Inhalt

[Startup 3](#_Toc467050020)

[Beispiel 3](#_Toc467050021)

[Paketinstallation 4](#_Toc467050022)

[Vorteil 4](#_Toc467050023)

[Beispiel 4](#_Toc467050024)

[Hello World 5](#_Toc467050025)

[Verwendete Pakete 5](#_Toc467050026)

[Express laden 5](#_Toc467050027)

[Browser URL bestimmen 5](#_Toc467050028)

[Server starten 5](#_Toc467050029)

[Text im Browser ausgeben 5](#_Toc467050030)

[Ein weiteres Beispiel eines anderen Routing Pfades 6](#_Toc467050031)

[Modules 6](#_Toc467050032)

[Verwendete Pakete 6](#_Toc467050033)

[Erstellung und Einbindung 6](#_Toc467050034)

[Config.js 6](#_Toc467050035)

[Anwendung der exportierten Funktionen und Zeichenketten 7](#_Toc467050036)

[Morgan 7](#_Toc467050037)

[Verwendete Pakete 7](#_Toc467050038)

[Einbindung 8](#_Toc467050039)

[Datenspeicherung in externer Datei 8](#_Toc467050040)

[Browseraufruf 8](#_Toc467050041)

[BodyParser 8](#_Toc467050042)

[Verwendete Pakete 8](#_Toc467050043)

[Einbindung 8](#_Toc467050044)

[Formulardaten empfangen und zurücksenden 9](#_Toc467050045)

[Ergebnis 9](#_Toc467050046)

[Pug 9](#_Toc467050047)

[Verwendete Pakete 9](#_Toc467050048)

[Einbindung 10](#_Toc467050049)

[Index.pug 10](#_Toc467050050)

[Im Browser aufrufen 10](#_Toc467050051)

[Kombination mit body-parser 10](#_Toc467050052)

[RethinkDB 11](#_Toc467050053)

[Verwendete Pakete 11](#_Toc467050054)

[Erstellung und Einbindung 11](#_Toc467050055)

[Datenbankabfragen 11](#_Toc467050056)

[initDB 12](#_Toc467050057)

[insertTestData 13](#_Toc467050058)

[Server 13](#_Toc467050059)

[Ergebnis 14](#_Toc467050060)

[Final 14](#_Toc467050061)

[Verwendete Pakete 14](#_Toc467050062)

[Ordnerstruktur 14](#_Toc467050063)

[Model – View – Controller Modell 14](#_Toc467050064)

[Config 15](#_Toc467050065)

[Einbindung 15](#_Toc467050066)

[Controller 16](#_Toc467050067)

[Views 17](#_Toc467050068)

[Models 17](#_Toc467050069)

[Logs 18](#_Toc467050070)

[Ergebnis 19](#_Toc467050071)

[Verfügbarkeit der Prototypen 19](#_Toc467050072)

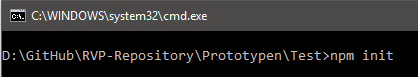
Prototypen

# Startup

Zu Beginn erstellt man immer als aller erstes ein - **package.json -** File. In diesem Dokument befinden sich allerlei Projektinformationen, sowie notwendige Pakete, die man installieren muss, damit das zugehörige Programm funktionsfähig ist. Die Paketinstallation wird später näher erleutert.

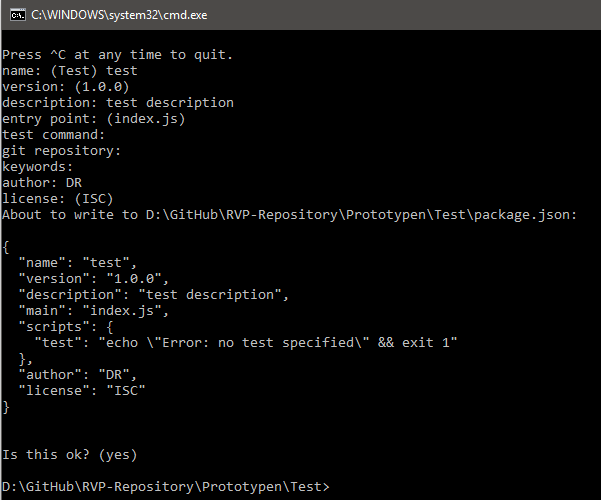
## Beispiel

Als Beispiel wird ein neuer Ordner unter dem Namen - **Test** - erstellt. Um jetzt einfach das nötige JSON Dokument zu erstellen, öffnet man in CMD diesen Ordner und gibt den Befehl - **npm init** - ein.



