Hochschule Neubrandenburg
University of Applied Sciences

Hochschule Neubrandenburg
Studiengang Geoinformatik

Prototypische Entwicklung einer Software-Lösung zur Verwaltung von Sensor-Netzwerken am GFZ Potsdam auf Basis von Node.js

Bachelorarbeit

vorgelegt von: *Paul David*

Zum Erlangen des akademischen Grades
„Bachelor of Engineering“ (B.Eng.)

Erstprüfer: Prof. Dr.-Ing. Andreas Wehrenpfennig
Zweitprüfer: Prof. Joachim Wächter

Bearbeitungszeitraum: 15.01.2018 bis 12.03.2018

URN: **urn:nbn:de:gbv:519-thesis2018-0001-4**

Kurzfassung

Diese Arbeit beschäftigt sich mit der Schaffung einer prototypischen Softwarelösung zur Verwaltung und Umsetzung von Sensornetzwerken. Diese Software stützt sich auf JavaScript und die JavaScript-Laufzeitumgebung Node.js. Im Ergebnis wird ein Testaufbau vorgestellt. In diesem ist ein Mikrocontroller mit einem angeschlossenen Sensor für die Messung der Umweltparameter veranwortlich. Die Befehle für diese Messung erhält der Mikrocontroller von der zuständigen Basisstation, welche wiederum die Ergebnisse an eine Web-Schnittstelle weiterleitet. Diese kümmert sich um die Speicherung der Daten in einer verbundenen Datenbank sowie um die Auslieferung der Daten zur Visualisierung.

Abstract

This bachelor thesis focusses on the creation of a prototypical software solution for the management of sensor networks. The software relies on JavaScript and the JavaScript runtime Node.js. As a result, a test setup is presented. This setup contains a microcontroller with a connected sensor which is responsible for the measurement of the environmental parameters. The microcontroller receives the commands for this measurement from the responsible base, which then passes the results to a web interface. This interface takes care of storing of the data in a connected database and provides the data for visualization.

Danksagung

An dieser Stelle möchte ich all jenen danken, die durch ihre fachliche und persönliche Unterstützung zur Erstellung dieser Arbeit beigetragen haben.

Zuerst gebührt mein Dank Herrn Prof. Wehrenpfennig und Herrn Prof. Wächter, die diese Arbeit betreut und begutachtet haben.

Meiner Familie und meiner Partnerin gebührt zudem Dank für die moralische Unterstützung und die aufgebrachte Geduld während der Erstellung dieser Arbeit.

Inhaltsverzeichnis

1 Einleitung

In vielen Bereichen des heutigen Lebens kommen Sensornetzwerke zum Einsatz - von Klimamonitoring über die Überwachung von Bauwerken bis hin zu Katastrophenfrühwarnsystemen.
In solchen Frühwarnsystemen messen Sensoren zahlreiche Parameter, die Vorhersagen anstehender Katastrophen ermöglichen. Personen vor Ort können informiert und die betroffenen Gebiete evakuiert bzw. geschützt werden. Am GeoForschungsZentrum Potsdam werden solche Systeme oft abteilungsübergreifend und auf Basis diverser Programmiersprachen entwickelt und eingesetzt.

Bisher wurde die sensorische Steuerung durch in C programmierte Skripte umgesetzt, das Backend in Java oder ähnlichen Sprachen sowie das Frontend bzw. die Nutzeroberfläche mit Hilfe darauf spezialisierter Sprachen, wie z.B. einer Web-Oberfläche auf Basis von HTML, CSS, PHP und JavaScript, gebildet.

Was aber passiert, wenn die Zuständigkeiten in den Teams wechseln? Müssen die Entwickler in einem solchen Fall in der jeweils genutzten Sprache geschult werden? Wie viele Ressourcen werden dadurch gebunden? Was passiert, wenn eine der genutzten Sprache nicht oder nur gering weiter entwickelt wird? Ist die Funktionsfähigkeit des Systems in diesem Fall weiterhin gewährleistet?

Die technische Entwicklung macht es mittlerweile möglich, mit nur einer Programmiersprache mehrere der genannten Probleme anzugehen. Wenn sich das oben genannte Frühwarnsystem nicht mehr vordergründig auf die Bewältigung eventueller Konflikte zwischen Programmiersprachen konzentrieren muss, kann der Fokus auf die Optimierung gelegt werden. Dazu gehören zum Beispiel die Präzision der Vorhersagen oder die Reaktions- und Benachrichtigungszeiten während einer Katastrophe.

Während meines Industriepraktikums war es daher meine Aufgabe, eine Software für ein Sensornetzwerk zu entwickeln, die auf möglichst wenig Programmiersprachen basiert. Der Fokus lag dabei auf *JavaScript* bzw. der JavaScript-Laufzeitumgebung *Node.js*.

Dazu werden im ersten Teil der Arbeit allgemeine Grundlagen zum Thema Sensornetzwerke behandelt sowie Anforderungen an dasselbe spezifiert. Zu diesem Zweck werden die oben angedeuteten Probleme analysiert und auf ausgewählte Aspekte beschränkt.

In einer Auseinandersetzung mit beispielhaften Systemen wie *TERENO* und

DABAMOS werden bisherige Vorgehensweisen des Aufbaus analysiert. *TERENO* nimmt dabei eine größere Priorität ein, da das in dieser Arbeit zu entwickelnde System eine Erweiterung darstellen soll.

Auf Grundlage der daraus gewonnen Erkenntnisse wird eine prototypische Softwarelösung entworfen. Dabei spielen Ideen zu Architektur, Datenfluss und Gestaltung einzelner Komponenten eine Rolle.

Im Anschluss wird die prototypische Umsetzung dieser Ideen vorgestellt. Im Einzelnen werden dabei Funktionsweise, Aufbau und Kommunikation der Komponenten untereinander beschrieben sowie die für die Umsetzung genutzten Module erläutert.

Das darauf folgende Kapitel beschäftigt sich mit der Demonstration der erzielten Ergebnisse. Dabei werden sowohl Elemente des Testaufbaus als auch die Funktionsweise des Backends und der Aufbau des Frontends visualisiert.

Abschließend werden die Ergebnisse mit den Anforderungen und den zu Beginn gesetzten Zielen abgeglichen. Darüber hinaus wird ein Ausblick für zukünftige Erweiterungs- bzw. Verbesserungsmöglichkeiten gegeben.

2 Problemstellung

In diesem Kapitel werden die für diese Arbeit grundlegenden Begriffe definiert, Problemstellungen von Sensornetzwerken analysiert und Anforderungen für das zu schaffende System beschrieben.

2.1 Was ist ein Sensornetzwerk?

Ein Sensornetzwerk ist ein aus kommunizierenden Knoten aufgebautes Netzwerk. Die Knoten bilden dabei Messstationen, die aus einem zentralen Knoten und mindestens einem Mikrocontroller und daran angeschlossenen Sensoren besteht. Die Mikrocontroller übernehmen dabei die unmittelbare Steuerung bzw. das Auslesen der Sensoren und leiten diese Daten an den zentralen Knoten bzw. die Basisstation weiter. Die Kommunikation kann dabei über übliche Kommunikationswege aufgebaut werden, wie z.B. über serielle Schnittstellen, WiFi oder kabelgebundene Netzwerke. Jedoch kann diese auch durch andere Funk-Kommunikationsmittel, wie z.B. LoRa[1], über größere Distanzen hinweg ausgebreitet werden. Nachdem die Daten von der Basisstation empfangen und verarbeitet wurden, können diese mit Hilfe des Internets an eine Schnittstelle weitergeleitet werden, die diese Daten z.B. in einer Datenbank speichert.
Solche Sensornetzwerke sind häufig hoch komplexe Systeme, in denen eine große Vielzahl an diversen Programmiersprachen zum Einsatz kommt.

(Vgl. Mattern; Römer, 2003)

2.2 Was ist ein Mikrocontroller?

Mikrocontroller (kurz: MCU für microcontroller unit) sind kleine Computer, die sich auf einer Platine bzw. einem einzigen Schaltkreis befinden. Ihre Bestandteile sind:

- Prozessor (1 oder mehr Kerne)
- Speicher (üblicherweise Flash)
- Anschlüsse für Input und Output

Mikrocontroller werden genutzt, um die Kommunikation mit bzw. die Steuerung der Hardware, also z.B. den Sensoren, herzustellen. Dazu können

[1] „LoRaWAN (kurz für: Long Range Wide Area Network) ist eine „Low Power Wide Area Network“ (kurz: LPWAN) Spezifikation für drahtlose batteriebetriebene Systeme in einem regionalen, nationalen oder auch globalen Netzwerk. Die Spezifikation zielt dabei vor allem auf Anforderungen des Internet of Things.“ (Vgl. LoRa Alliance™, 2018)

sowohl analoge Signale als auch Schnittstellenprotokolle, wie z.B. I2C[2], genutzt werden.
Nicht zu verwechseln mit Mikrocontrollern sind Mikroprozessoren. Diese beiden Begriffe werden häufig als Synonyme genutzt, obwohl es signifikante Unterschiede gibt.
Während Mikrocontroller Prozessor, Arbeitsspeicher und weitere Peripheriegeräte auf einem Chip kombinieren bzw. integrieren, umfassen Mikroprozessoren nur eine CPU[3] und evtl. Cache-Speicher. Alle weiteren Komponenten werden später über Schnittstellen hinzugefügt.
Weiterhin unterscheiden sich die Geräte in den Taktgeschwindigkeiten. So arbeiten Mikrocontroller meist unter 100 MHz, Mikroprozessoren meist über 1 GHz. Zum Einsatz kommen Mikroprozessoren vor allem in Desktop-PC's, Notebooks, Tablets und Handys. (Vgl. Choudhary, 2012)
Ähnlich zu Mikrocontrollern sind auch Einplatinencomputer bzw. Systems-on-a-chip (kurz: SoC) Systeme, die die Komponenten eines Computers auf einer einzelnen Platine bzw. einem einzigen Schaltkreis verbinden. Üblicherweise bieten diese jedoch mehr Hardware, wie z.B. eine GPU, WLAN-Module, USB- und Ethernet-Anschlüsse oder zusätzliche Prozessoren oder Mikrocontroller. Hinzu kommt, dass nicht alle SoCs integrierten persistenten Speicher bieten und so beispielsweise mit Speichermedien, wie z.B. SD-Karten, bestückt werden müssen.
Bekanntestes Beispiel für einen Einplatinencomputer ist der Raspberry Pi.

(Vgl. Brinkschulte; Ungerer, 2010)

[2](kurz für: Inter-Integrated Circuit) 1982 von Philips Semiconductors entwickelter serieller Datenbus zur geräteinternen Kommunikation zwischen Teilen von integrierten Schaltungen ((Vgl. Dembowski, 2015))

[3]kurz für: Central Processing Unit; deutsch: zentrale Verarbeitungseinheit (kurz: ZVE) oder auch Prozessor

2.3 Problembeschreibung und Ziele

Der Betrieb von Sensornetzwerken birgt verschiedene Herausforderungen.

- Nutzung diverser Programmiersprachen
- Automatisierung
- Konnektivität
- Langlebigkeit der Komponenten
- Stromversorgung der Komponenten

Nutzung diverser Programmiersprachen

Wie bereits im Vorhergehenden beschrieben, finden in komplexen Systemen, wie Sensornetzwerken, häufig eine große Bandbreite verschiedener Programmiersprachen Anwendung. Dadurch fällt es oft schwer, einen Überblick und ein Verständnis für das gesamte System zu bewahren.
Einige Programmier- und Beschreibungssprachen, die dabei zum Einsatz kommen können, sind im Folgenden aufgelistet:

- Steuerung von Sensoren:
  - C
  - Python
- Backend bzw. Web-Schnittstelle:
  - C / C++
  - Java
  - Python
  - Ruby
  - Go
  - JavaScript / Node.js
- Datenbank:
  - SQL
- Frontend bzw. Visualisierung:
  - HTML
  - PHP
  - JavaScript
  - CSS
  - Java
  - C, C++, C#
  - Python
  - Go

Viele Programmiersprachen sind für die Erfüllung bestimmter Ziele abgestimmt und entwickelt worden.
In großen Projekten ist es bisher üblich, dass die Entwicklung in diverse

Teams aufgeteilt wird. Jedes dieser Teams bearbeitet eine spezialisierte Teilaufgabe des jeweiligen Projekts und nutzt somit unter Umständen die für die Aufgabe am besten geeignete Sprache.
Um dieses Problem zu beheben, sollen die Komponenten des Systems weitestgehend auf einer einheitlichen Programmiersprache basieren.

Automatisierung

Manche Messstationen von Sensornetzwerken erfordern bis zum heutigen Tag ein manuelles Eingreifen, z.B. zum Auslesen der Daten, durch Manpower. Da solche Stationen oft in abgeschiedenen Gebieten liegen, verlangt das Eingreifen nicht nur Zeit, sondern auch Kosten für Anreise o.ä.
Das Ziel sollte somit die vollständige Automatisierung des Workflows vom Auslesen bis zur Speicherung der Daten eines solchen Systems sein.

Konnektivität

Die Messtechnik in solchen Netzwerken nutzt häufig proprietäre Anschlüsse. Dadurch wird der Aufbau der Software meist verkompliziert. Sie muss in solchen Fällen in der Lage sein, die diversen Protokolle dieser Anschlüsse zu unterstützen.
Ein Lösungsansatz wäre die Nutzung einheitlicher Anschlüssmöglichkeiten und eines einheitlichen Kommunikationprotokolls.

Langlebigkeit der Komponenten

Die Komponenten sollten eine lange Lebensdauer aufweisen, um Kosten durch Wartung und Reparatur zu minimieren. Dieses Problem ist jedoch nicht Gegenstand dieser Arbeit.

Stromversorgung der Komponenten

In abgeschiedenen Gebieten ist die Stromversorgung oft nicht zu gewährleisten. Sollte diese nicht durch das Stromnetz gegeben sein, wäre die autonome Versorgung der Messstationen wünschenswert. Auch dieses Problem ist nicht Teil dieser Arbeit.

2.4 Anforderungen

Die beschriebenen Probleme spezifizieren die folgenden Anforderungen an das System:

- Nutzung Netzwerk-fähiger Mikrocontroller
- Internetzugang für die Basisstationen
- Automatisierter Workflow
- Messungen in - für das Szenario realistischen - regelmäßigen Abständen
- Ergonomische, nutzerfreundliche Oberfläche
- Abdeckung großer geographischer Bereiche
- Autonome Stromversorgung in abgeschiedenen Gebieten (Akku + Solar)
- Möglichst langlebige Komponenten
- Möglichst günstige Komponenten
- Wetterbeständigkeit der Messinstrumente

(Vgl. Mainwaring; Culler u. a., 2002; Nebel; Awad; German; Dressler, 2007)

Die fünf zuletzt genannten Anforderungen sind nicht Gegenstand dieser Arbeit.

3 Vorstellung beispielhafter Systeme

Aus der Vielzahl existierender Sensornetzwerke sollen in diesem Kapitel zwei beispielhafte Systeme näher beleuchtet werden. Dabei werden inhaltliche Ausrichtung und Struktur betrachtet.

3.1 TERENO

TERENO (kurz für: **Ter**restrial **En**vironment **O**bservatoria) ist eine Initiative der Helmholtz-Gemeinschaft und „zielt auf die Schaffung einer Beobachtungsplattform, die verschiedenste terrestrische Observatorien in unterschiedlichen Regionen verbindet.“ (Sauer, 2016)

3.1.1 Inhaltliche Ausrichtung

Die Beobachtungsplattform soll eine Serie langfristiger statistischer Systemvariablen für die Analyse und Vorhersage der Folgen des globalen Wandels liefern.
Dafür werden integrative Modellierungssysteme eingesetzt, um wirksame Präventions-, Vermeidungs- und Anpassungsstrategien zu entwickeln.
Wichtige Systemvariablen dafür sind:

- Verlagerungs- und Austauschprozesse von Wasser, Materie und Energie im Kontinuum des Grundwasser-Boden-Vegetation-Atmosphäre-Systems,
- langfristige Veränderungen des Aufbaus und der Arbeitsweise von Mikroorganismen, der Flora und Fauna,
- sozioökonomische Rahmenbedingungen, die mit ausreichender zeitlicher und räumlicher Auflösung bestimmt werden müssen.

