

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

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

Inhalt

[1. Vorwort 4](#_Toc467849987)

[1.1 Installation 4](#_Toc467849988)

[1.1.1 Node.js 4](#_Toc467849989)

[1.1.2 RethinkDB 4](#_Toc467849990)

[1.1.3 Paket/Modulinstallation 5](#_Toc467849991)

[1.1.3.1 package.json 5](#_Toc467849992)

[1.1.3.2 Node Package Manager 6](#_Toc467849993)

[1.1.3.3 Vorteil 7](#_Toc467849994)

[1.2 Server starten 7](#_Toc467849995)

[1.3 MVC 7](#_Toc467849996)

[1.4 Verfügbarkeit der Prototypen 7](#_Toc467849997)

[2. Prototypen 8](#_Toc467849998)

[2.1 Hello World 8](#_Toc467849999)

[2.1.1 Verwendete Pakete 8](#_Toc467850000)

[2.1.2 Express einbinden 8](#_Toc467850001)

[2.1.3 Browser Aufruf ermöglichen 8](#_Toc467850002)

[2.1.4 Server starten 8](#_Toc467850003)

[2.1.5 Hello World im Browser ausgeben 9](#_Toc467850004)

[2.1.5.1 Ein weiteres Beispiel eines anderen Routing Pfades 9](#_Toc467850005)

[2.2 Modules 9](#_Toc467850006)

[Verwendete Pakete 9](#_Toc467850007)

[Erstellung und Einbindung 9](#_Toc467850008)

[Config.js 10](#_Toc467850009)

[Anwendung der exportierten Funktionen und Zeichenketten 10](#_Toc467850010)

[Formulare 11](#_Toc467850011)

[Body-Parser 11](#_Toc467850012)

[Verwendete Pakete 11](#_Toc467850013)

[Einbindung und Anwendung 11](#_Toc467850014)

[Formulardaten empfangen und zurücksenden 12](#_Toc467850015)

[Ergebnis 13](#_Toc467850016)

[Pug 13](#_Toc467850017)

[Verwendete Pakete 13](#_Toc467850018)

[Pug Syntax 13](#_Toc467850019)

[Einbindung 14](#_Toc467850020)

[index.pug 14](#_Toc467850021)

[hello.pug 15](#_Toc467850022)

[Im Browser aufrufen 15](#_Toc467850023)

[Ergebnis 16](#_Toc467850024)

[Logging 16](#_Toc467850025)

[Final 16](#_Toc467850026)

[Abbildung 1: Node.js Installer Download 4](#_Toc467850057)

[Abbildung 2: RethinkDB Download 5](#_Toc467850058)

[Abbildung 3: package.json Dateierstellung beginnen mit: npm init 5](#_Toc467850059)

[Abbildung 4: Angaben der Projektinformationen und Endergebnis 6](#_Toc467850060)

[Abbildung 5: Paketinstallation 6](#_Toc467850061)

[Abbildung 6: Express in den Server einbinden 8](#_Toc467850062)

[Abbildung 7: Server „horcht“ auf localhost:3000 8](#_Toc467850063)

[Abbildung 8: Server starten mit - node index.js 8](#_Toc467850064)

[Abbildung 9: Antwort an die GET-Anfrage programmieren 9](#_Toc467850065)

[Abbildung 10: Anderen Routing-Pfad definieren 9](#_Toc467850066)

[Abbildung 11: Config-Datei einbinden 9](#_Toc467850067)

[Abbildung 12: Einfache Funktionen und Informationen exportieren 10](#_Toc467850068)

[Abbildung 13: Exportierte Funktionen mithilfe der Shorthand-Methode 10](#_Toc467850069)

[Abbildung 14: Anwendungsbeispiele unseres Moduls 11](#_Toc467850070)

[Abbildung 15: Ausgaben in der Konsole nach Serverstart 11](#_Toc467850071)

[Abbildung 16: body-parser einbinden und anwenden 11](#_Toc467850072)

[Abbildung 17: Ordner für die statischen Dateien festlegen 12](#_Toc467850073)

[Abbildung 18: Inhalt von index.html 12](#_Toc467850074)

[Abbildung 19: POST-Methode definieren 12](#_Toc467850075)

[Abbildung 20: Formulareingabe 13](#_Toc467850076)

[Abbildung 21: Ausgabe des übergebenen Wertes 13](#_Toc467850077)

[Abbildung 22: Einfache Pug Seite 13](#_Toc467850078)

[Abbildung 23: Einfache HTML Seite 14](#_Toc467850079)

[Abbildung 24: view engine auf pug setzen 14](#_Toc467850080)

[Abbildung 25: Inhalt von index.pug 14](#_Toc467850081)

[Abbildung 26: Inhalt von hello.pug 15](#_Toc467850082)

[Abbildung 27: Antwort an die GET-Anfrage programmieren 15](#_Toc467850083)