Währenddessen werden verschiedene Projektinformationen abgefragt. Wird nichts Spezifisches angegeben, so wird der Wert in der Klammer übernommen.

Folgend wird der Verlauf von - **npm init** - dargestellt. Am Ende der Befragung wird das entstandene Dokument dargestellt und gefragt, ob dies richtig ist. Der - **entry point** - bestimmt das Hauptprogramm.



# Paketinstallation

Um für das Programm notwendige Pakete zu installieren, öffnet man in CMD den Ordner, indem sich die – **package.json** – Datei befindet.

Als nächstes führt man folgenden Befehl aus: **npm install [paketname] [--save]**

Die Parameter in eckiger Klammer sind optional.

* Werden alle 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**

## Beispiel

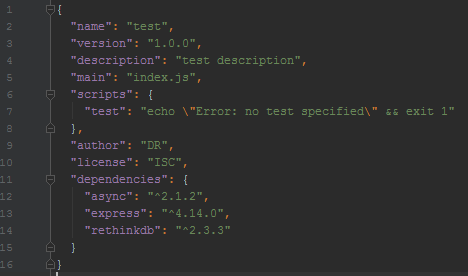
Als Beispiel wird dieser Befehl im Test-Ordner ausgeführt: **npm install express rethinkdb async –save**

Darauf werden die 3 Pakete

* express
* rethinkdb
* async

installiert und im JSON Dokument vermerkt.

In folgendem Screenshot wird nun das komplette JSON Dokument gezeigt. Die installierten Pakete wurden unter dem Namen – **dependencies** – vermerkt.



# Hello World

In diesem Programm sollen, durch Aufruf des Servers im Browser, einfache Texte ausgegeben werden.

## Verwendete Pakete

Pakete in Node, werden auch Module genannt

* express

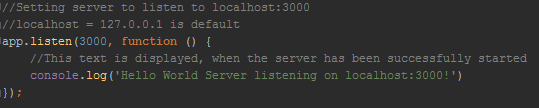
## Express laden

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



## Browser URL bestimmen

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



## Server starten

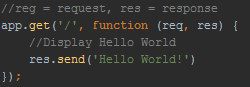
Um den Server zu starten müssen wir in CMD den Befehl – **node servername.js** – ausführen. Klappt alles, wird unser vorgegebener Text ausgegeben, wie man folgend sehen kann.



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

## Text im Browser ausgeben

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



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.

**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.



# 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.

## Erstellung und Einbindung

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

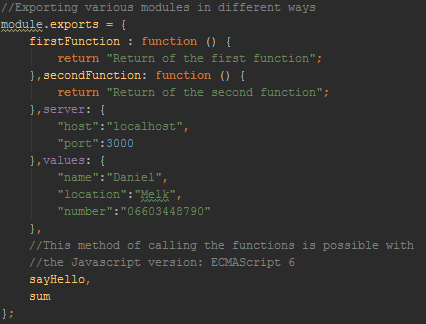
Um das File einzubinden, reicht diese Zeile, welche in der Server Datei gespeichert wird:



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

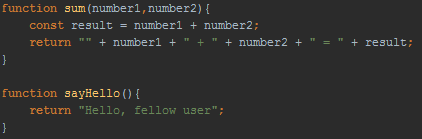
## Config.js

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



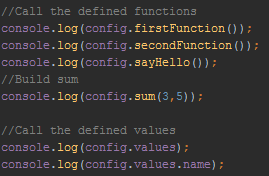
Als erstes wird immer der Name der Funktion oder der Zeichenkette angeben. Mit diesen Namen, können alle Informationen später aufgerufen werden.

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

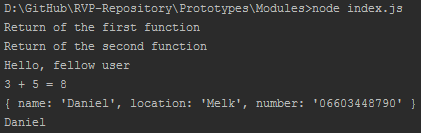


## Anwendung der exportierten Funktionen und Zeichenketten

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



Wird der Server gestartet, erhält man folgendes Ergebnis.