Die Ergebnisse der Observationen dienen hauptsächlich der Klärung folgender wichtiger Fragen:

- Welche Folgen haben die erwarteten Klimaänderungen auf die terrestrischen Kompartimente (Grundwasser, Böden, Vegetation, Oberflächengewässer)?
- Wie beeinflussen Rückkopplungsmechanismen der Austauschprozesse terrestrischer Systeme (z.B. Rückkopplungen zwischen der Erdoberfläche und der Atmosphäre) die terrestrischen Wasser- und Materieflüsse?
- Welche direkten Einflüsse haben Veränderungen der Boden- und Landnutzung (z.B. infolge der Cross-Compliance-Richtlinie der EU zur Förderung von Energiepflanzen) auf den Wasserhaushalt, die Bodenfruchtbarkeit, die Biodiversität und das regionale Klima?

- Was sind die Auswirkungen großflächiger anthropogener Eingriffe (z. B. Tagebau, Abholzung) auf terrestrische Systeme?

Die von den Observatorien produzierten Datensätze sollen folgende Aufgaben erfüllen:

- Beitrag zur Validierung, Weiterentwicklung und Integration von terrestrischen Modellen (z.B. Grund- und Bodenwasserhaushaltsmodelle, Modelle zur regionalen Klima- und Wetterprognose, Luftqualitäts- und Abflussratenmodelle, forst- und agrarökonomische Modelle sowie Modelle zur Biodiversifikation und sozioökonomische Modelle)
- Unterstützung der Verwaltung land- und forstwirtschaftlicher Ökosysteme mit integrativen Modellierungsverfahren (z.B. durch die Optimierung von Bewässerungssystemen sowie die Entwicklung von Frühwarnsystemen für Unwetter und Überschwemmungen, integrierte Kontrollsysteme für wasserwirtschaftliche Ausbauten sowie Systeme zur Überwachung der Qualität von Luft, Grundwasser und Oberflächengewässern)

(Vgl. Forschungszentrum Jülich GmbH, 2017c)

3.1.2 Struktur

In TERENO arbeiten die vier folgenden deutschen Observatorien zusammen:

- Observatorium Eifel / Niederrheinische Bucht (Koordination: FZJ)
- Observatorium Harz / Mitteldeutsches Tiefland (Koordination: UFZ)
- Observatorium bayerische Alpen / Voralpenland (Koordination: KIT und HMGU)
- Observatorium Norddeutsches Tiefland (Koordination: GFZ)

(Vgl. Forschungszentrum Jülich GmbH, 2017b)

Diese Observatorien arbeiten sowohl mit festen Messstationen als auch mit mobilen Messplattformen und bedienen sich dabei folgender Ausstattung.

- Mess-Systeme zur Bestimmung regionaler Niederschlagsfelder auf räumlicher und zeitlicher Ebene (z.B. durch Einsatz zusätzlicher Wetterradare oder Verdichtung der Netzwerke zur Niederschlagsmessung)
- Mikrometeorologisches Eddy-Kovarianz-System und Szintillometer zur Bestimmung der atmosphärischen Parameter und Austauschraten von Wasserdampf-, Energie und Spurengasen in unterschiedlichem Maßstab

- Sensornetzwerke zur Bestimmung der Umweltparameter mit hoher räumlicher und zeitlicher Auflösung (z.B. Verteilungsmuster dynamischen Verhaltens von Bodentemperatur und -feuchte)
- Monitoringsysteme zur Quantifizierung der Abflussrate von Wasser und gelösten Stoffen

Zudem umfassen die Observationen folgende Bereiche der Fernerkundung:

- Betrieb und Weiterentwicklung bodengebundener und luftgestützter Messplattformen (z.B. Türme, Ultraleichtflugzeuge)
- Anschaffung geophysikalischer und spektraler Sensoren (z.B. hyperspektrale Systeme und Infrarotkameras, Mikrowellenradiometer, SAR, LIDAR-, luftchemische Instrumente)

Nicht zuletzt bedarf die Vielzahl der durch die Observationen gesammelten Daten mehrerer hoch performanter Systeme zur Datenverarbeitung und Kommunikation, um schnelle Verfügbarkeit und langfristigen Schutz der Daten zu gewährleisten. Der Aufbau dieses Systems ist Abbildung 1 zu entnehmen.

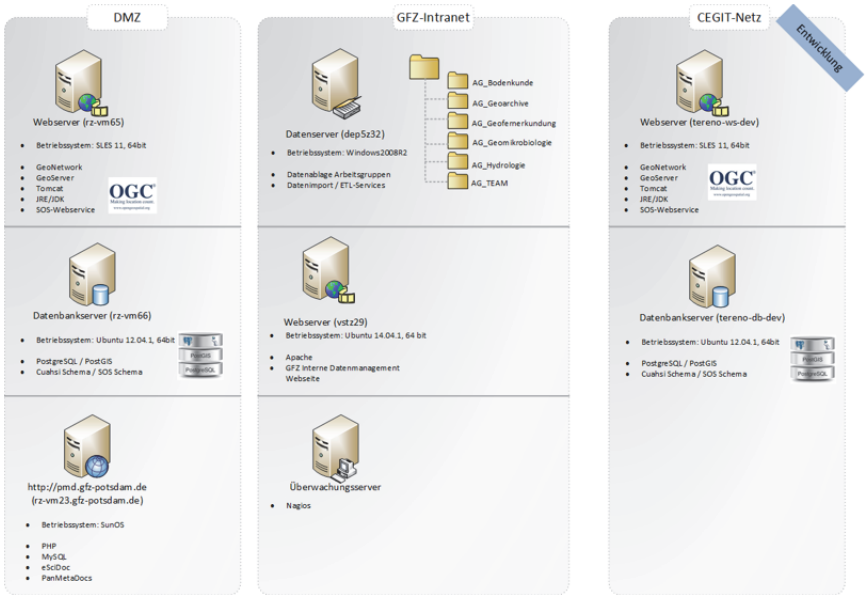


Abbildung 1: Übersicht über von TERENO genutzte Server (Stender, 2017)

(Vgl. Forschungszentrum Jülich GmbH, 2017a)

Der Datenfluss innerhalb des gesamten Systems ist in Abbildung 2 dargestellt.

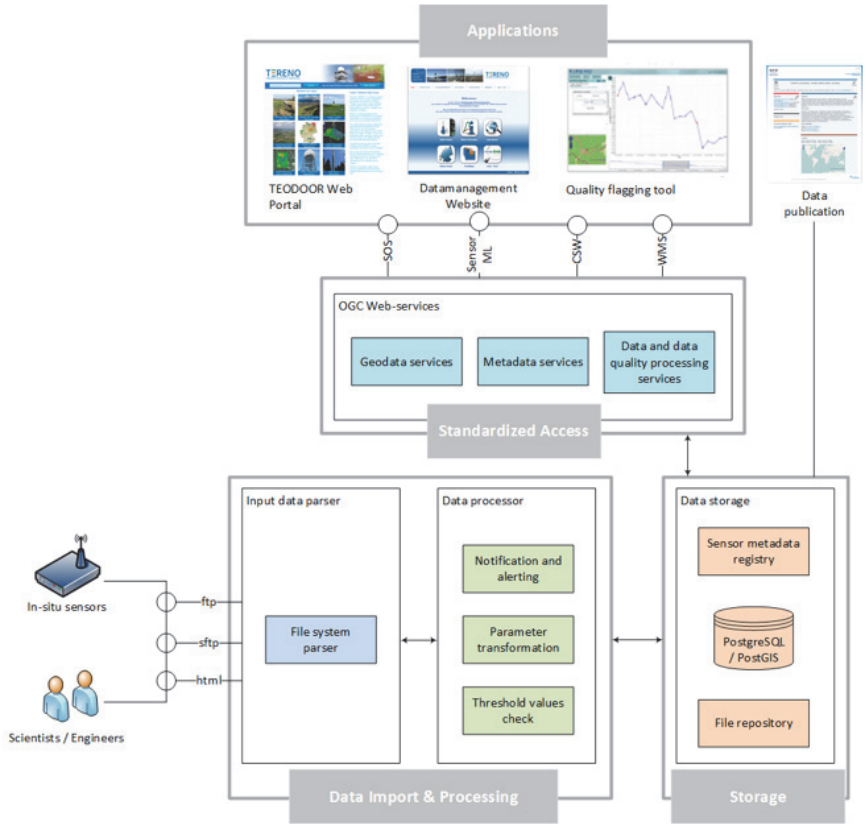


Abbildung 2: Schematischer Aufbau des TERENO (Stender, 2017)

3.2 DABAMOS

In diesem Abschnitt soll ein System mit ähnlicher Zielsetzung vorgestellt werden. Dieses System ist das „Datenbank-orientierte Monitoring- und Analyse-System“ (kurz **DABAMOS**).

3.2.1 Inhaltliche Ausrichtung

DABAMOS ist ein open-source Monitoring-System zur automatisierten „Überwachung von Bauwerken und Geländeabschnitten mit Hilfe geodätischer und geotechnischer Sensoren.“ (Engel, 2013)

3.2.2 Struktur

Der physische Aufbau des Systems ist schematisch in Abbildung 3 dargestellt.

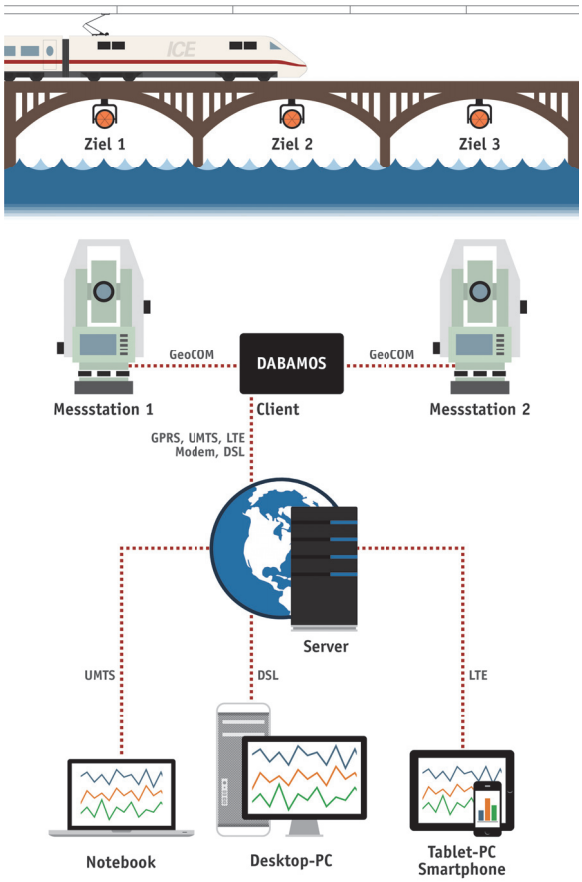


Abbildung 3: Schematischer Aufbau des DABAMOS-Netzes

Ursprünglich entwickelt wird DABAMOS seit 2009 an der Hochschule Neubrandenburg. Erweitert wurde es 2013 im Rahmen der Masterarbeit von Philipp Engel. (Siehe auch: Engel, 2013)
Ein Vergleich der genutzten Technologien der Versionen von 2009 und von 2013 ist in Tabelle 1 ersichtlich.

		2009	**2013**
Sprache	**Mess-Client**	Java SE	Go
	Server	Java Enterprise	Go
	Frontend	XHTML & AJAX	HTML incl. Go-Backend
Datenbank		db4o	MongoDB
Schnittstelle		JavaServer Faces	REST

Tabelle 1: Vergleich der Versionen von DABAMOS (Vgl. Engel, 2013)

4 Entwurf eines Lösungsansatzes

Dieses Kapitel bietet einen Überblick über das gesamte System sowie eine Betrachtung der Funktionsweise einzelner Komponenten und die Kommunikation dieser untereinander.

4.1 Überblick des gesamten Aufbaus

Zu Beginn der Betrachtungen soll ein kurzer Überblick über die Funktionsweise des Systems gegeben werden. Dabei werden zunächst die Architektur und der Datenfluss dargestellt.

4.1.1 Architektur

Das System erstreckt sich über mehrere Ebenen. Diese sind verantwortlich für die Messung, das Speichern und die Darstellung von Umweltdaten. Abbildung 4 stellt einen schematischen Überblick des gesamten Systems dar.

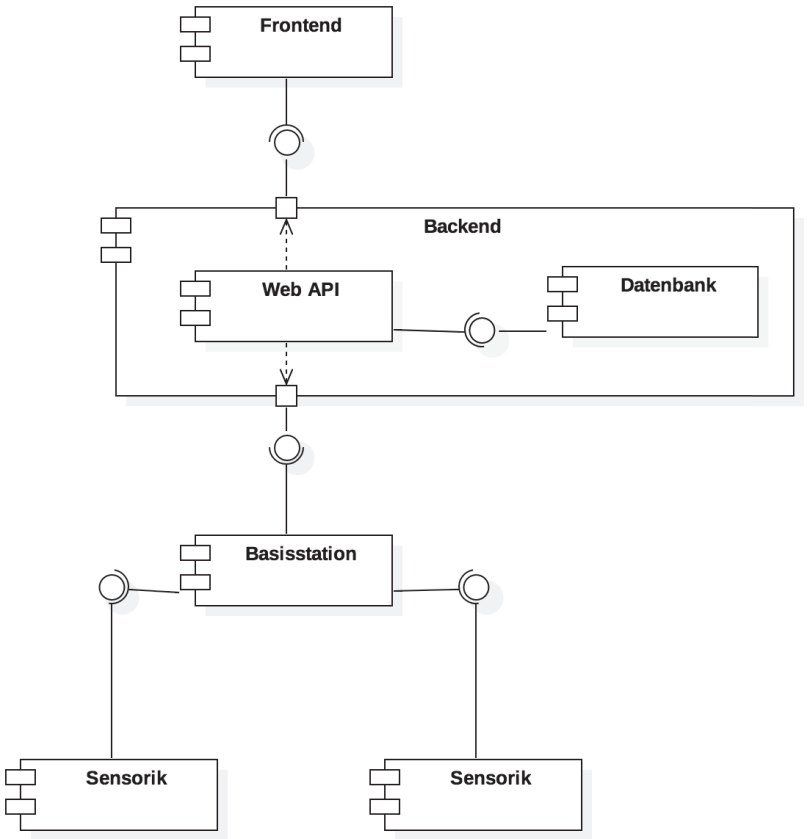


Abbildung 4: Schema der Architektur bzw. des Aufbaus des Systems

Die Architektur beinhaltet mehrere Komponenten:

Die Sensorik stellt den direkten Kontakt mit der Natur dar. Sie wird genauer in Kapitel 4.2.1 beleuchtet.

Die Basisstation vermittelt zwischen der Sensorik und dem Backend und wird in Kapitel 4.2.2 genauer beschrieben.

Das Backend umfasst die Web-Schnittstelle und die Datenbank. Es dient zur Speicherung und Wiedergabe von Daten und wird in Kapitel 4.2.3 vorgestellt.

Das Frontend dient der Visualisierung der Daten. Es wird in Kapitel 4.2.4 behandelt.

