toc468445558)

[Abbildung 17: Ausgaben in der Konsole nach Serverstart 14](#_Toc468445559)

[Abbildung 18: BodyParser Ordnerstruktur 15](#_Toc468445560)

[Abbildung 19: body-parser einbinden und anwenden 15](#_Toc468445561)

[Abbildung 20: Ordner für die statischen Dateien festlegen 15](#_Toc468445562)

[Abbildung 21: Inhalt von index.html 16](#_Toc468445563)

[Abbildung 22: POST-Methode definieren 16](#_Toc468445564)

[Abbildung 23: Formulareingabe 16](#_Toc468445565)

[Abbildung 24: Ausgabe des übergebenen Wertes 16](#_Toc468445566)

[Abbildung 25: Pug Ordnerstruktur 17](#_Toc468445567)

[Abbildung 26: Einfache Pug Seite 18](#_Toc468445568)

[Abbildung 27: Einfache HTML Seite 18](#_Toc468445569)

[Abbildung 28: view engine auf pug setzen 18](#_Toc468445570)

[Abbildung 29: Inhalt von index.pug 19](#_Toc468445571)

[Abbildung 30: Inhalt von hello.pug 19](#_Toc468445572)

[Abbildung 31: Antwort an die GET-Anfrage programmieren 19](#_Toc468445573)

[Abbildung 32: Formulardaten mittels POST-Methode annehmen und danach ausgeben 20](#_Toc468445574)

[Abbildung 33: Formulareingabe 20](#_Toc468445575)

[Abbildung 34: Ausgabe des Übergebenen Wertes 20](#_Toc468445576)

[Abbildung 35: Winston Ordnerstruktur 21](#_Toc468445577)

[Abbildung 36: Winston einbinden 22](#_Toc468445578)

[Abbildung 37: Logger erstellen 22](#_Toc468445579)

[Abbildung 38: Weitere Einstellungen definieren 23](#_Toc468445580)

[Abbildung 39: Logger exportieren 23](#_Toc468445581)

[Abbildung 40: Logger einbinden 23](#_Toc468445582)

[Abbildung 41: Logging Anwendungsbeispiele 23](#_Toc468445583)

[Abbildung 42: Andere Schreibweise für Logs 24](#_Toc468445584)

[Abbildung 43: Logging Ausgabe in der Konsole 24](#_Toc468445585)

[Abbildung 44: Ändern des Log-Levels der Konsole 24](#_Toc468445586)

[Abbildung 45: Neue Logging Ausgabe in der Konsole 24](#_Toc468445587)

[Abbildung 46: Nicht abgefangener Fehler 25](#_Toc468445588)

[Abbildung 47: Protokollierte Fehlermeldung 25](#_Toc468445589)

[Abbildung 48: Morgan Ordnerstruktur 26](#_Toc468445590)

[Abbildung 49: writeStream erstellen und Morgan anwenden 26](#_Toc468445591)

[Abbildung 50: 3 definierte Routing-Pfade 27](#_Toc468445592)

[Abbildung 51: Ausschnitt der Log-Datei 27](#_Toc468445593)

[Abbildung 52: RethinkDB Ordnerstruktur 27](#_Toc468445594)

[Abbildung 53: Datenbankinformationen 28](#_Toc468445595)

[Abbildung 54: Benötigte Module laden 28](#_Toc468445596)

[Abbildung 55: Export von 2 Datenbankabfragen 28](#_Toc468445597)

[Abbildung 56: Sicherstellen, dass Datenbank und Table erstellt sind 29](#_Toc468445598)

[Abbildung 57: Einfache Testdaten einfügen 30](#_Toc468445599)

[Abbildung 58: Exportierte Datenbankabfragen mit Zeitverzögerung ausführen 30](#_Toc468445600)

[Abbildung 59: Ergebnisse der ausgeführten Datenbankabfragen 31](#_Toc468445601)