# Morgan

Morgan ist ein Paket, das zur Protokollierung der Nutzerzugriffe dient. Dabei werden nicht nur Zugriffe protokolliert, sondern auch Information wie z.B.: mit welchem Browser.

## Verwendete Pakete

* express
* morgan

## Einbindung

Es werden hier 2 Module (Express ausgeschlossen) eingebunden. Das zweite Modul – **fs** – steht für File System und dient zur Datenspeicherung, in einer externen Datei. Das Modul muss nicht zuerst heruntergeladen werden, weil es bereits unter den Standardpaketen vorhanden ist.



**Achtung:** Natürlich muss auch Express eingebunden werden

## Datenspeicherung in externer Datei

Als erstes wird in einer Konstanten definiert, was getan wird und in welche Datei geschrieben wird. Danach wird Morgan angewendet mithilfe von Express, das wieder unter der Konstanten – **app** – eingebunden wurde.



## Browseraufruf

Um das ganze nun zu testen, müssen natürlich die notwendigen Express Funktionen auch erstellt sein (im Prototyp Hello World sichtbar).

Jedes Mal, wenn der Server aufgerufen wird, wird in der externen Datei protokolliert.

# BodyParser

Mithilfe des – **body-parser** – Paketes, können HTML Elemente in Javascript Elemente umgewandelt werden. Dadurch können gesendete Formulardaten angenommen und bearbeitet werden.

## Verwendete Pakete

* express
* body-parser

## Einbindung

Das Paket wird eingebunden, wie jedes andere. Wie gewohnt, ist Express in – **app** – initialisiert.

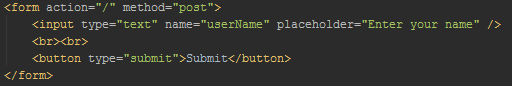


Als nächstes wird der Ordner festgelegt, indem sich die statischen HTML, CSS, usw. befinden.



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

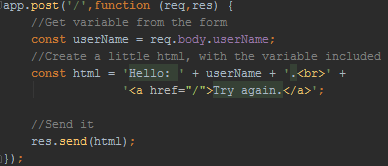
Folgend wird das Formular der HTML Datei abgebildet.



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.



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.

## Ergebnis

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

|  |  |
| --- | --- |
|  |  |

# Pug

Mithilfe von Pug können Formulardaten optimal wieder an den Client gesendet werden. Im Prototyp BodyParser, wurde HTML im Server erzeugt, was natürlich nicht optimal ist. Jedoch hat Pug nicht nur diese Funktion, es verändert die gesamte Schreibweise von HTML Seiten.

## Verwendete Pakete

* express
* pug

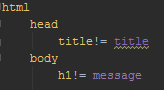
## Einbindung

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



## Index.pug

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. Folgende Zeilen ist der, Inhalt der erstellt Datei. Die beiden Variablen werden ersetzt, durch mitgesendete Werte.

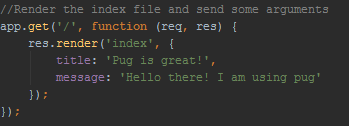


Pug ist vom Aufbau ziemlich dasselbe wie HTML, jedoch mit starken Syntax unterschieden. Diese Seite erklärt die Unterschiede gut: <https://www.sitepoint.com/jade-tutorial-for-beginners/>

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

## 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 noch Werte übergeben, welche Formulardaten sein könnten

Im Browser aufgerufen, ist der Titel der Seite – **Pug is great!** – und als Überschrift steht – **Hello there! I am using pug**.

## Kombination mit body-parser

Kombiniert man Pug mit body-parser, ist es nicht mehr notwendig in Javascript Dateien, HTML zu erstellen. Dies wird im finalen Prototyp angewendet.

# RethinkDB

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

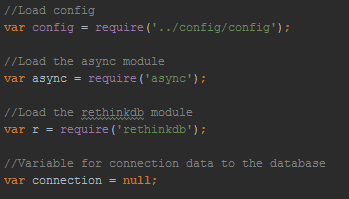
## Verwendete Pakete

* async
* rethinkdb

## Erstellung und Einbindung

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

In dieser Javascript Datei werden die folgenden Module eingebunden und eine Variable für die Datenbankverbindung erstellt.