4.1.2 Datenfluss

Vom Auslesen bis hin zur Speicherung der Daten in der Datenbank durchlaufen diese einige Stationen. In den Abbildungen 5 und 6 wird der Datenfluss innerhalb des Systems visualisiert. Abbildung 5 stellt dabei die Aktivitäten sowie die Entscheidungen, Abbildung 6 die zeitliche Abfolge der Kommunikation der Komponenten untereinander dar. Da der Fluss in den dargestellten Komponenten in regelmäßigen Abständen stattfindet, jedoch keine Abfrage durch das Frontend bzw. den Nutzer voraussetzt oder benötigt, wurde diese Komponente zunächst nicht betrachtet.

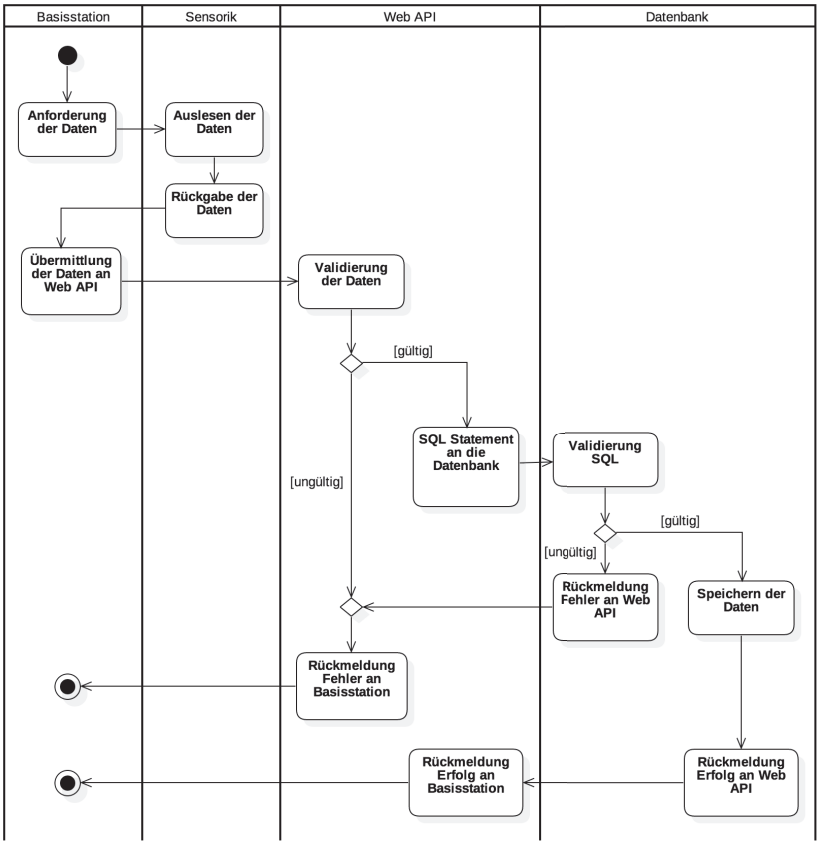


Abbildung 5: Datenfluss innerhalb des Systems

Der in Abbildung 5 und 6 dargestellte Prozess beginnt auf der Ebene der Basisstation. Diese übermittelt den Mikrocontrollern den Befehl zum Auslesen der Sensoren. Nachdem die Werte zurückgegeben wurden, sendet die Basisstation diese mittels HTTP-Post an die Web-Schnittstelle. Dort werden die Daten zunächst validiert. Sollten diese nicht gültig sein, wird eine Fehlermeldung in Form eines HTTP-Statuscodes zurückgegeben. Wenn die Daten jedoch gültig sind, wird ein Befehl erzeugt und abgesetzt. Dieser veranlasst die Kommunikation mit der Datenbank und die Speicherung der Daten in selbiger. Diese meldet daraufhin das Ergebnis der Operation an die Web-Schnittstelle zurück, so dass diese eine entsprechende Meldung an die Basisstation zurückgeben kann.

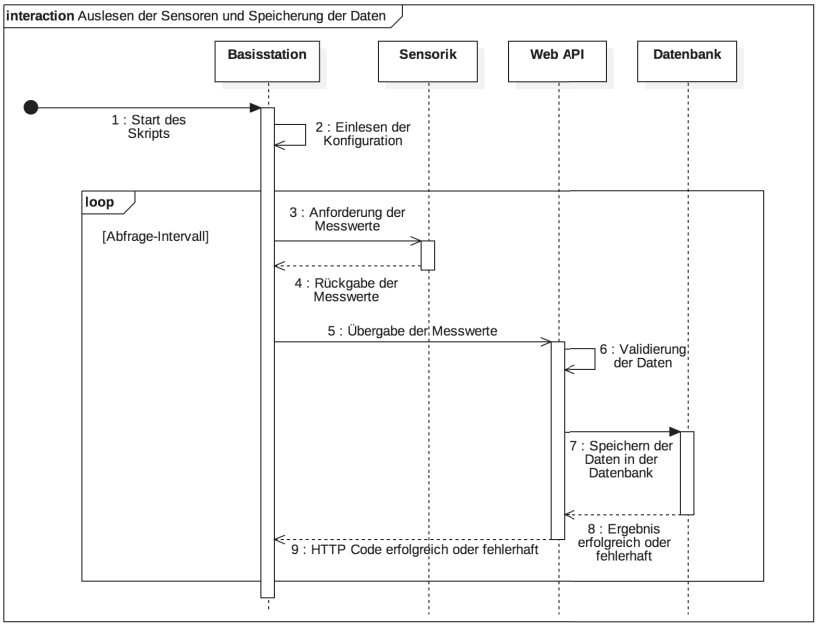


Abbildung 6: Datenfluss innerhalb des Systems

Bei Abfrage der Daten durch einen Nutzer bzw. das Frontend gestaltet sich das Zusammenspiel aus Aktivitäten und Entscheidungen wie in Abbildung 7 aufgezeigt.

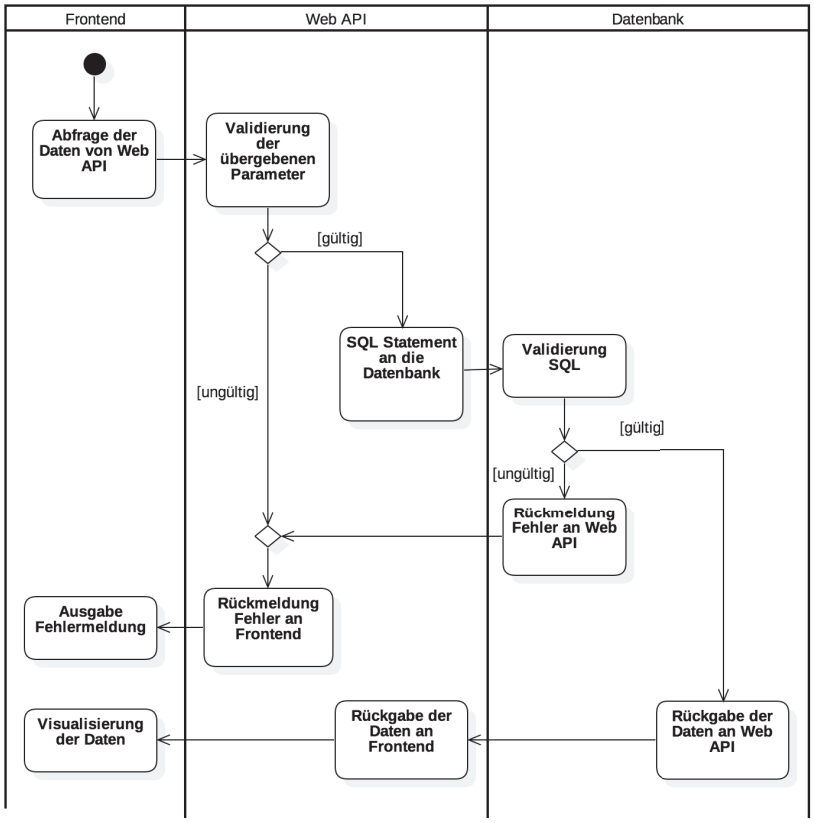


Abbildung 7: Darstellung des Datenflusses während einer Abfrage durch das Frontend

Beim Start des Programms werden die Daten zunächst mit den Ausgangs-Parametern abgefragt. Das Frontend stellt also einen HTTP-Request an die Web-Schnittstelle. Diese überprüft die übergebenen Parameter, liefert bei Erfolg die Daten aus der Datenbank, andernfalls eine entsprechende Fehlermeldung. Anschließend übergibt die Web-Schnittstelle die erhaltenen Daten an das Frontend, welches daraufhin die Daten visualisiert bzw. rendert.

Bei der Änderung von Parametern, wie z.B. dem abzufragenden Datum, durch den Nutzer, wird dieser Prozess erneut durchlaufen.
Die zeitliche Abfolge der Kommunikation der Komponenten untereinander ist in Abbildung 8 dargestellt.

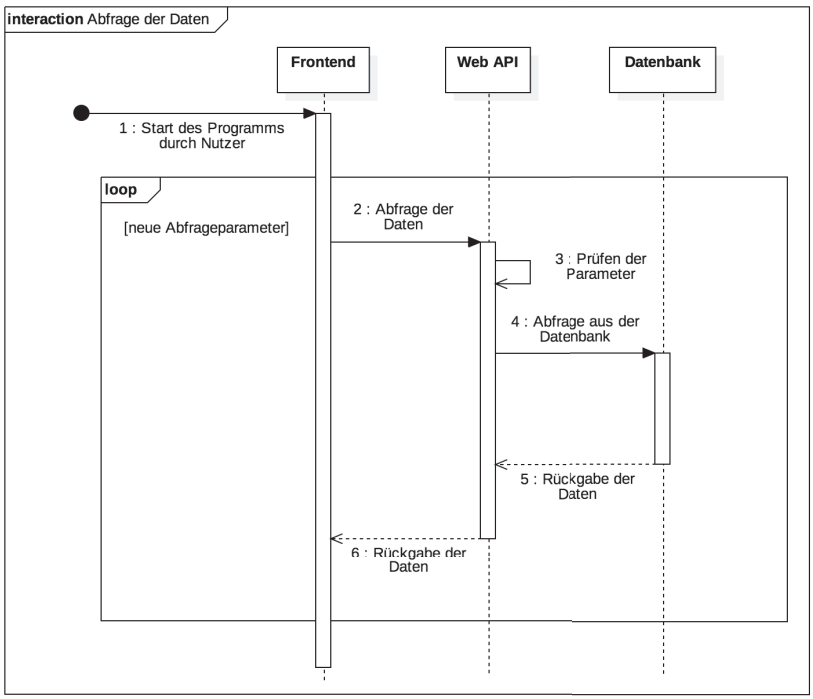


Abbildung 8: Darstellung des Datenflusses während einer Abfrage durch das Frontend

4.2 Betrachtung einzelner Komponenten

4.2.1 Sensorik

Ein beispielhafter schematischer Aufbau der Sensorik ist in Abbildung 9 dargestellt. Dieser besteht aus einem Mikrocontroller und zwei daran angeschlossenen Sensoren. Möglich ist dieser Aufbau nur, wenn der genutzte Mikrocontroller genügend Anschlüsse bietet, da einige nur die Möglichkeit zum Anschluss eines Sensors über eine analoge Schnittstelle besitzen und somit bei der Nutzung von zwei analogen Sensoren nicht geeignet wären.

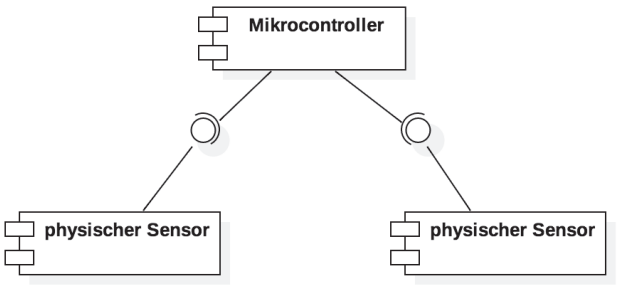


Abbildung 9: Schematischer Aufbau der Sensorik

4.2.2 Basisstation

Die Basisstation übernimmt in dem System die Abfragen zur Beschaffung der Umweltparameter und das Versenden dieser Daten an die Web-Schnittstelle. Dazu wird ein Netzwerk bereitgestellt, in das die mit den Sensoren verbundenen Mikrocontroller integriert werden. Über dieses Netzwerk wird die Kommunikation zwischen den Geräten gesichert, die die Abfrage der Daten ermöglicht. Abbildung 10 zeigt einen beispielhaften schematischen Aufbau einer Basisstation.

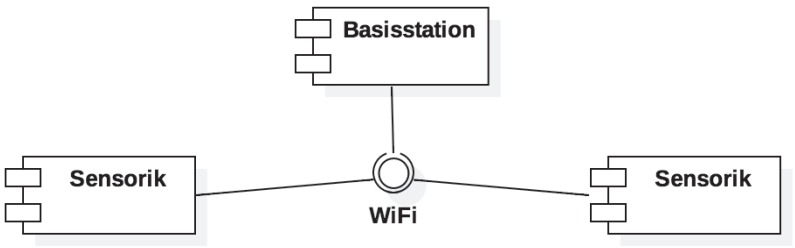


Abbildung 10: Schematischer Aufbau zur Verbindung der Sensorik und der Basisstation

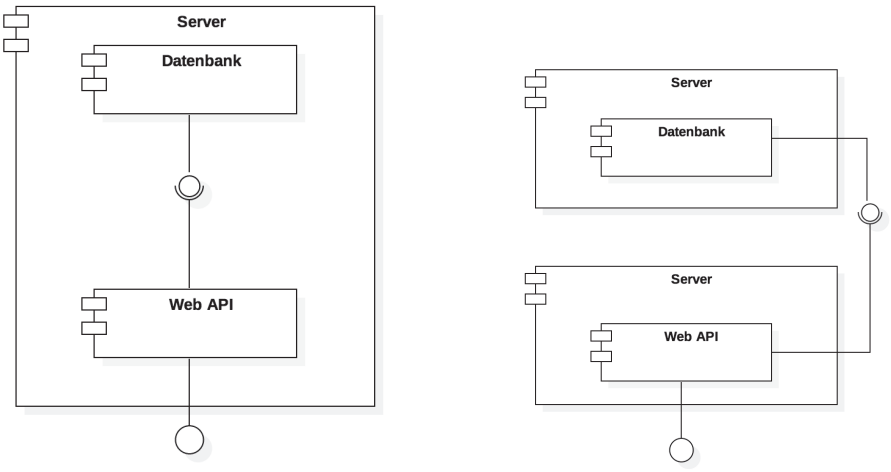
Der in der Abbildung gezeigte Aufbau beinhaltet ein drahtloses Netzwerk. Grundsätzlich ist jedoch auch die Nutzung eines kabelgebundenen Netzwerkes möglich.

4.2.3 Backend mit Web-Schnittstelle und Datenbank

In dieser Ebene geht es darum, eine Datenbank und eine Schnittstelle zu dieser im Internet bereitzustellen.
Das System sollte hier möglichst modular gestaltet werden, um zu ermöglichen, dass das Datenbanksystem bei Bedarf modifiziert oder sogar durch ein anderes ersetzt werden könnte.

Physischer Aufbau des Backends

Der physische Aufbau kann variabel gestaltet werden. Grundsätzlich ist es möglich Datenbank und Schnittstelle auf einem Computer bzw. Server bereitzustellen. Genauso denkbar ist die Betreibung von Datenbank und Schnittstelle auf getrennten Servern. Diese Arten des Aufbaus sind in Abbildung 11 dargestellt.