1. Vorbereitung

# 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/Module

## 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.

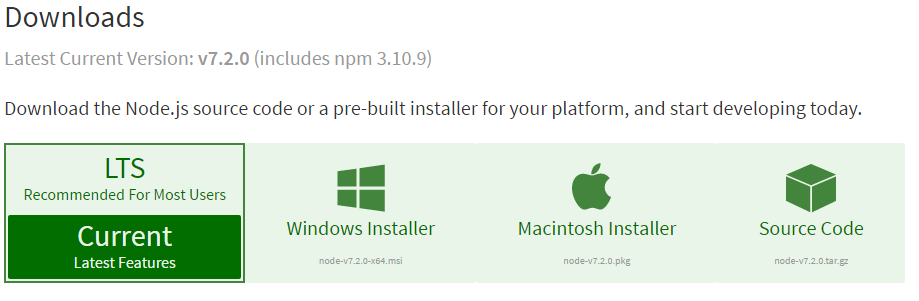


Abbildung 1: Node.js Installer Download

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

## 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.

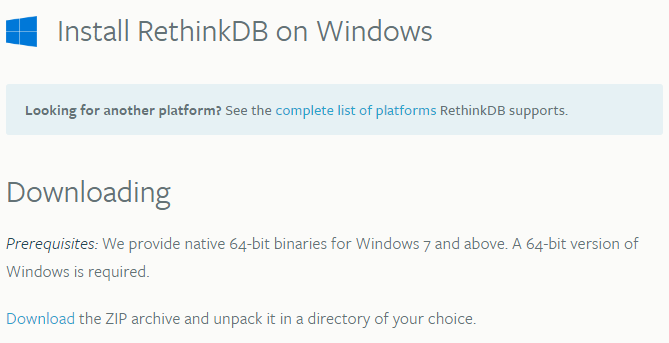


Abbildung 2: RethinkDB Download

## Paket/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.

### 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.



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:

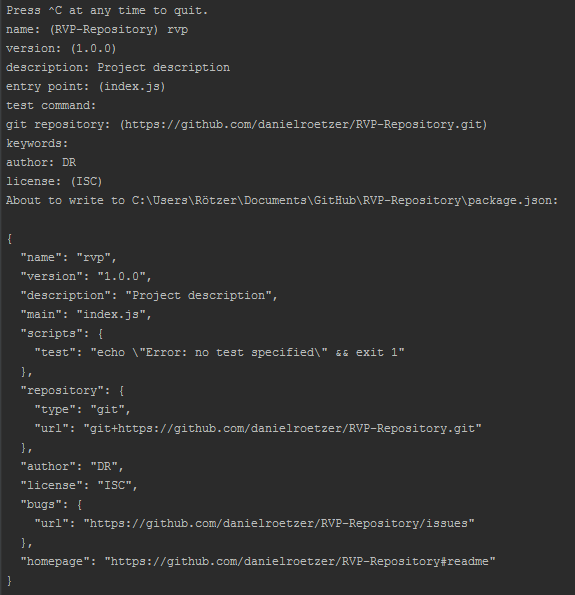


Abbildung 4: Angaben der Projektinformationen und Endergebnis

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

Nun sind die Vorbereitungen abgeschlossen.

### 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.



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.

### 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**

# 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.

# Datenbank starten

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

# 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.

## Ordnerstruktur

Folgende übersichtliche Ordnerstruktur unterstützt das MVC Prinzip.

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

## 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.

Folgende Grafik Zeigt die Kommunikation zwischen den einzelnen Komponenten:

GRAFIK

# 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.

1. Prototypen

# HelloWorld

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

## Verwendete Pakete

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

* Express

## 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.

## Ordnerstruktur

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

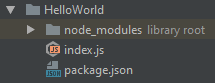


Abbildung 6: HelloWorld Ordnerstruktur

## Express einbinden

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

init_express

Abbildung 7: Express in den Server einbinden

## Browser Aufruf ermöglichen

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

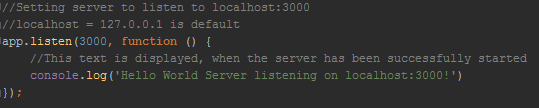


Abbildung 8: Server „horcht“ auf localhost:3000

## 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.



Abbildung 9: Server starten mit - node index.js

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

## Hello World im Browser ausgeben

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

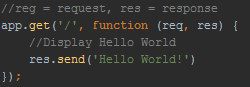


Abbildung 10: Antwort an die GET-Anfrage programmieren

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.

### 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

Abbildung 11: Anderen Routing-Pfad definieren

# 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.

## Verwendete Pakete

Keine Pakete verwendet.