[Abbildung 28: Formulardaten mittels POST-Methode annehmen und danach ausgeben 15](#_Toc467850084)

[Abbildung 29: Formulareingabe 16](#_Toc467850085)

[Abbildung 30: Ausgabe des Übergebenen Wertes 16](#_Toc467850086)

1. Vorwort

# 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/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.

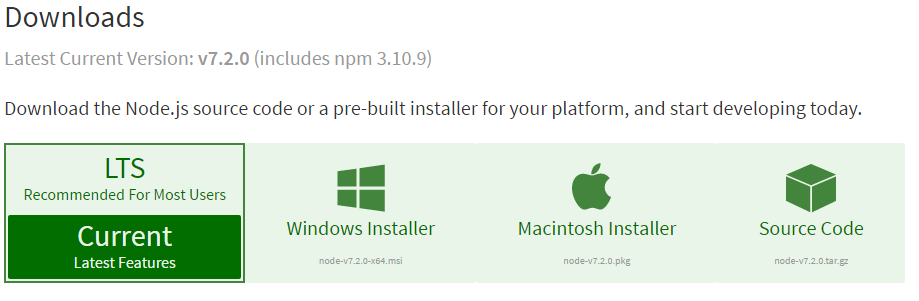


Abbildung 1: Node.js Installer Download

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 einfach entpacken 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.

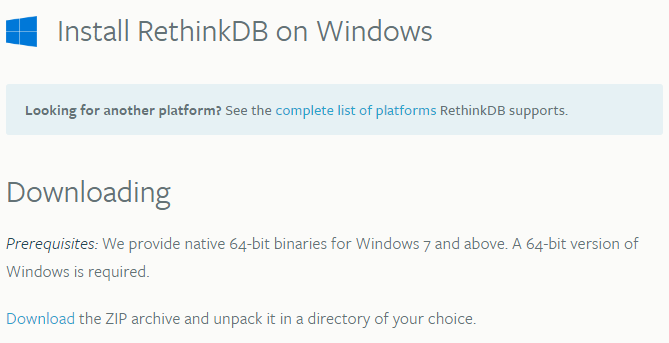


Abbildung 2: RethinkDB Download

## 1.1.3 Paket/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.



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:

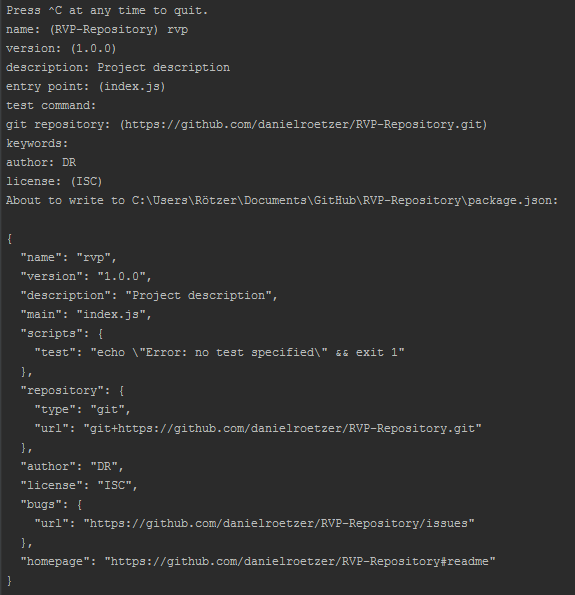


Abbildung 4: Angaben der Projektinformationen und Endergebnis

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

Nun sind die Vorbereitungen abgeschlossen.

### 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.



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 unter dem Punkt „2.1.4 Server starten“ gezeigt.

# 1.3 MVC

…

# 1.4 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 Prototypen installiert sind, wie oben unter „Installation/Node Package Manager“ beschrieben wird.

2. Prototypen

# 2.1 Hello World

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 Express einbinden

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

init_express

Abbildung 6: Express in den Server einbinden

## 2.1.3 Browser Aufruf ermöglichen

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

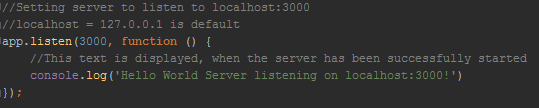


Abbildung 7: Server „horcht“ auf localhost:3000

## 2.1.4 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 8: Server starten mit - node index.js

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

## 2.1.5 Hello World im Browser ausgeben

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

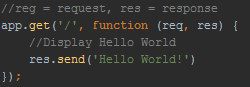


Abbildung 9: Antwort an die GET-Anfrage programmieren

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.

### 2.1.5.1 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

Abbildung 10: 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.

## 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:

config_import

Abbildung 11: 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.