(a) Aufbau mit einem Server (b) Aufbau mit zwei Servern

Abbildung 11: Schematischer Aufbaus mit einem und zwei Servern

Des Weiteren ist die Nutzung von Clustering-Möglichkeiten denkbar, die es erlauben, die eintreffenden Anfragen von diversen Servern bearbeiten zu lassen und so die Last aufzuteilen. In Kombination mit einer Datenbank, die ebenfalls die Möglichkeit des Clusterings bietet, kann ein auf Seiten der Hardware skalierbares Backend geschaffen werden, welches auf Zu- und Abnahme der Nutzerzugriffe angepasst werden kann. Der Aufbau eines solchen Systems ist in Abbildung 12 dargestellt.

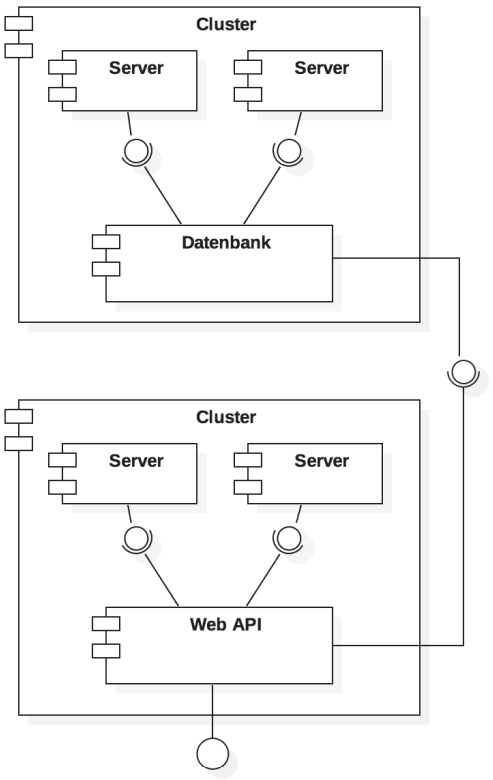


Abbildung 12: Schematischer Aufbau eines Server-Clusters

Datenbank

Das Datenbankmanagementsystem ist entsprechend der Bedürfnisse frei wählbar. Je nach Ansprüchen können dazu SQL[4]- oder NoSQL[5]-Datenbanken genutzt werden.
Während SQL bei relationen Datenbanken Anwendung findet, verfolgen NoSQL-Datenbanken einen nicht-relationalen Ansatz. Im Folgenden wird ein kurzer Vergleich dieser angestellt, welcher in Tabelle 2 ersichtlich ist.

	SQL	**NoSQL**
Eigenschaften	Relationales Datenbankmodell	Nicht-relationales Datenbankmodell
	Vertikale Skalierung	Horizontale Skalierung
	ACID	CAP
	Erstellung des Schemas vor Eintragen von Daten	Erstellung des Schemas während der Laufzeit möglich
Vorteile	Fähigkeit zur Unterstützung von Transaktionen	Fähigkeit der Speicherung komplexer Datentypen
	Fähigkeit zur Unterstützung von Aggregation zwischen Daten	
		Ausbleiben signifikanter Änderungen im Quellcode bei Änderungen der Datenstruktur
Nachteile	Schwerfällig im Umgang mit großen Datenmengen	
		Fehlen eines standardisierten Models

Tabelle 2: Vergleich von SQL- und NoSQL-Datenbanken

Die Begriffe ACID (kurz für: **A**tomicity **C**onsistency **I**solation **D**urability) und CAP (kurz für: **C**onsistency **A**vailability **P**artition Tolerance) repräsentieren in diesem Zusammenhang Theorien zur Beschreibung der Eigenschaften von Datenbankensystemen. Dabei konzentriert sich das CAP-Theorem vor allem auf die Eigenschaften von verteilten Systemen.

(Vgl. Meier; Kaufmann, 2016)

[4]kurz für: **S**tructured **Q**uery **L**anguage
[5]kurz für: **N**ot **O**nly **SQL**

Web-Schnittstelle

Die Web-Schnittstelle stellt eine Form zur Eingabe neuer Daten und zur Ausgabe bereits vorhandener Daten zur Verfügung. Sie ermöglicht damit die Kommunikation der Geräte untereinander.
Die Schnittstelle muss folgende Funktionen erfüllen:

- Entgegennahme neuer Daten
- Überprüfung und Validierung neuer Daten
- Speicherung dieser in der Datenbank
- Abfrage von gespeicherten Datensätzen von der Datenbank
- Auslieferung der abgefragten Datensätze
- Bereitstellung von Filtermechanismen zur Spezifizierung der abzufragenden Daten
- Sicherung der Unabhängigkeit vom genutzten Datenbanksystem
- Authentifizierung von Nutzern bzw. Geräten
- Unterstützung eines Rechte- und Rollensystem sowie einer Nutzerverwaltung

4.2.4 Frontend

Dem User-Interface werden diverse Funktionen abverlangt. Die wichtigste dabei ist die Visualisierung der gespeicherten Daten als Diagramme bzw. Graphen.
Außerdem soll das User-Interface dem Nutzer die Möglichkeit bieten, ein Datum bzw. einen Zeitraum zu wählen, aus welchem die Daten dargestellt werden sollen. Zudem kann eine Übersicht der zuletzt gemessenen Werte gezeigt werden.
Da in einem realen Umfeld eine Vielzahl an Sensoren in dem System agieren, sollten auch diese vom Nutzer zur Visualisierung spezifiziert werden können. Nützlich wäre dies vor allem in Fällen, in denen nur ein spezielles Gebiet betrachtet werden soll. Neben der Wahl der Sensoren sollte auch eine Darstellung der Position des momentan gewählten Sensors bzw. Messstation einfließen.
Um die Nutzung der Software bzw. die Ausgabe der Daten einzuschränken, muss ein Rollen-und-Rechte-System entworfen werden. Die Verwaltung der Nutzer muss durch einen Systemadministrator vorgenommen werden. Zusätzliche Bedienelemente in der Nutzeroberfläche zur Unterstützung dieses Prozesses wären wünschenswert.
Die beschriebenen Funktionen sind in Abbildung 13 noch einmal dargestellt.

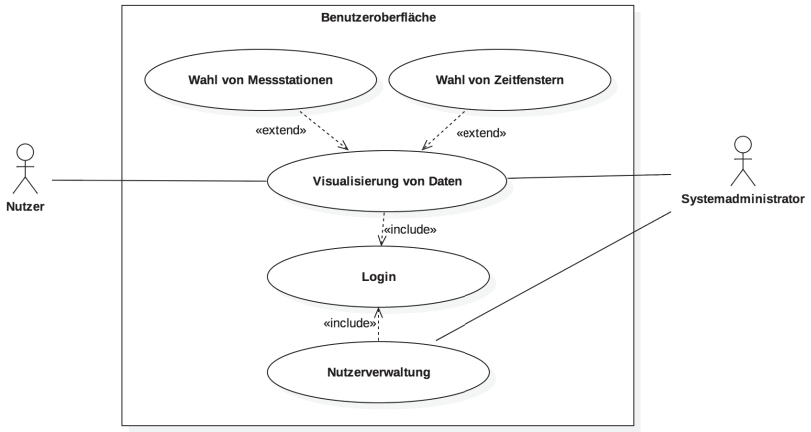


Abbildung 13: Anwendungsmöglichkeiten der Benutzeroberfläche

5 Prototypische Umsetzung des Entwurfs

In diesem Kapitel wird eine Umsetzung des im Vorherigen beschriebenen Entwurfs behandelt.

5.1 Genutzte Technologien

5.1.1 Sprache

Bisherige Systeme nutzen diverse Programmiersprachen. Um deren Vielfalt zu verringern, kommen diverse Sprachen in Betracht, die eine Full-Stack[6]-Entwicklung unterstützen. Dazu zählen unter anderem folgende:

- Java
- Go
- Python
- Swift (für Apple-Geräte)
- Ruby / Ruby on Rails
- JavaScript / Node.js

Für die Umsetzung dieses Projektes wurde JavaScript gewählt, welches mit Hilfe von *Node.js* auch außerhalb des Browsers betrieben werden kann. *Node.js* ist eine JavaScript-Laufzeitumgebung, die auf einer asynchronen Architektur basiert. Die Grundlage dafür bildet die V8-Engine[7].
Die wichtigsten Grundsätze der Laufzeitumgebung sind:

JavaScript Die Programmierung der Skripte findet ausschließlich in JavaScript statt. Für die Nutzung von Ressourcen der Betriebssysteme wurden konkrete APIs implementiert.

Non-blocking IO Alle Methoden der *Node.js*-Bibliothek sind nicht blockierend bzw. asynchron. Das heißt, dass diese bei Aufruf die Ausführung des Skripts nicht pausieren oder weitere Aktionen blockieren. Aufgaben, die nicht direkt bearbeitet werden müssen, werden an das Betriebssystem oder andere Applikationen übergeben. Nach Beendigung dieser Aufgabe erhält der *Node.js*-Prozess eine Rückmeldung und kann mit Hilfe von Callback-Funktionen die Informationen weiter verarbeiten.

Single-Threaded Ohne weitere Modifikationen läuft eine *Node.js*-Applikation in nur einem Prozess und bietet Parallelisierung nur durch den nicht-blockierenden Aufbau oder das Erzeugen von Prozessen mit Hilfe des child_process-Moduls.

[6] bezeichnet die gesamte Spanne der Software-Entwicklung, also Backend, Frontend und Qualitätssicherung inklusive Testing (Vgl. Gellert, 2012)

[7] V8 ist Google's hoch performante Open-Source-JavaScript-Engine. Diese ist in C++ geschrieben und kommt im Open-Source-Browser Google Chrome zum Einsatz. Sie kann eigenständig verwendet oder in C++-Applikationen eingebunden werden. (Vgl. Hablich, 2017)

Node.js stützt sich auf eine Sammlung verschiedener eigenständiger Bibliotheken. Obwohl diese unabhängig voneinander entwickelt und gepflegt werden, bilden sie den Kern von *Node.js*.
Dieser modulare Aufbau findet sich auch in dem Modulsystem von *Node.js* wieder. Für die Erweiterung des eigenen Skripts kann eine Vielzahl von Modulen heruntergeladen und eingesetzt werden.
NPM - der Paketmanager, welcher mit *Node.js* ausgeliefert wird - beinhaltet mittlerweile die größte Software-Registry der Welt. Diese umfasst derzeit ca. 475.000 Pakete bzw. Module sowie 7 Millionen Nutzer bzw. Entwickler und erzielt derzeit mehr als 3 Milliarden Modul-Downloads pro Woche. (*npm*, Stand: Januar 2018)
Diese Module können mit dem Befehl `npm install <Name des Moduls> -g` global installiert werden, sodass sie unter Linux-Systemen unter dem Pfad `/usr/local/lib` bzw. unter Windows-Systemen im Ordner `AppData` des jeweiligen Nutzers gespeichert werden.
Alternativ kann die Installation auch lokal, also nur für das betroffene Projekt, ausgeführt werden. Dazu genügt der Aufruf `npm install <Name des Moduls>` innerhalb des Projektordners. Dabei werden Module immer im Ordner *node_ modules* innerhalb des Projektordners gespeichert.
Programme können generell mit dem Befehl `node <Name des Skripts>.js` gestartet werden. Wird dem Aufruf von `node` keine Datei übergeben, so wird ein REPL[8] gestartet.
Für den Aufbau eines eigenen Moduls ist eine *package.json*-Datei notwendig. Diese beschreibt sowohl Eigenschaften, wie z.B. Name, Autor, optional Git-Repository und Lizenz des Moduls, als auch Abhängigkeiten des Programms, also welche Module in dem Programm eingesetzt werden.
Des Weiteren können in der *package.json*-Datei Skripte definiert werden, wie z.B. das Start-Skript. Ist dieses definiert, kann das Programm auch mit dem Befehl `npm start` gestartet werden.
Darüber hinaus können eigene Skripte definiert werden. Wenn der Quellcode beispielsweise vor dem Start des Programms kompiliert werden muss, kann ein *Build*-Skript definiert werden. Da dieses jedoch kein Standard-Skript von *npm* ist, muss der Aufruf mit einem vorangestellten *run* erfolgen, also `npm run build`.
Zur Verdeutlichung des Prinzips ist eine beispielhafte *package.json*-Datei in Codebeispiel 1 dargestellt. Zusätzlich eingeflossen ist bei diesem Beispiel das Feld *devDependencies*. Dieses wird genutzt, um Abhängigkeiten zu definieren, die für die Entwicklung der Software von Belang sind, wie in diesem Fall ein Modul für automatisierte Tests des Quellcodes.

[8] kurz für: Read-Eval-Print Loop, also ein interaktiver Interpreter

```
{
  "name": "rest_server",
  "description": "REST Server to handle new incoming data and
  ↪  outputting existing data",
  "version": "0.1.0",
  "dependencies": {
    "express": "4.16.2"
  },
  "devDependencies": {
    "mocha": "5.0.0"
  }
}
```

Codebeispiel 1: Beispielhafte *package.json*-Datei

Sind alle Abhängigkeiten des Moduls, wie in dem Beispiel gezeigt, in der *package.json* beschrieben, so reicht ein einfacher Aufruf von `npm install` im Projektordner, um diese lokal zu installieren.

(Vgl. Springer, 2016)

5.1.2 Schnittstelle

Für die Umsetzung einer web-basierten Schnittstelle kommen verschiedene Prinzipien in Frage. Im Rahmen dieses Projekts wird für die Umsetzung der Schnittstelle das *REST*-Prinzip genutzt.
Representational state transfer (kurz: *REST* oder auch *REST*ful Web-Service) ist eine Architektur, die beschreibt, wie Maschinen in Netzwerk-basierten Umfeldern miteinander kommunizieren können. Entwickelt und beschrieben wurde diese Architektur von Roy Fielding in seiner Dissertation im Jahr 2000. (Siehe auch: Fielding, 2000)
Sie ist nicht standardisiert, nutzt dabei allerdings standardisierte Verfahren, wie *HTTP*[9], *HTTPS*[9], *URI*[10] und *JSON*[11].

[9] kurz für: Hypertext Transfer Protocol (HTTP) (Clark, 2013a) bzw. Hypertext Transfer Protocol Secure (HTTPS) (Clark, 2013b)
[10] kurz für: Uniform Resource Identifier (Clark, 2013c)
[11] kurz für: JavaScript Object Notation (Pomaska, 2012)

Die Architektur legt keine Regeln für die spezielle Implementierung vor, wird aber durch die folgenden Prinzipien genauer beschrieben:

Eindeutige Identifikation von Ressourcen Nutzung der URI im Web zur eindeutigen Bestimmung von Ressourcen

Hypermedia / Verknüpfungen Verknüpfung von Ressourcen ähnlich dem Aufbau relationaler Datenbanken

Nutzung standardisierter Verfahren Verwendung der von HTTP bereit gestellten Methoden zum Zugriff auf Ressourcen

Präsentation der Ressourcen Durch Setzen des *Accept*-Headers kann der Client bestimmen, in welchem Format die erwartete Antwort ausgeliefert werden soll

Statuslose Kommunikation Verzicht auf das Erzeugen einer Sitzung zu Gunsten von Skalierbarkeit und Verringerung der Kopplung zwischen Client und Server

Beim Zugriff auf eine REST API über das HTTP-/HTTPS-Protokoll bestimmt die HTTP-Methode die Art des Zugriffs. Dazu gehören hauptsächlich die CRUD[12]-Operationen, welche im Folgenden erläutert sind:

GET Abfrage einer Ressource vom Server

POST Hinzufügen einer neuen Ressource

PUT Hinzufügen einer Ressource bzw. Überschreiben einer identischen vorhandenen Ressource

PATCH Ändern einer vorhandenen Ressource

DELETE Löschen einer vorhandenen Ressource

Die nachfolgenden Methoden sind weniger gebräuchlich. Sie liefern beispielsweise Metadaten oder bieten die Möglichkeit der Verschlüsselung des Zugriffs hinter einem Proxy.