## MVC

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

## Ordnerstruktur

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

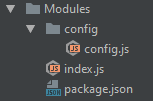


Abbildung 12: Modules Ordnerstruktur

## 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:

config_import

Abbildung 13: Config-Datei einbinden

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

## Config.js

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

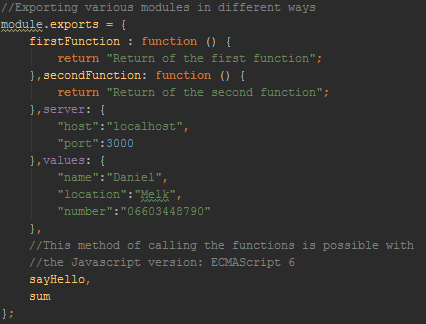


Abbildung 14: Einfache Funktionen und Informationen exportieren

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.

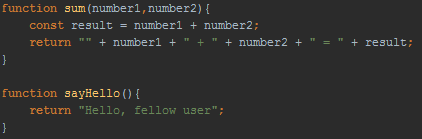


Abbildung 15: Exportierte Funktionen mithilfe der Shorthand-Methode

## Anwendung der exportierten Funktionen und Zeichenketten

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

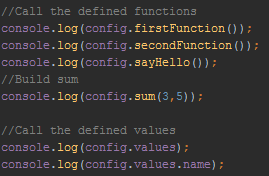


Abbildung 16: Anwendungsbeispiele unseres Moduls

Wird der Server gestartet, erhält man folgendes Ergebnis.

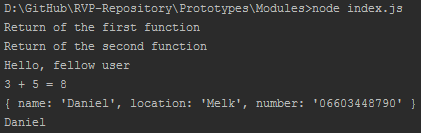


Abbildung 17: Ausgaben in der Konsole nach Serverstart

# Formulare

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

## 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.

### Verwendete Pakete

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

* express
* body-parser

### MVC

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

### Ordnerstruktur

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

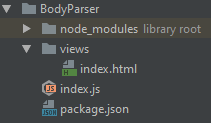


Abbildung 18: BodyParser Ordnerstruktur

### 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.

bodyparser_import

Abbildung 19: body-parser einbinden und anwenden

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

bodyparser_view

Abbildung 20: 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.

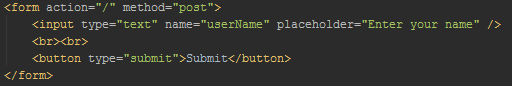


Abbildung 21: Inhalt von index.html

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

### Formulardaten empfangen und zurücksenden

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

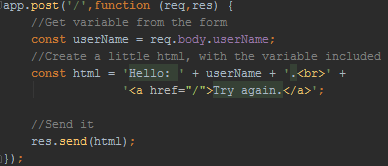


Abbildung 22: POST-Methode definieren

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).

### Ausführung

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

|  |  |
| --- | --- |
| bodyparser_result1  Abbildung 23: Formulareingabe | bodyparser_result2  Abbildung 24: Ausgabe des übergebenen Wertes |

## 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.

### Verwendete Pakete

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

* express
* body-parser
* pug

### MVC

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

### Ordnerstruktur

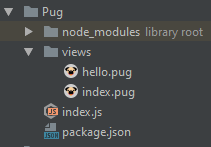


Abbildung 25: Pug Ordnerstruktur

### 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.

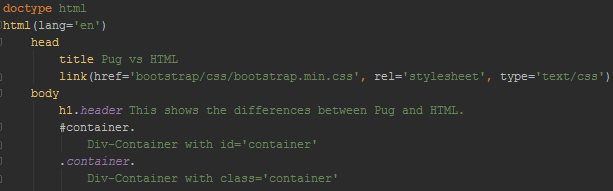


Abbildung 26: Einfache Pug Seite

Wird zu:

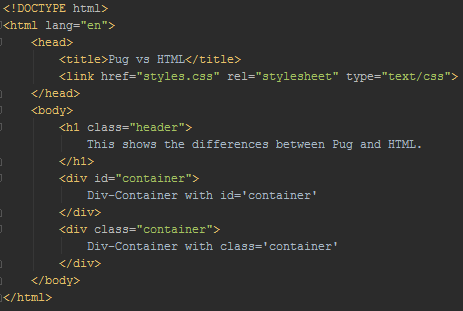


Abbildung 27: Einfache HTML Seite

### Einbindung

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

pug_import

Abbildung 28: 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.