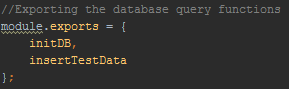


Im Config File befinden sich notwendige Informationen zur Datenbankanbindung.



## Datenbankabfragen

Nach dem einbinden der Module, werden in der Datei – **db.js** – 2 Funktionen definiert und exportiert.

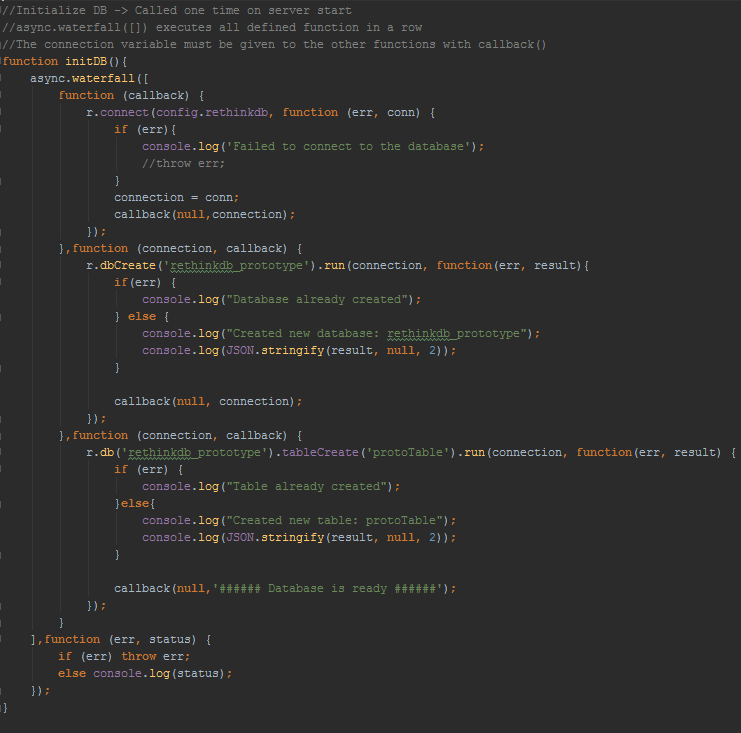


Wie die Namen verraten, Initialisiert die erste Funktion die Datenbank und die zweite fügt einfache Testdaten hinzu.

### 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.

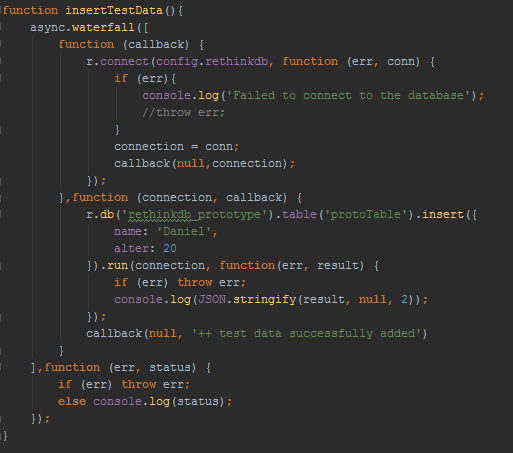


**Aufgaben der 3 anonymen Funktionen**:

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.

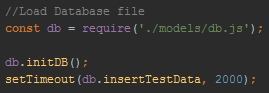


**Aufgaben der 2 anonymen Funktionen:**

1. Verbindung zur Datenbank herstellen und diese weitergeben
2. Testdaten einfügen

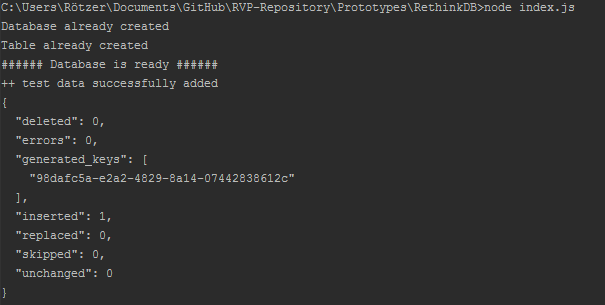
## Server

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.