HEAD Abfragen von Metadaten einer Ressource

OPTIONS Abfragen von verfügbaren Methoden einer Ressource

CONNECT Verbindung durch einen Proxy

TRACE Rückgabe der Kopie einer Anfrage, wie vom Server erhalten (dient der Ermittlung von Manipulationen während der Übermittlung)

(Vgl. Tilkov, 2011)

[12]kurz für: Create, Read, Update, Delete; dt: Erstellen, Lesen, Verändern, Löschen

5.2 Komponente: Sensorik

5.2.1 Physischer Aufbau

Der physische Aufbau für das Testsystem beinhaltet jeweils ein Arduino-kompatibles Entwicklungs-Board, speziell einen *Adafruit HUZZAH ESP8266*, an das ein oder mehr Sensoren angeschlossen werden. Grundsätzlich kann dieses Board jedoch mit jedem Gerät ausgetauscht werden, das eine physische Verbindung zu Sensoren ermöglicht und eine Netzwerkverbindung bietet. Ein entsprechendes Schaltdiagramm mit einem Adafruit ALS-PT19-Sensor zur Messung von Lichteinstrahlung ist in Abbildung 14 zu sehen.

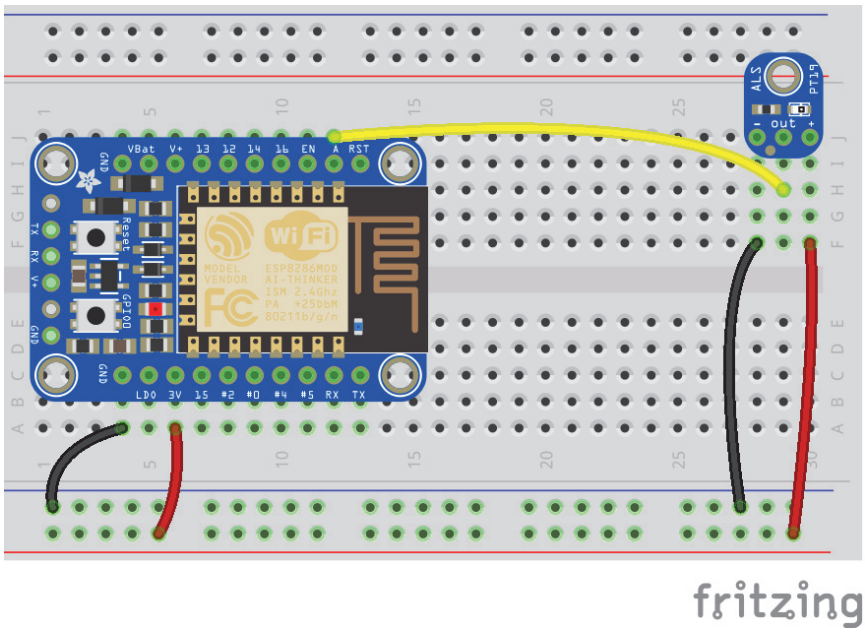


Abbildung 14: Verkabelung des ESP8266-Boards mit dem ALS-PT19-Sensor

Das ESP8266-Board ist durch das integrierte WiFi-Modul mit einem WLAN-Netz verbunden, welches von der Basisstation aufgespannt wird. Alternativ können auch Mikrocontroller genutzt werden, die durch ein Kabel mit dem Netzwerk verbunden werden können.

5.2.2 Software

Da die Mikrocontroller in Hinblick auf die Hardware stark eingeschränkt sind, ist der Einsatz einer JavaScript-Umgebung auf Kosten der Ressourcen nicht ratsam. Aus diesem Grund führen die Mikrocontroller ein C-Skript zur Implementierung des Firmata-Protokolls[13] aus. Hierbei handelt es sich um die einzige Ausnahme von der JavaScript-Nutzung, da es bereits durch die Arduino IDE[14] vorgegeben ist, sodass es zur Inbetriebnahme nur noch konfiguriert werden muss. Dazu gehören unter anderem Parameter, wie die SSID, das Passwort und die Art der Authentifizierung des zu verbindenden Netzwerkes für die Nutzung des Protokolls über eine WLAN-Verbindung. Der Quellcode der Programme ist unter `https://github.com/firmata/arduino/tree/master/examples` einzusehen.
Nachdem die Konfiguration vollständig abgeschlossen ist, muss das Programm kompiliert und auf das Gerät transferiert werden. In der Arduino IDE geschieht dieser Vorgang durch die Betätigung des Buttons *Hochladen*. Zuvor muss das jeweils verwendete Board jedoch installiert werden. Die entsprechenden Anleitungen dazu sind meist auf der Herstellerseite ersichtlich. Nachdem das Gerät daraufhin mit dem Programm gestartet ist, wartet es auf Befehle über die definierte Schnittstelle. Diese werden von der Basisstation abgesetzt.

[13] Firmata ist ein Protokoll zur Kommunikation mit Mikrocontrollern mittels Software auf einem Computer. Es basiert auf einem Midi-Nachrichten-Format. (Vgl. Hoefs, 2018)

[14] kurz für: Integrated Development Environment, dt.: integrierte Entwicklungsumgebung; Programm zur Erstellung von Software

5.3 Komponente: Basisstation

Die Basisstation bildet ein Raspberry Pi 3 Model B. Betrieben wird dieser mit dem Raspbian Jessie Lite Betriebssystem in der aktuellsten Version vom 05. Juli 2017.
Das Gerät wird in dem Testaufbau so konfiguriert, dass ein Access-Point gebildet wird, mit dessen Netzwerk sich die Mikrocontroller verbinden können. Dazu existiert bereits eine Schritt-für-Schritt-Anleitung auf der Website der Raspberry Pi Foundation. (Siehe auch: *Setting up a Raspberry Pi as an Access Point in a standalone Network*).
Für den Testaufbau fungiert der Raspberry Pi zunächst als *LAN-Bridge*. Das Gerät kommuniziert per WLAN mit den Mikrocontrollern und sendet über die LAN-Verbindung die Daten an das Backend.
Des Weiteren wird auf diesem Gerät *Node.js* installiert.
Für die Kommunikation mit den Mikrocontrollern wird das *Johnny-Five*-Framework eingesetzt. Durch das Framework werden Befehle bereit gestellt, welche auf Basis des *Firmata*-Protokolls zur Steuerung der Mikrocontroller dienen. Vorteil dieses Protokolls ist, dass es nicht von der Art der Verbindung abhängt. So können beispielsweise Geräte über das Netzwerk, Bluetooth oder eine USB angesteuert werden.
Zunächst müssen die Zugriffsmethoden auf die Mikrocontroller in dem Skript definiert werden. Das Minimalbeispiel dazu ist in Codebeispiel 2 dargestellt.

```
const five = require('johnny-five');
const board = new five.Board();
```

Codebeispiel 2: Grundlegende Einbindung eines Mikrocontrollers

Soll der Mikrocontroller über WLAN gesteuert werden, so kommen zwei weitere JavaScript-Module dazu: *Firmata* und *Etherport-Client*. Die Konfiguration ist in Codebeispiel 3 zu sehen.

```
const five = require('johnny-five'),
  firmata = require('firmata'),
  epc = require('etherport-client').EtherPortClient;

const board = new five.Board({
  io: new firmata(new epc({
    host: 'BOARD_IP',
    port: 3030
  })),
  timeout: 100000,
  repl: false,
  debug: true
});
```

Codebeispiel 3: Einbindung eines Mikrocontrollers über WLAN

Die Ordnerstruktur auf Seiten der Basisstation ist sehr simpel gehalten und zu Zwecken der Veranschaulichung in Abbildung 15 zu sehen.

```
Sensor
├── conf
│   ├── api.json
│   ├── boards.js
│   └── station.json
├── node_modules
├── index.js
└── package.json
```

Abbildung 15: Ordnerstruktur des Skripts der Basisstation

Die *index.js* bindet zunächst die *boards.js*-Datei ein, welche die Konfiguration der Mikrocontroller enthält. Anschließend wird die Verbindung zu diesen hergestellt. Nachdem der Vorgang beendet ist, stehen die Mikrocontroller für Befehle zur Verfügung und das Programm läuft in einer Schleife zur Abfrage der Werte in einem regelmäßigen Interval. Nach der Abfrage der Werte werden diese mit Hilfe des Moduls *request* per POST-Request an die Web-Schnittstelle übermittelt. Ein entsprechender Programmablauf des Skripts ist in Abbildung 16 dargestellt.

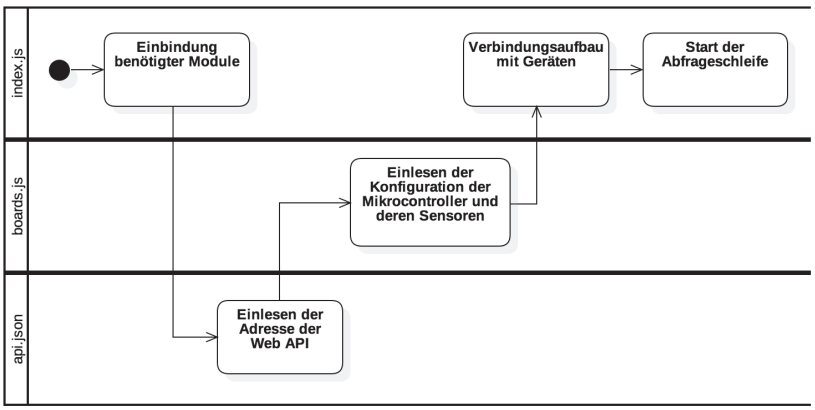


Abbildung 16: Programmablauf des Skripts der Basisstation

Das Skript für einen solchen Request ist in Codebeispiel 4 dargestellt. An Stelle des direkten Aufrufs der Adresse der Web-Schnittstelle können auch die in der *api.json* beschriebenen Werte genutzt werden.

```
request.post({
    url: 'http://WEB_API_IP:8000/post',
    form: messDaten
}, (err, res, body) => {
    console.log(res);
});
```

Codebeispiel 4: Beispielhafter POST-Request mit Hilfe des *request*-Moduls

5.4 Komponente: Backend

5.4.1 Physischer Aufbau

Das Backend wird in der Testumgebung auf dem Entwicklungsrechner betrieben. Das heißt, dass sowohl Web-Schnittstelle als auch Datenbank auf diesem Gerät laufen. In Hinblick auf die Entwürfe des physischen Aufbaus des Backends entspräche dies dem Aufbau mit einem Server.

5.4.2 Datenbank

Modell

Aufgrund der zentralen Verwaltung der Daten auf den Servern des GFZ wird in diesem Projekt eine SQL-Datenbank eingesetzt. Das relationale Datenmodell ist in Abbildung 17 dargestellt.

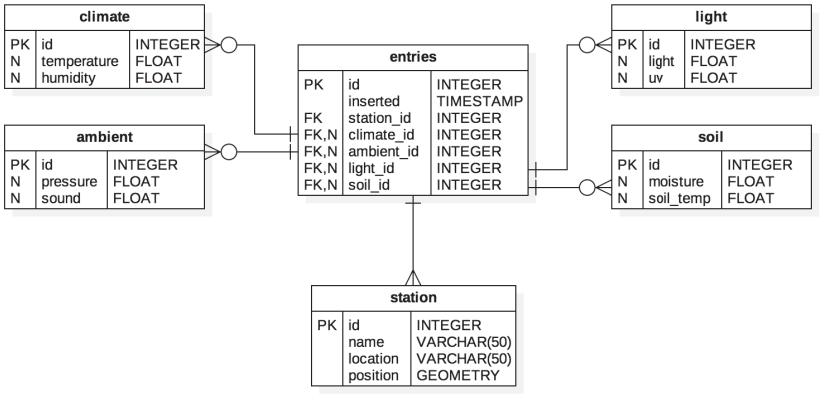


Abbildung 17: Datenbankmodell

Darin enthalten sind Daten über die Messstation, die Zeit des Eintrages sowie die gemessenen Umweltdaten. Einträge über Parameter der Orientierung des Gerätes und Bodenfeuchte und -temperatur sind optional (*nullable*). Nicht berücksichtigt wurde sowohl im Modell als auch im weiteren Verlauf das Rechten- und Rollen-System.
Nach der Erstellung des Datenmodells kann dieses mit Hilfe der Software *DBWrench* auf dem Server als Datenbank erstellt werden.

5.4.3 Web-Schnittstelle

Express

Die Web-Schnittstelle wird mit Hilfe des *Express*-Moduls umgesetzt. Zu den Haupt-Features des Moduls gehören unter anderem die folgenden:

- Robustes Routing
- Hohe Performance
- Umfangreicher Test-Umfang
- Unterstützung von HTTP-Helper-Funktionen

(Siehe auch: Wilson, 2017)

Das übliche *Hello World* lässt sich hier mit dem in Codebeispiel 5 gezeigten Quellcode umsetzen. Wird der Server gestartet, stellt er die beschriebenen Routen zur Verfügung.
Eine Route ist eine URL. Den Aufruf dieser URL kann der Server mit der Ausführung einer Funktion und der Rückgabe der entsprechenden Ergebnisse beantworten, z.B. die Abfrage einer Datenbank. Die auszuführende Funktion wird zudem über die HTTP-Methode bestimmt, mit der die Route aufgerufen wird.
In diesem Beispiel wird nur eine Route (`/`) genutzt. Ein Aufruf der URL `http://localhost:3000/` im Browser erzeugt die in Abbildung 18 dargestellte Antwort.