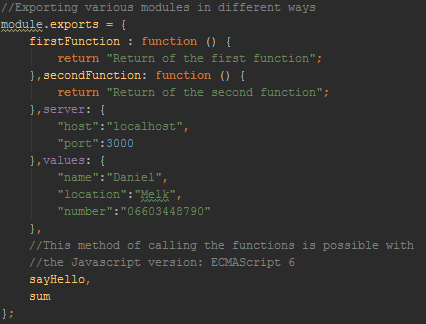


Abbildung 12: Einfache Funktionen und Informationen exportieren

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.

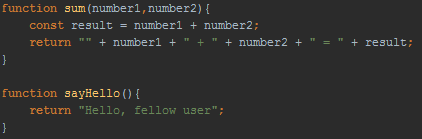


Abbildung 13: Exportierte Funktionen mithilfe der Shorthand-Methode

## Anwendung der exportierten Funktionen und Zeichenketten

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

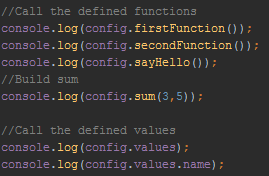


Abbildung 14: Anwendungsbeispiele unseres Moduls

Wird der Server gestartet, erhält man folgendes Ergebnis.

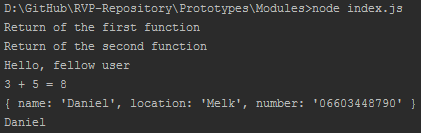


Abbildung 15: Ausgaben in der Konsole nach Serverstart

# Formulare

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

## Body-Parser

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

* express
* body-parser

### 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 16: body-parser einbinden und anwenden

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

bodyparser_view

Abbildung 17: 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.

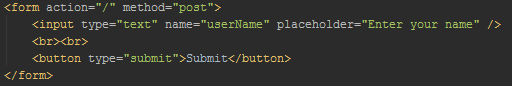


Abbildung 18: Inhalt von index.html

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.

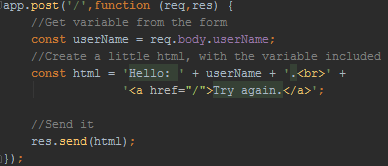


Abbildung 19: POST-Methode definieren

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).

### Ergebnis

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

|  |  |
| --- | --- |
| bodyparser_result1  Abbildung 20: Formulareingabe | bodyparser_result2  Abbildung 21: 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

* express
* pug

### 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.

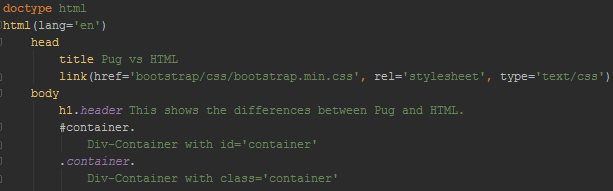


Abbildung 22: Einfache Pug Seite

Wird zu:

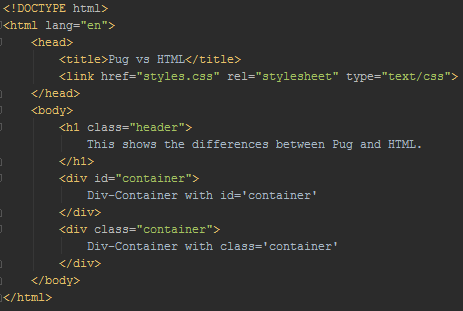


Abbildung 23: Einfache HTML Seite

### 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 24: 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 erstellt Datei, welche als Startseite dient. Die beiden Variablen „**title**“ und „**message**“ werden durch mitgesendete Werte ersetzt.

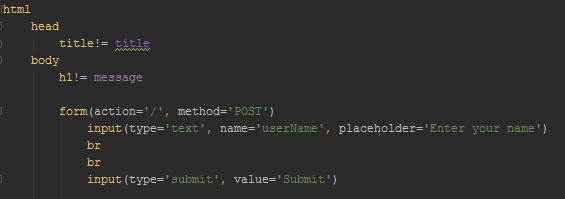


Abbildung 25: Inhalt von index.pug

### 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.

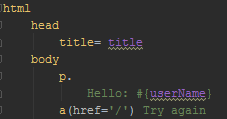


Abbildung 26: Inhalt von hello.pug

### 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.

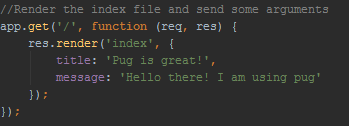


Abbildung 27: Antwort an die GET-Anfrage programmieren

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

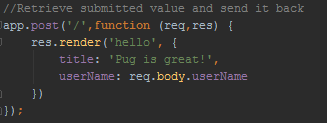


Abbildung 28: Formulardaten mittels POST-Methode annehmen und danach ausgeben

### Ergebnis

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 29: Formulareingabe | C:\Users\Rötzer\AppData\Local\Microsoft\Windows\INetCache\Content.Word\pug-ergebnis2.png  Abbildung 30: Ausgabe des Übergebenen Wertes |

# Logging

…

# Final