## Ergebnis

Die Ausgaben in der Konsole, wenn alles erstellt werden muss, sind viel zu lang. Also hier die Ausgaben, wenn alles bereits erstellt ist.



# Final

In diesem 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

## Ordnerstruktur

Da dieses Projekt bereits etwas größer ist, lohnt es sich das MVC Modell anzuwenden.

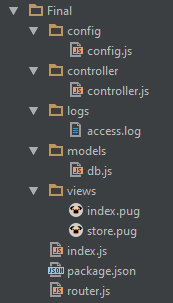
### Model – View – Controller Modell

**Models:** In diesem Bereich befinden sich sämtliche Anwendungen, bei denen Daten gespeichert werden.

**Views:** Hier wird alles erledigt, was der Nutzer sehen kann. Also die Daten aus den Models werden mithilfe von HTML, CSS, … dargestellt.

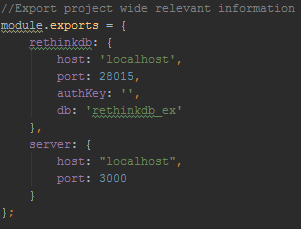
**Controller:** Der Controller dient dazu, Models und Views miteinander zu Verbinden. Er übernimmt also die Rolle der Schnittstelle.

Vollständige Ordnerstruktur mit allen benötigten Dateien:



## Config

Unsere Konfiguration unterscheidet sich nicht von anderen Beispielen.



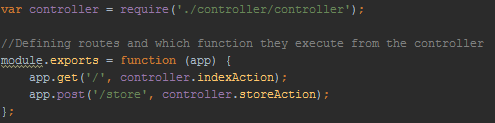
## Einbindung

Alle Einbindungen sämtlicher Module sind das Gleiche wie bei den vorigen Prototypen. Folgende Module werden im Server File eingebunden:

* express
* config
* body-parser
* morgan + fs
* Die exportierten Datenbankabfragen aus unseren Models
* pug

Zusätzlich, um das Server File kurz zu halten, wurden die Routing Methoden von Express ausgelagert. Diese werden in die Datei – **router.js** – gespeichert.

Als erstes wird der Controller eingebunden und danach die Routen und die zugehörige Funktion, die aus dem Controller entnommen werden.

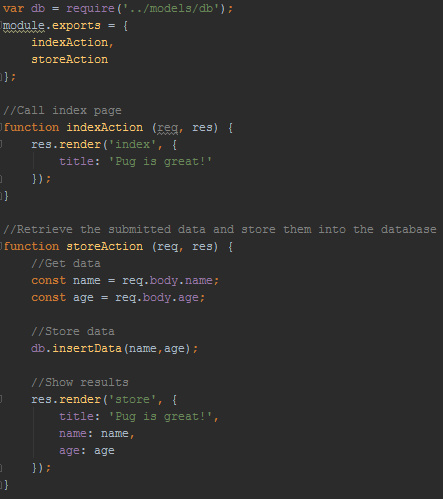


Nun wird der Router in unseren Server eingebunden.



## Controller

Im Controller befinden sich die Funktionen, die im Router angegeben werden. Hier findet die Verbindung zwischen den Models und den Views statt.

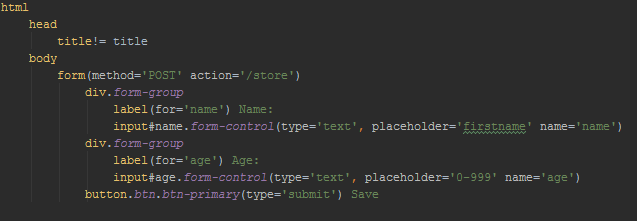


**indexAction:** Lädt die Startseite, wenn der Server unter localhost:3000 aufgerufen wird.

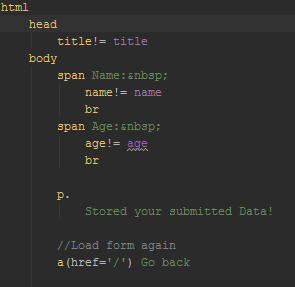
**storeAction:** Nimmt Formulardaten an, speichert diese in die Datenbank und lädt anschließen eine andere HTML Seite.