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

app.get('/', (req, res) => {
    res.end('Hello World!');
});

app.listen(3000, () => {
    console.log('Server listening on port 3000!');
});
```

Codebeispiel 5: Umsetzung eines einfachen Servers mit dem *Express*-Framework

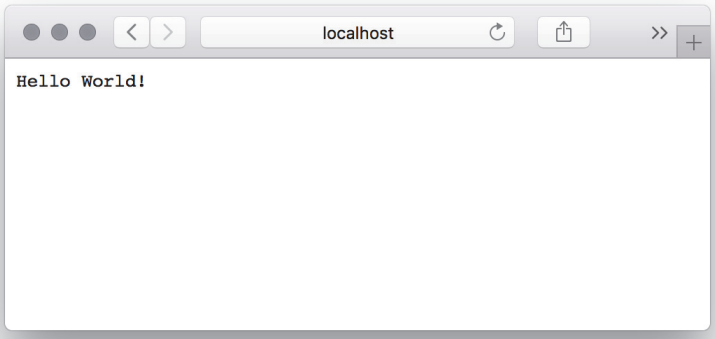


Abbildung 18: Antwort des Servers

Ein Vorteil der Nutzung von *Express* bzw. des REST-Prinzips ist, dass Routen, die nicht spezifiziert sind, blockiert werden. Dies umfasst sowohl die URL als auch die Methode des Aufrufs. Somit müssen sich Entwickler solcher Anwendungen vorerst keine Gedanken um den Schutz derartiger Routen machen. In Abbildung 19 ist die Antwort des Servers zu sehen, die entsteht, wenn eine nicht zulässige Route aufgerufen wird. Es ist jedoch üblich, solche Aufrufe abzufangen und mit dem Fehlercode 404, d.h. Ressource nicht gefunden, zu reagieren.

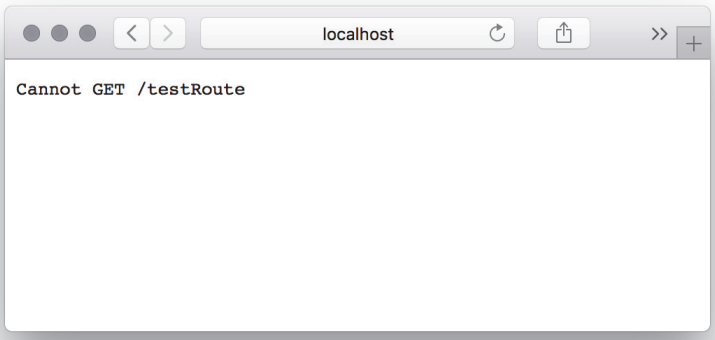


Abbildung 19: Antwort des Servers bei Aufruf einer nicht existierenden Route

Routen

Die REST API stellt die Verbindung zwischen der Anwendung und der Datenbank dar. (Siehe auch Kapitel 5.1.2) Sie nimmt Anfragen, wie z.B. lesende Zugriffe des Frontends und schreibende Zugriffe der Sensorstationen, entgegen und verarbeitet diese in Zusammenarbeit mit der Datenbank. Die vorhandenen Routen bzw. URLs für den lesenden Zugriff sind die folgenden:

- /get/all
- /get/climate
- /get/ambient
- /get/light
- /get/all/:date
- /get/climate/:date
- /get/ambient/:date
- /get/light/:date

Doppelpunkte symbolisieren dabei Parameter bzw. Variablen, die durch die konkrete URL spezifiert werden. So können beispielsweise die Daten des **01.01.2018** mit der URL `/get/all/01.01.2018` abgerufen werden.
Weitere Routen könnten die Auswahl spezifischer Sensorenstationen beinhalten, sind jedoch im weiteren Verlauf nicht umgesetzt. Dazu bestünde die Möglichkeit, die bereits erwähnten Routen um die ID oder den Namen einer Sensorstation zu erweitern, wie z.B.:

- /get/:id/all
- /get/:id/all/:date
- usw.

Für schreibende Zugriffe genügt grundlegend eine Route, welche mit der Methode *POST* aufgerufen werden muss.

- /post/

An dieser Stelle wurde eine Middleware implementiert, die die per POST-Methode an diese URL übermittelten Daten analysiert und weiterverarbeitet. Sie enthält Methoden, wie z.B. die Kontrolle der enthaltenen Parameter, die Überprüfung von Daten über die Messstation und der Validierung der erhaltenen Daten.

Kommunikation mit der Datenbank

Für die Anbindung an die Datenbank wird ein weiteres Modul aus der *npm*-Bibliothek genutzt: *pg*. Dieses dient als PostgreSQL-Client. (Siehe auch: Carlson, 2018)

Die Verbindung einer Datenbank erfolgt wie in Codebeispiel 8 gezeigt. Die Verbindungsparameter können dabei sowohl als JavaScript-Objekt als auch als Verbindungs-URI übergeben werden, wie die Codebeispiele 6 und 7 zeigen.

```
const connection = {
    user: 'dbuser',
    host: 'dbhost',
    database: 'dbname',
    password: 'dbpw',
    port: 5432
}
```

Codebeispiel 6: Spezifikation der Datenbankverbindung mittels Objekt

```
const connection = {
    connectionString:
    ↪  'postgresql://dbuser:dbpw@dbhost:5432/dbname'
}
```

Codebeispiel 7: Spezifikation der Datenbankverbindung mittels URI

```
const { Client } = require('pg');

const client = new Client(connection);

client.connect();
```

Codebeispiel 8: Aufbau einer Datenbankverbindung mit dem *pg*-Modul und der Codebeispiel 6 bzw. 7 bestimmten Verbindungsvariablen

Nach erfolgreichem Aufbau der Verbindung können Abfragen bzw. Queries an die Datenbank gestellt werden. In Promise-Schreibweise[15], welche seit ECMAScript[16] Version 6 in JavaScript unterstützt wird, werden solche Queries, wie in Codebeispiel 9 zu sehen, formuliert.

[15] Promises werden für asynchrone Berechnungen genutzt und verbessern die Übersichtlichkeit des Quellcodes. (Siehe auch: Lindström, 2017) Sie können sich dabei in einem von drei Zuständen befinden: pending (ausstehend), fulfilled (erfolgreich) oder rejected (gescheitert).

[16] Standardisierte Spezifikation von JavaScript (Siehe auch: ECMA International, 2017)

```
client.query('SELECT * from <Name der Tabelle>')
    .then(result => {
        console.log(result.rows);
    })
    .catch(error => {
        console.error(error);
    });
```

Codebeispiel 9: Beispielhafte Abfrage aller Werte und Ausgabe dieser bzw. auftretender Fehler auf der Konsole

Nachdem die Kommunikation mit der Datenbank beendet ist, wie z.B. beim Herunterfahren oder Neustarten des Servers, kann diese mit dem Befehl `client.end()` sicher geschlossen werden.
(Vgl. *node-postgres*)

Sicherheit

Sicherheitsmaßnahmen, wie die Authorisierung von Zugriffen oder die Einrichtung von HTTPS, werden in diesem Umfeld nicht weiter betrachtet. Zum Schutz der API wird vorerst nur das Modul *Helmet* eingesetzt. Dieses Modul ist eine Sammlung von 12 kleineren Middleware-Funktionen und setzt mit diesen die sicherheitsrelevanten HTTP-Header.
In die Express-Umgebung wird *Helmet*, wie in Codebeispiel 10 dargestellt, eingebunden.

```
const express = require('express');
const helmet = require('helmet');

const app = express();

app.use(helmet());
```

Codebeispiel 10: Aktivierung der Funktion von *Helmet*

Standardmäßig werden dabei die folgenden Header gesetzt:

- X-DNS-Prefetch-Control
- X-Frame-Options
- X-Powered-By
- Strict-Transport-Security
- X-Download-Options
- X-Content-Type-Options
- X-XSS-Protection

Allerdings kann dieses Verhalten auch konfiguriert werden, so können noch weitere Header aktiviert oder die standardmäßig aktivierten Header deaktiviert werden. Bei Nutzung weiterer Header müssen diese zuvor installiert werden. Die Header und die dazugehörigen Module können der Dokumentation entnommen werden.
Die Nutzung dieser Konfigurationsmöglichkeiten ist in Codebeispiel 11 gezeigt.

```
app.use(helmet({
    frameguard: false,
    noCache: true
}));
```

Codebeispiel 11: Konfigurationsmöglichkeiten von *Helmet*

(Vgl. Hahn, 2018)

Erreichbarkeit des Servers

Um eine konstante Erreichbarkeit des Servers zu erzielen, kann das Skript mit Hilfe eines Moduls zur Beobachtung des Status ausgeführt werden. Dieses Modul überwacht sowohl Änderungen im Quellcode des Skripts als auch Abstürze während der Laufzeit. Bei Änderungen oder nach einem Absturz wird das Skript somit unmittelbar neu gestartet und die Ausfallzeit des Servers minimiert.
Um Fehler während der Laufzeit und somit auch Absturzursachen zu ermitteln, kann solch ein Modul mit einem Logger kombiniert werden. In der weiteren Betrachtung dieses Projekts wird dieser aber nicht berücksichtigt.
Das in diesem Projekt genutzte Modul zur Überwachung des Servers heißt *supervisor*.
Um es einzusetzen, genügt der Aufruf `supervisor <Name des Skripts>.js` bzw. die Definition dessen in der *package.json* als *Start*-Skript und der Start per `npm start`.

(Vgl. Isfan, 2017)

Tests

Um die Funktionsfähigkeit der Web API zu testen, können automatisierte Tests entwickelt werden. Diese starten den Server, überprüfen die Funktionalitäten der einzelnen Endpunkte bzw. Routen und liefern ein visuelles Ergebnis über den Ausgang der Tests.
Für die Umsetzung dieser wird das Modul *Mocha* in Kombination mit dem

Modul *Chai* genutzt.
Mocha ist ein Testing-Framework, das die grundlegenden Funktionalitäten zur Verfügung stellt. Zur Einbindung in die Projektumgebung kann ein *Test*-Skript in der *package.json* angelegt werden, welches den Befehl `mocha` aufruft. (Vgl. mochajs.org, 2018)
Chai ist eine sogenannte *Assertion*[17]-Bibliothek, die diverse Methoden zur Betrachtung bzw. zum Vergleich von Variablen in den Testumgebungen zur Verfügung stellt. (Vgl. Chai.js, 2017)
Ein einfaches Beispiel für die Kontrolle einer Berechnung ist in Codebeispiel 12 dargestellt und liefert das in Abbildung 20 gezeigte Ergebnis.

```
const { expect } = require('chai');

describe('1 + 1', () => {
    it('should return 2', () => {
        expect(1+1).to.be.equal(2);
    });
});
```

Codebeispiel 12: Test zur Betrachtung des Ergebnisses einer Berechnung

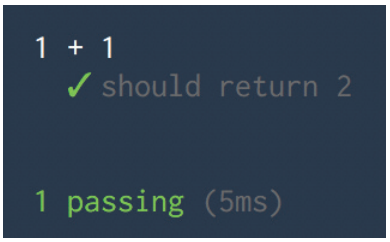


Abbildung 20: Darstellung des Testergebnisses auf Kommandozeilenebene

Für die Schnittstelle wurden Tests implementiert, die die vorliegenden Routen auf ihre Funktionsfähigkeit überprüfen. Das Ergebnis dieser Tests ist im Kapitel 6.3 zu sehen.

[17] deutsch: Behauptung

Struktur des Web-Servers

Um die Software möglichst unabhängig von konkreten Modulen zu gestalten, wurden die Teile jeweils in eigene Skripte ausgelagert. Diese Unterteilung ist in Abbildung 21 in einem Komponentendiagramm dargestellt.

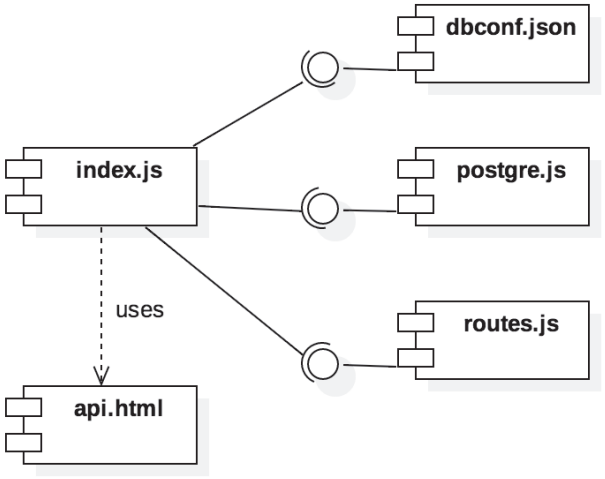


Abbildung 21: Unterteilung des Web-Servers in Komponenten

Um diese Aufteilung in Komponenten zu verdeutlichen, wurde eine klar definierte Ordnerstruktur genutzt. Der Ordner *controller* enthält Steuerelemente, in diesem Fall die Verbindung zur Datenbank. Darin sind des Weiteren alle Methoden beschrieben, die für die Kommunikation und den Datenaustausch mit der Datenbank benötigt werden, wie z.B. Methoden zum Abfragen bestehender und zum Speichern neuer Datensätze. Der Verbindungsaufbau geschieht über die im Ordner *conf* definierte *dbconf.json*. Diese enthält alle Parameter, wie z.B. IP-Adresse, Port, Login und Passwort, die zur Verbindung benötigt werden.

Die *routes.js* beinhaltet alle Routen, die der Server zur Verfügung stellen soll. Darin ist außerdem definiert, wie die Routen auf Aufrufe reagieren, also z.B. durch die Abfrage der Datenbank und die Auslieferung der Daten.
Die *index.js* bildet die Kernkomponente des Servers, welche sowohl alle nötigen Module, wie Express und Helmet einbindet als auch die Objekte der anderen lokalen Dateien zusammenführt. Sie ist auch für den endgültigen Start des Servers verantwortlich.
Die dementsprechende Ordnerstruktur ist in Abbildung 22 vorgestellt.

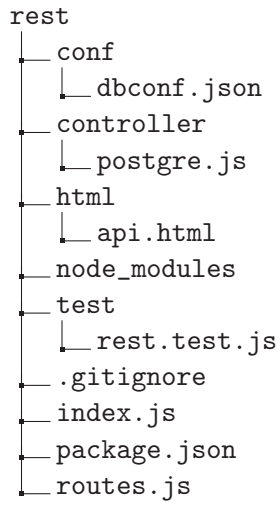


Abbildung 22: Ordnerstruktur des Web-Servers

Programmablauf des Web-Servers

Der Programmablauf beim Start des Web-Servers ist in Abbildung 23 visualisiert.
Anschließende Requests durch Clients, ob zur Abfrage oder Speicherung von Daten, erfolgen wie in Abbildung 5 bzw. 7 gezeigt. Greift ein Client auf eine nicht vorhandene Route zu, so wird er auf die *Home*-Route, also / weitergeleitet, auf der die *api.html* dargestellt wird. Diese liefert einen Überblick über alle verfügbaren Routen.

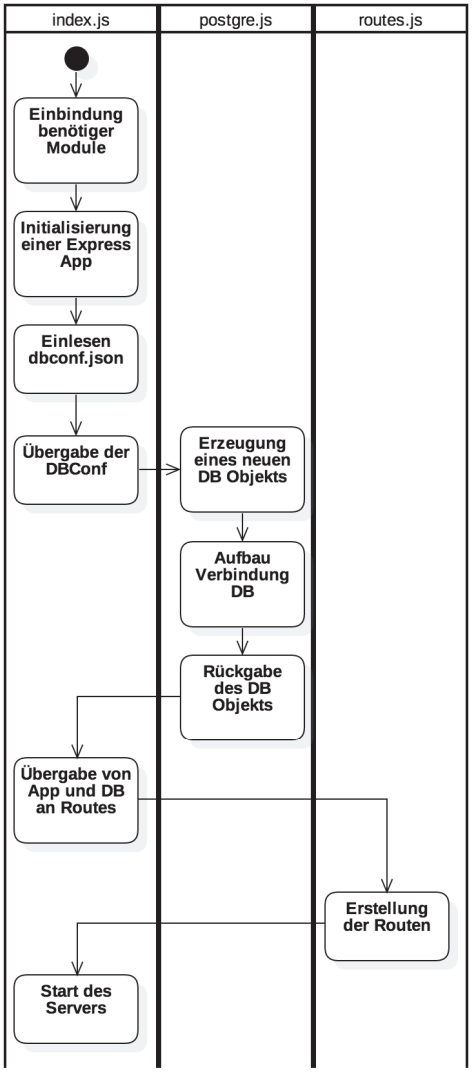


Abbildung 23: Programmablauf beim Start des Web-Servers

5.5 Komponente: Frontend

5.5.1 Electron

Für die Umsetzung des Frontends wurde das Modul *Electron* eingesetzt. Dieses von *GitHub* entwickelte Framework ermöglicht die Erstellung von cross-platform-fähigen Desktop-Applikationen auf Basis herkömmlicher Webtechnologien. Da *Electron* für das Rendern der Benutzeroberfläche *Chromium*[18] nutzt, ist es des Weiteren möglich, das Interface einer *Electron*-App mit geringem Aufwand für eine Webseite zu portieren, da es mit HTML, JavaScript, CSS, usw. erstellt wird. (Vgl. GitHub, Inc., 2018)
Ein Minimalbeispiel für eine *Electron*-Applikation ist in Codebeispiel 13 dargestellt.