### 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.

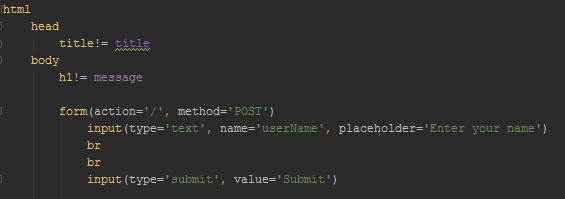


Abbildung 29: Inhalt von index.pug

### 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.

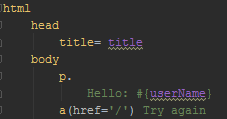


Abbildung 30: Inhalt von hello.pug

### 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.

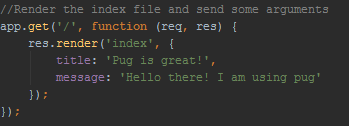


Abbildung 31: Antwort an die GET-Anfrage programmieren

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

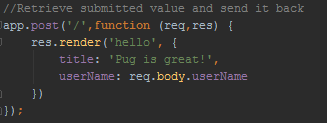


Abbildung 32: Formulardaten mittels POST-Methode annehmen und danach ausgeben

### 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.

|  |  |
| --- | --- |
| C:\Users\Rötzer\AppData\Local\Microsoft\Windows\INetCache\Content.Word\pug-ergebnis1.png  Abbildung 33: Formulareingabe | C:\Users\Rötzer\AppData\Local\Microsoft\Windows\INetCache\Content.Word\pug-ergebnis2.png  Abbildung 34: Ausgabe des Übergebenen Wertes |

# 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.

## 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.

## 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.

### Verwendete Pakete

* winston

### 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.

### Ordnerstruktur

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

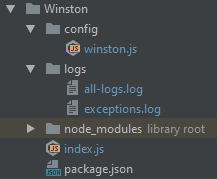


Abbildung 35: Winston Ordnerstruktur

### 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).

### 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.



Abbildung 36: Winston einbinden

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

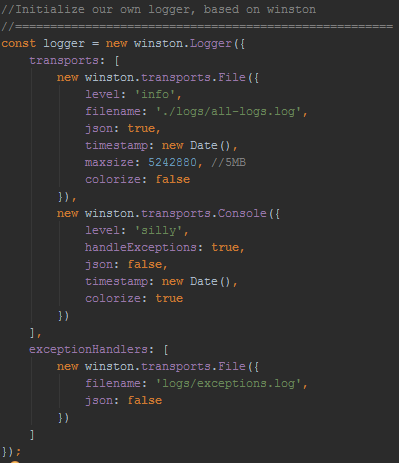


Abbildung 37: Logger erstellen

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.

#### 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

#### 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.

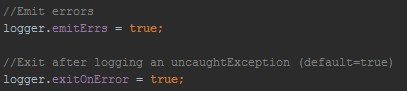


Abbildung 38: Weitere Einstellungen definieren

#### Export unseres Loggers

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



Abbildung 39: Logger exportieren

### Einbindung und Anwendung in unserem Server

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



Abbildung 40: Logger einbinden

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

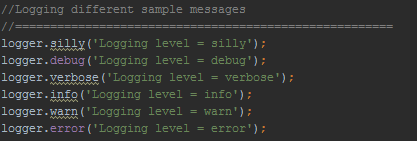


Abbildung 41: Logging Anwendungsbeispiele

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

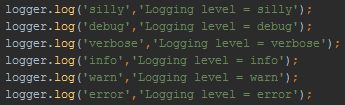


Abbildung 42: Andere Schreibweise für Logs

### 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.

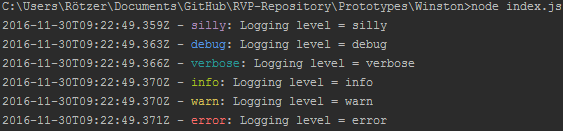


Abbildung 43: Logging Ausgabe in der Konsole

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

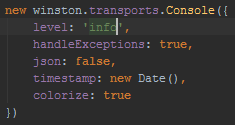


Abbildung 44: Ändern des Log-Levels der Konsole

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

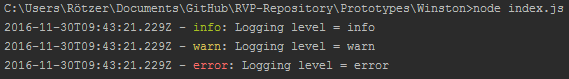


Abbildung 45: Neue Logging Ausgabe in der Konsole

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 46: 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 47: Protokollierte Fehlermeldung

Mit solchen Einträgen, kann ein Programmierer gut Fehlerbehebungen vornehmen.