## Views

In der Index Datei, befindet sich lediglich ein Formular mit 2 Eingabefeldern.

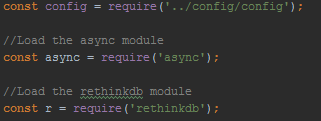


Die Store Datei dient lediglich zur Bestätigung, dass die Daten gespeichert wurden.

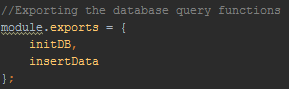


## Models

Hier befindet sich der gesamte Code zur Datenspeicherung in der Datenbank. Als erstes werden die benötigten Module importiert.

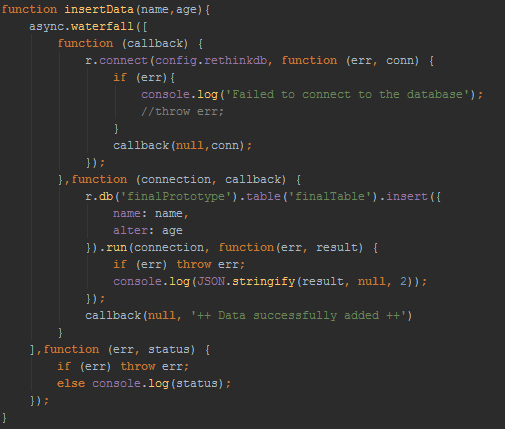


Anschließend werden alle Funktionen als Modul exportiert.



**initDB:** Stellt sicher, dass die benötigte Datenbank und zugehörigen Tables erstellt sind. Kann aus dem Prototyp RethinkDB entnommen werden. Um dies zu garantieren, muss die Funktion im Server File ausgeführt werden.

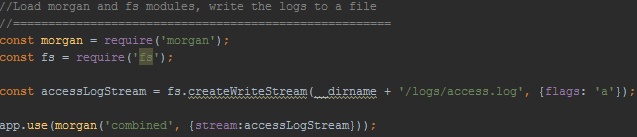
**insertData:** Speichert die übergebenen Daten aus dem Formular.



Die Parameter **name** und **age**, werden bei Aufruf der Funktion im Controller, übergeben.

## Logs

Die Datei, in welcher das Modul Morgan protokolliert, muss nicht vorher erstellt werden. Es wird alles aus dem Prototyp Morgan verwendet, lediglich der Pfad zur Datei muss angepasst werden.



## Ergebnis

Als erstes führt uns der Server zum Formular, was im ersten Screenshot abgebildet wird. Sendet man das Formular mit dem Button ab, erhält man das Ergebnis aus dem zweiten Screenshot nebenbei.

|  |  |
| --- | --- |
| C:\Users\Rötzer\AppData\Local\Microsoft\Windows\INetCache\Content.Word\final_result1.png |  |

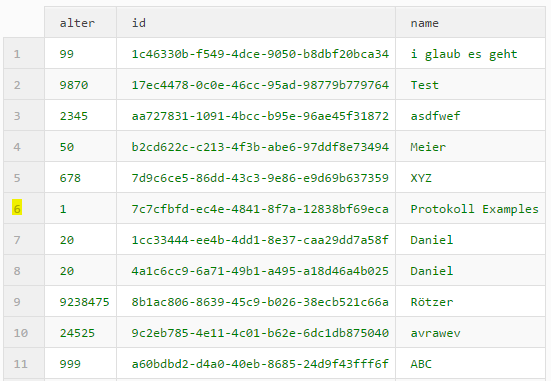
Um zu überprüfen, ob die Daten wirklich in der Datenbank gespeichert wurden, ruft man diesen Link auf: <http://localhost:8080/#dataexplorer>

Unter dem Port 8080, befindet sich ein Client zur Datenüberwachung von RethinkDB.

Nun erteilt man im Data Explorer den Befehl, alle Daten auszugeben:

* r.db('finalPrototype').table('finalTable')

Anschließend den Button **Run** betätigen und man erhält alle Daten.



# Verfügbarkeit der Prototypen

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