```
const {app, BrowserWindow} = require('electron');
const path = require('path');
const url = require('url');

let win;

function createWindow() {
    win = new BrowserWindow({width: 800, height: 600});

    win.loadURL(url.format({
        pathname: path.join(__dirname, 'index.html'),
        protocol: 'file:',
        slashes: true
    }));

    win.on('closed', () => {
        win = null;
    });
}

app.on('ready', createWindow);
```

Codebeispiel 13: Minimalbeispiel für eine *Electron*-Applikation

Beim Start der Applikation wird eine *Node.js*-Instanz erzeugt. Diese wird auch *Main*-Prozess genannt. Anschließend wird von diesem Prozess ein *Chromium*-Browser-Fenster geöffnet, in dem die spezifizierte HTML-Datei geren-

[18] Chromium ist ein Open-Source-Browser-Projekt mit dem Ziel Nutzern einen sichereren, schnelleren und stabileren Umgang mit dem Internet zu bieten. (Vgl. Google LLC, 2018)

dert wird. Dies ist der so genannte *Renderer*-Prozess. Dieser Ablauf ist in Abbildung 24 visualisiert.

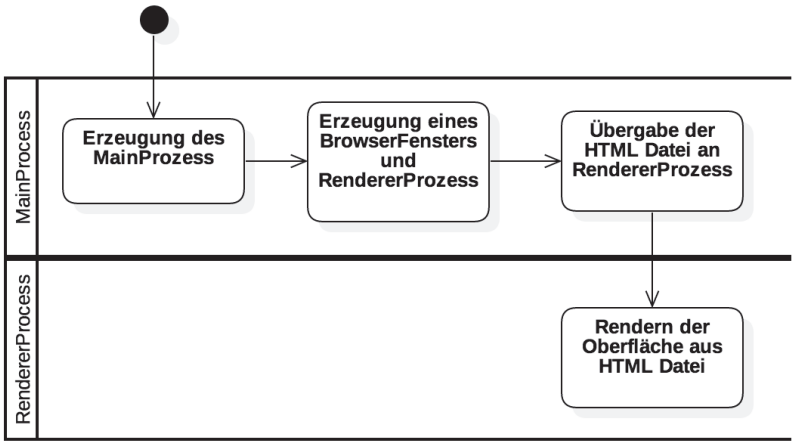


Abbildung 24: Start der Applikation

Der *Main*-Prozess einer *Electron*-Applikation ist einzigartig. Er ist für die Erstellung von *Renderer*-Prozessen, die Kommunikation mit dem Betriebssystem und das Starten und Beenden der Applikation zuständig.
Dem gegenüber steht der *Renderer*-Prozess, welcher die Darstellung der ihm übergebenen HTML-Dateien übernimmt. Da für jedes geöffnete Fenster der Applikation ein eigener *Renderer*-Prozess gestartet wird, können anders als beim *Main*-Prozess mehrere dieser Prozesse existieren.
Die Kommunikation zwischen diesen Prozessen erfolgt durch die Methoden der IPC[19]-Module, die event-basiert auf die Übermittlung von Nachrichten des jeweils anderen Prozesses reagieren.
Bei Start des Minimalbeispiels und der Übergabe einer HTML-Datei mit einem *Hello World*-Beispiel, welches in Codebeispiel 14 dargestellt ist, erscheint das in Abbildung 25 gezeigte Ergebnis.

[19]kurz für: Inter-process communication, also Prozess übergreifende Kommunikation

```
<!DOCTYPE html>
<html>
<head>
    <title>Electron Applikation</title>
</head>
<body>
    <h1>Hello World!</h1>
</body>
</html>
```

Codebeispiel 14: *Hello World*-Beispiel in HTML

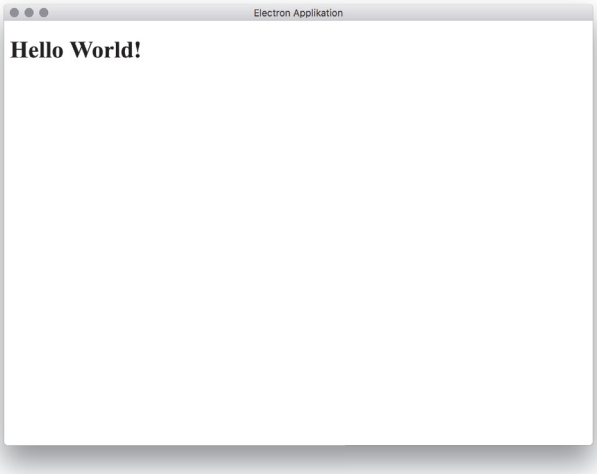


Abbildung 25: Geöffnete Applikation des Minimalbeispiels

Kommunikation mit der Web-Schnittstelle

Die Kommunikation mit der Web-Schnittstelle geschieht auf Seiten des *Main*-Prozesses, um IP-Adressen oder ähnliche Konfigurationen nicht in den Quellcode des UI[20] einfliessen zu lassen.
Es wird zunächst im *Renderer*-Prozess ein Event ausgelöst, dass einen entsprechenden Event-Listener auf Seiten des *Main*-Prozesses benötigt. Um solche Events zu spezifizieren, werden sie mit Hilfe von Strings benannt. Optional können Daten übertragen werden.

[20] kurz für: User Interface

Das Auslösen eines Events ist beispielhaft in Codebeispiel 15 dargestellt.

```
const { ipcRenderer } = window.require("electron");

ipcRenderer.send("data-request");
```

Codebeispiel 15: Auslösen eines Events im *Renderer*-Prozess

Der Event-Listener des *Main*-Prozesses, welcher mit Hilfe der Methode *on* registriert werden kann, ist beispielhaft in Codebeispiel 16 dargestellt. Dieser besorgt die von der Web-Schnittstelle benötigten Daten mit Hilfe eines GET-Requests.

```
const { ipcMain } = require("electron");

ipcMain.on("data-request", (event, ...args) => {
    // get data
    request.get({
        url: 'http://WEB_API_IP:8000/get/all',
    }, (err, res, body) => {
        // send data to renderer process
        event.sender.send("data-response", body);
    });
})
```

Codebeispiel 16: Event-Listener des *Main*-Prozesses

Über die Methode *send* des *sender*-Objekts, welches mit dem Event übergeben wird, kann eine Antwort an den anfragenden Prozess gesendet werden, welche die Daten beinhaltet. Da es sich bei der Antwort auch um ein Event handelt, muss dafür ein Event-Listener auf Seiten des *Renderer*-Prozesses implementiert sein. Dieser wird erneut durch einen String spezifiziert und erhält als zweites Argument eine Funktion, in der die Daten zur weiteren Verarbeitung genutzt werden können. Ein solcher Event-Listener auf Seiten des *Renderer*-Prozesses ist beispielhaft in Codebeispiel 17 abgebildet.

```
ipcRenderer.on("data-response", dataProcessingFunction);
```

Codebeispiel 17: Registierung einer Funktion als Event-Listener des *Renderer*-Prozesses

Der Ablauf der Kommunikation ist in Abbildung 26 visualisiert.

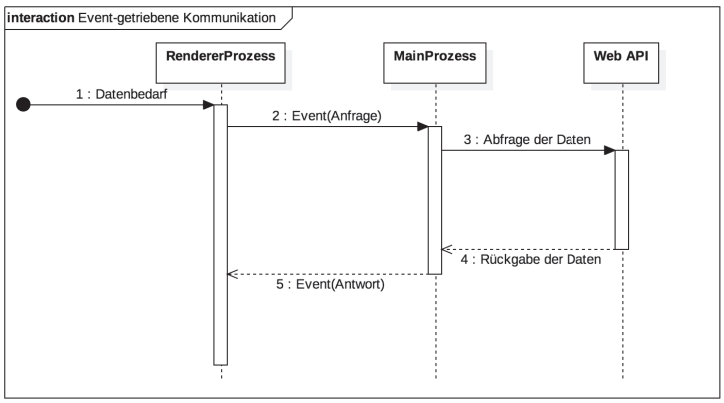


Abbildung 26: Kommunikation mittels IPC

5.5.2 React

Die Nutzeroberfläche wurde mit der JavaScript-Bibliothek *React* erstellt. Diese ermöglicht die Erstellung von Komponenten und die Einbindung dieser in einer HTML-ähnlichen Syntax namens *JSX*[21]. Vorteilhaft dabei ist die Wiederverwendbarkeit der Komponenten sowie die Möglichkeit, die Komponenten über Attribute bzw. so genannte *Properties* genauer zu definieren.
Properties werden ebenso übergeben, wie Attribute in HTML, wie beispielsweise der Typ in diesem Beispiel: `<input type="text" />`. Diese Properties können innerhalb der Komponente genutzt werden, um diese auf verschiedene Arten zu rendern.
Des Weiteren ist es möglich, dass die Komponenten unabhängig voneinander bzw. autark agieren. So kann beispielsweise eine Komponente, die Messwerte in Form eines Diagramms visualisiert, diese Werte auch selbstständig von einer API abfragen. Die Daten werden dann in der Komponente bzw. genauer im *State* der Komponente zwischengespeichert. *React* achtet dabei auf signifikante Änderungen des *State* und rendert bei Bedarf die betroffene Komponente neu. Die Signifikanz bzw das. Maß an Toleranz, welches für ein erneutes Rendern erreicht sein muss, kann im Quellcode definiert werden. Im Falle eines Verzichts auf diese Bestimmungen führt jegliche Änderung des *State* zu einem neuen Rendern. Dadurch können dynamische Seiten erstellt werden, die kein Erneuern der gesamten Seite benötigen. Diese Komponenten werden *Stateful* genannt.

[21] kurz für: JavaScript Syntax Extension

Neben den dynamischen Komponenten können auch statische Komponenten erstellt werden. Dazu zählen unter anderem Elemente, wie z.B. eine Navigationsleiste oder Logos. Diese sind grundlegend immer gleich aufgebaut ändern sich nicht oder nur geringfügig und werden nur über die *Properties* bestimmt. Diese Komponenten nennt man *Stateless*.
Eine solche Komponente ist beispielhaft in Codebeispiel 18 und die Einbindung dieser in ein HTML-Dokument mit einem Element mit der ID „root“ in Codebeispiel 19 dargestellt.

```
import React from 'react';

class BeispielKomponente extends React.Component {
    render () {
        return <h1>Hello, {this.props.name}!</h1>;
    }
}
```

Codebeispiel 18: Beispiel für eine *stateless* Komponente mit der *Property name*

```
import React from 'react';
import ReactDOM from 'react-dom';

import BeispielKomponente from './BeispielKomponente';

ReactDOM.render(<BeispielKomponente />,
↪  document.getElementById('root'));
```

Codebeispiel 19: Einbindung einer *React*-Applikation in ein HTML-Dokument

Das HTML-Dokument, welches das Element mit der ID „root“ bereitstellt, ist in Codebeispiel 20 aufgeführt. In dieses Dokument wird die *bundle.js*-Datei eingebunden. Diese ist Produkt des Kompilier-Prozesses, welcher im Kapitel 5.5.3 näher beschrieben wird.