## 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)

### 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.

### 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.

### Ordnerstruktur

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

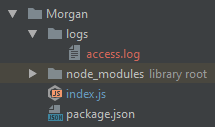
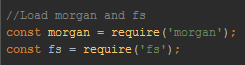


Abbildung 48: Morgan Ordnerstruktur

### 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.



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

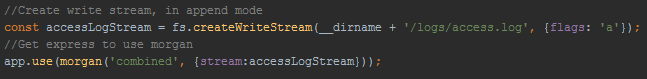


Abbildung 49: writeStream erstellen und Morgan anwenden

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.

### 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.

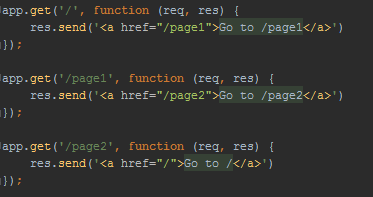


Abbildung 50: 3 definierte Routing-Pfade

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.



Abbildung 51: Ausschnitt der Log-Datei

# RethinkDB

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

## Verwendete Pakete

* async
* rethinkdb

## MVC

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

## Ordnerstruktur

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

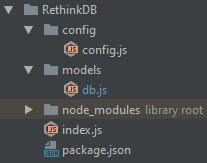


Abbildung 52: RethinkDB Ordnerstruktur

## Erstellung und Einbindung

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

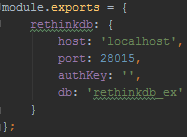


Abbildung 53: Datenbankinformationen

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.

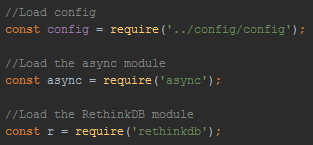


Abbildung 54: Benötigte Module laden

## 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.

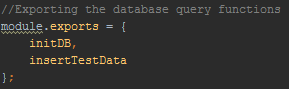


Abbildung 55: Export von 2 Datenbankabfragen

### 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.



Abbildung 56: 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

### 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.

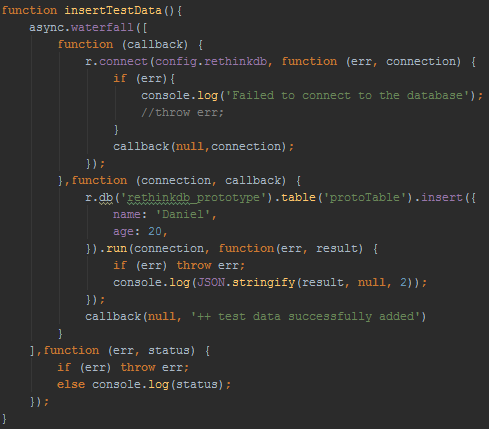


Abbildung 57: Einfache Testdaten einfügen

**Aufgaben der 2 anonymen Funktionen:**

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

## 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.

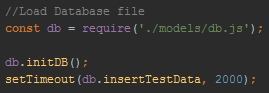


Abbildung 58: Exportierte Datenbankabfragen mit Zeitverzögerung ausführen

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.

## 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.

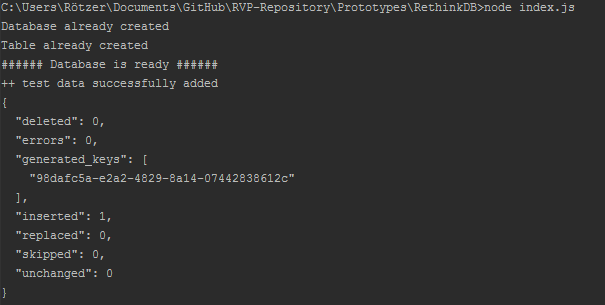


Abbildung 59: Ergebnisse der ausgeführten Datenbankabfragen

# 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.

## Verwendete Pakete

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

## 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.

## Ordnerstruktur