```
<!DOCTYPE html>
<html>
<body>
    <div id='root'></div>

    <script src='./built/bundle.js'></script>
</body>
</html>
```

Codebeispiel 20: HTML-Dokument zur Einbindung der *React*-Applikation

Lebenszyklus von Komponenten

Komponenten besitzen einen Lebenszyklus, welcher durch Methoden beschrieben wird. Er beschreibt sowohl den Vorgang beim Einbinden als auch beim Updaten[22] und beim Entfernen einer Komponente.
Die für dieses Projekt erstellten Komponenten stützen sich hauptsächlich auf den Zyklus während der Einbindung einer Komponente. Dieser beinhaltet den Aufruf von vier Methoden in der folgenden Reihenfolge:

constructor()
Initiale Definition der Eigenschaften des *State* der Komponente sowie in diesem Fall Registrierung der Event-Listener

componentWillMount()
Nicht belegt, da meist nur für server-seitiges Rendern genutzt

render()
Einzige notwendige Methode, dient zur Erstellung der UI-Elemente

componentDidMount()
Methode zur Datenbeschaffung, in diesem Fall zum Senden von Events an *Main*-Prozess

[22] Updates werden beim Erhalt neuer *Properties* bzw. bei Änderungen des *State* veranlasst.

Zur Beschränkung der Bedingungen des Neuladens einer Komponente können die Methoden des Update-Zyklus genutzt werden:

- componentWillReceiveProps()
- shouldComponentUpdate()
- componentWillUpdate()
- render()
- componentDidUpdate()

Für die Kommunikation mit dem *Main*-Prozess werden Event-Listener eingesetzt. Nachdem eine Komponente nicht mehr benötigt bzw. aus der Oberfläche entfernt wurde, müssen diese abgemeldet werden. Diese Operation kann in der Methode `componentWillUnmount()` vorgenommen werden.

(Vgl. Facebook Inc., 2018)

Implementierte Komponenten

Für die Umsetzung der Bedienelemente wurde *React Desktop* genutzt. (Siehe auch: Bull, 2018) Es handelt sich dabei um eine Bibliothek bestehend aus Interface-Elementen, deren Optik an Windows 10 bzw. an Mac OS angepasst ist.
Die Komponenten dieser Bibliothek wurden beispielsweise für Text und Navigationsmenüs genutzt.
Zur Anpassung an das jeweils genutzte Betriebssystem wurde eine Zwischenschicht implementiert. Sie fragt zunächst über die API des *Main*-Prozesses das Betriebssystem ab und liefert anschließend die entsprechenden Komponenten aus.
Für die Visualisierung der Position der Sensoren wird eine Karte genutzt. Ihr liegt die *Google Maps JavaScript API* zu Grunde. Für die Einbindung der API durch *React*-Komponenten wurde die Bibliothek *react-google-maps* genutzt. (Siehe auch: Chen, 2018)
Zur Veranschaulichung der Daten in Diagrammen wurde die Bibliothek *Recharts* eingesetzt. (Siehe auch: Recharts Group, 2018)

5.5.3 Bundler

Da der Browser bzw. das in *Electron* genutzte *Chromium* die Syntax von *JSX* nicht unterstützt, muss der Quellcode vorher in ein verständliches Format transformiert werden.
Grundsätzlich kann für diese Aufgabe *Babel* genutzt werden. Dies ist ein weiteres Modul aus der *npm*-Bibliothek, das die Umwandlung von Quellcode ermöglicht. Hauptgedanke hinter diesem Modul ist die Nutzung von Features im Quellcode, die bereits durch neuere Spezifikationen des ECMAScripts in JavaScript integriert wurden, allerdings noch nicht von den Browsern unterstützt werden. Dazu werden die im Quellcode genutzten Features neuerer JavaScript-Versionen während des Kompilierens in eine für die Browser verständliche Form umgewandelt.
Ebenso verhält es sich mit JSX und React. Der Quellcode wird dabei in reines JavaScript umgewandelt.

(Vgl. Pick, 2015)

Allerdings wird *Babel* noch mit einem weiteren kombiniert: *Webpack*. *Webpack* ermöglicht es, den gesamten Quellcode in einer Datei zu bündeln. Dabei können durch diverse Erweiterungen und vorgefertige Einstellungen (*presets*), wie z.B. *Babel*, während des Prozesses Modifikationen am Quellcode vorgenommen werden.
Weiterer Vorteil, der durch den Einsatz eines solchen *Bundlers* generiert wird, ist ein so genannter Entwicklungs-Server (*dev-server*). Das bedeutet, dass während der Entwicklung ein Server bereitgestellt wird, der auf Änderungen im Quellcode achtet und das Interface gegebenenfalls neu lädt. Dies ermöglicht einen kontinuierlichen Workflow bei der Erstellung des Interfaces. Nachdem die Entwicklung abgeschlossen ist, kann der Quellcode auf normalem Wege kompiliert werden, wodurch der daraus generierte Quellcode in Dateien abgelegt wird und vom System genutzt werden kann.

(Vgl. Hartmann, 2015)

5.5.4 Aufbau

Auch das Frontend folgt einer definierten Ordnerstruktur, die vor allem der Übersichtlichkeit und der Aufteilung in Komponenten dient, jedoch auch durch die Unterteilung in *Main-* und *Renderer*-Prozess zu Stande kommt. Diese Struktur ergibt sich wie folgt:

scripts beinhaltet alle Skripte, die für Aktionen auf Seiten des *Main-Process* von Electron notwendig sind. Dazu zählen unter anderem Event-Listener für die Kommunikation mittels *ipcMain*.

src beinhaltet jeglichen Quellcode des Frontends, der einer Kompilierung bedarf, bevor er genutzt werden kann. Dazu zählen alle Elemente, die für die Bildung der Benutzeroberfläche bzw. vom Renderer-Prozess benötigt werden, also alle Komponenten (Ordner *components*), deren Styling (Ordner *sass* sowie individuelles Styling in den Ordnern der Komponenten) und die Event-Listener für die Kommunikation mittels *ipcRenderer* (Ordner *utils*).

test beinhaltet alle automatisierten Tests des Systems.

Während des Kompilierprozesses wird der Ordner *build* erzeugt. Darin wird die *bundle.js*-Datei gespeichert. Diese Datei enthält sämtlichen kompilierten Quellcode der Oberfläche.
In der Wurzel des Projektordners befinden sich Dateien, die hauptsächlich der Konfiguration der Applikation dienen:

.babelrc Definiert die Kompilier-Regeln für Babel

index.html Ausgangs-HTML-Datei, die die React-Applikation einbindet

main.js Skript zur Konfiguration und zum Start von Electron sowie der Einbindung der *index.html* in das erzeugte Fenster

webpack.config.js Konfiguration von Webpack

Die entsprechende Ordnerstruktur der Software ist in Abbildung 27 zu sehen.

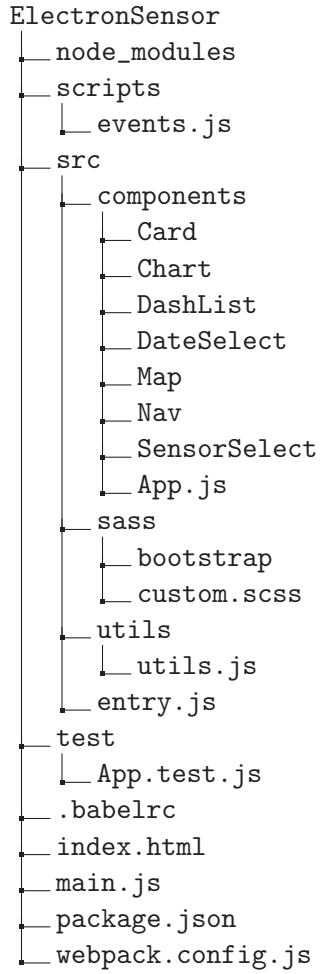


Abbildung 27: Ordnerstruktur des Frontends

Das Zusammenspiel der Dateien ist noch einmal in Abbildung 28 dargestellt.
Die *main.js*-Datei stellt dabei den *Main*-Prozess von *Electron* dar. Sie nutzt die *scripts.js*-Datei, in der die Event-Listener für die Kommunikation mit dem *Renderer*-Prozess beschrieben sind und übergibt die *index.html*-Datei nach der Erzeugung des *Renderer*-Prozesses an selbigen. Die *index.html*-Datei enthält lediglich eine `div` mit der ID „root“ und die Einbindung der *bundle.js*. Die *bundle.js*-Datei ist das Ergebnis des Kompiliervorganges durch *Webpack* und *Babel*. Sie enthält sämtlichen in reines JavaScript gewandelten Quellcode zur Erschaffung der Komponenten und deren Logik. Dazu zählen die *React*-Komponenten, deren Styling mit Hilfe von *Sass*[23] und die *utils.js*-Datei, die einige Funktionen beinhaltet, die komponentenübergreifend benötigt werden. Diese Bestandteile werden von der *App.js*-Datei eingebunden und eingesetzt. Sie stellt selbst eine Komponente dar. Allerdings dient sie der Bildung der gesamten Oberfläche, indem dort alle weiteren Komponenten zum Eunsatz kommen. Weiterer Bestandteil der *bundle.js* ist die *entry.js*-Datei, mit deren Hilfe die *App.js* in das HTML-Dokument integriert wird.

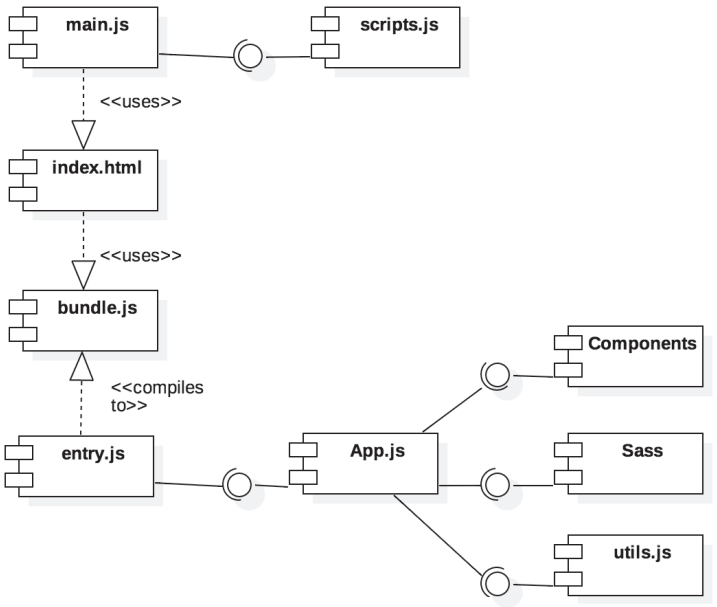


Abbildung 28: Zusammenarbeit der Komponenten des Frontends

[23] (kurz für: **S**yntactilly **A**wesome **S**tyle**s**heets) CSS-Präprozessor, der die Nutzung von Variablen, Schleifen, Bedingungen usw. in Stylesheets ermöglicht

5.5.5 Packager

Nachdem die Entwicklung beendet und der Quellcode kompiliert wurde, kann mit Hilfe des Moduls *electron-packager* eine Betriebssystem-spezifische, ausführbare Datei generiert werden, also beispielsweise eine *.exe*-Datei unter Windows.
Dazu können in der *package.json*-Datei Skripte erstellt werden, welche z.B. durch den Aufruf von `npm run build_win` oder `npm run build_mac` diese ausführbaren Dateien produzieren. Diese Skripte beinhalten den Aufruf von `electron-packager` und konfigurieren das zu erzeugende Programm über diverse Optionen bzw. Argumente.

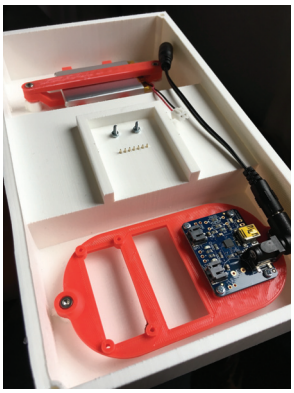
(Vgl. Lee, 2018)

6 Demonstration der praktischen Umsetzung

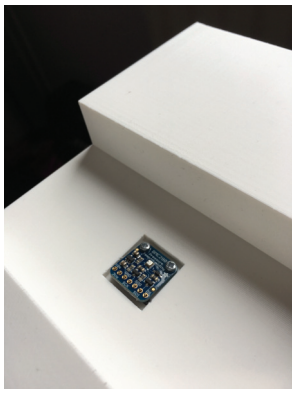
Dieses Kapitel beschäftigt sich mit der Demonstration der erzielten Ergebnisse. Es werden sowohl Elemente des Testaufbaus als auch die Funktionsweise des Backends und der Aufbau des Frontends visualisiert.

6.1 Komponente: Sensorik

Für die Sensorik des Testaufbaus wurde ein Gehäuse entworfen, in dem ein *Adafruit HUZZAH ESP8266*, ein Akku und ein Modul zum Laden des Akkus installiert werden können. Zudem kann auf dem Gehäuse ein Solar-Panel befestigt werden, welches den Akku lädt. An der Unterseite des Gehäuses ist eine Aussparung, die die Befestigung eines Sensors ermöglicht.
Das Gehäuse ist in Abbildung 29 dargestellt.



(a) Befestigungen für Akku, Ladegerät und Mikrocontroller



(b) Aussparung für den Sensor

(c) Solar-Modul auf dem Gehäuse

Abbildung 29: Gehäuse der Sensorik des Testaufbaus

6.2 Komponente: Basisstation

Die Basisstation bildet ein Raspberry Pi Model 3, welcher im dazugehörigen Gehäuse verbaut ist. Das Gerät ist in Abbildung 30 dargestellt.

Abbildung 30: Basisstation des Testaufbaus

6.3 Komponente: Backend

Um Daten an die Schnittstelle zu übertragen, übermittelt die Basisstation einen POST-Request an die Route `/post/`. Dieser wird durch das JavaScript-Modul *request* gebildet und erzeugt bei Daten, wie in Tabelle 3 gezeigt, die folgende URL `http://WEB_API_IP:8000/post?stationId=1&temperature=24&humidity=67&pressure=1013&sound=43` und ruft diese auf.

Parameter	**Wert**
Stations-ID	1
Temperatur	24 °C
Luftfeuchtigkeit	67 %
Luftdruck	1013 hPa
Geräuschpegel	43 %

Tabelle 3: Beispieldaten zur Übertragung an die Web-Schnittstelle

Die erhaltene Antwort erhält bei Erfolg den HTTP-Statuscode **200** und ein JSON-Objekt, welches in Abbildung 31 dargestellt ist.

```
{
    "command": "INSERT",
    "rowCount": 1,
    "rows": [
        {
            "id": 14,
            "inserted": "2018-02-23T08:25:41.379Z",
            "station_id": 1,
            "climate_id": 13,
            "ambient_id": 2,
            "light_id": null,
            "soil_id": null
        }
    ]
}
```


Abbildung 31: JSON-Objekt der Antwort des POST-Requests

Um die im Vorherigen eingegebenen Daten abzurufen, kann die URL `http://WEB_API_IP:8000/get/all/` mit Hilfe der GET-Methode aufgerufen werden. Das zurückgegebene JSON-Objekt ist in Abbildung 32 zu sehen.


```
{
    "command": "SELECT",
    "rowCount": 1,
    "rows": [
        {
            "inserted": "2018-02-23T08:25:41.379Z",
            "temperature": 24,
            "humidity": 67,
            "pressure": 1013,
            "light": null,
            "uv": null,
            "sound": 43,
            "soil_temp": null,
            "moisture": null
        }
    ]
}
```


Abbildung 32: JSON-Objekt der Antwort des GET-Requests

Der Aufruf der Wurzel-Route der Web-Schnittstelle (also `http://WEB_API_IP:8000/`) bietet einen Überblick über die verfügbaren Routen, eine Beschreibung sowie Ausgabe- und Eingabeparameter der jeweiligen Route. Diese kann von Entwicklern eingesehen werden, sodass die Schnittstelle auch in weiteren Projekten genutzt werden kann. Diese Übersicht ist in Abbildung 33 dargestellt.

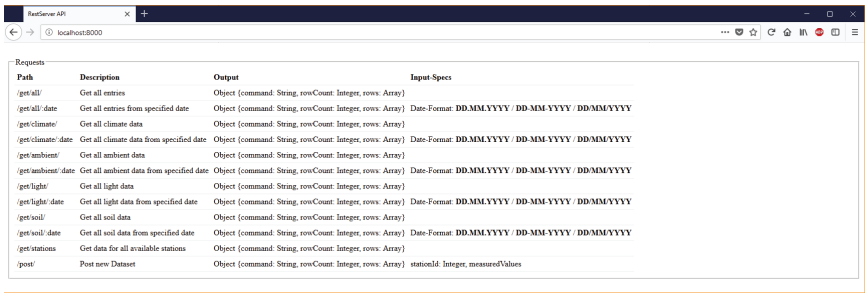


Abbildung 33: Übersichtsseite der Web-Schnittstelle

6.4 Komponente: Frontend

Das Frontend kann auf allen gängigen Betriebssystemen genutzt werden. Bisher integriert ist jedoch nur eine an Windows 10 und Mac OS X angepasste Oberfläche, welche sich je nach Betriebssystem adaptiert.
Grundlegend ist die Oberfläche so gestaltet, dass im oberen Bereich zunächst eine Karte dargestellt wird. Diese wird auf die Position der gewählten Messstation zentriert. Unterstützt wird die Visualisierung der Position zusätzlich durch eine Markierung auf der Karte.
In die Karte eingelassen sind Auswahlfelder für die Messstation und das Datum, mit denen die Parameter der zu betrachtenden Daten bestimmt werden können.
Darunter befindet sich der Bereich, in dem die Daten dargestellt werden. Dieser beinhaltet mehrere Ansichten, welche durch ein Menü gewählt werden können.

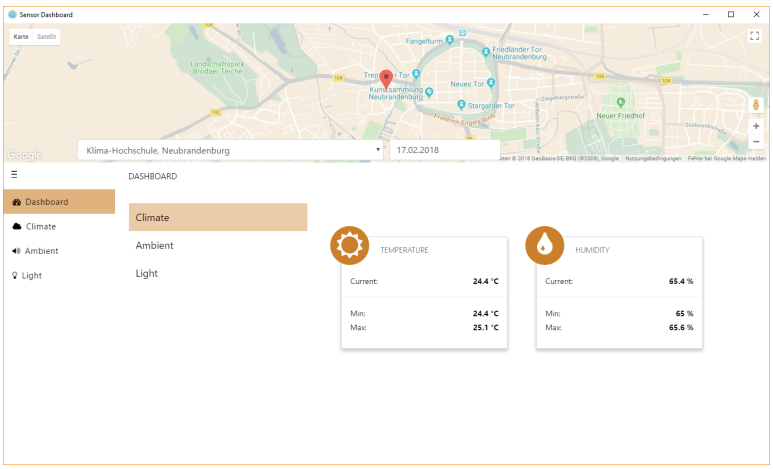


Abbildung 34: Übersichtsseite der Frontend-Applikation unter Windows 10

Das Dashboard bietet einen Überblick über die gemessenen Daten. Darin dargestellt sind jeweils der zuletzt gemessene sowie der maximale und der minimale Wert des gewählten Tages für alle gemessenen Variablen. Unterteilt sind diese dabei in Bereiche wie Klima, Licht, usw. Diese Ansicht unter Windows 10 ist in Abbildung 34 dargestellt.

Die Diagramme sind wie das Dashboard in Bereiche unterteilt. So sind in jedem Diagramm zwei gemessene Variablen zu betrachten. Die Ansicht dieser unter Windows 10 ist in Abbildung 35 und unter Mac OS X in Abbildung 36 zu sehen.

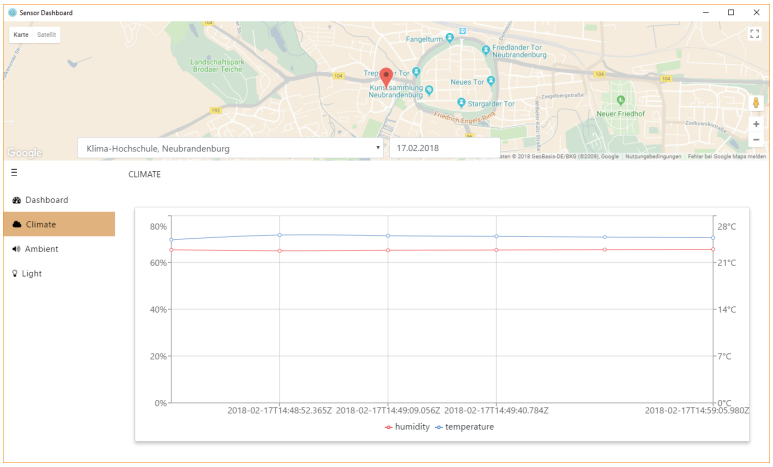


Abbildung 35: Diagramm der Frontend-Applikation unter Windows 10

Abbildung 36: Diagramm der Frontend-Applikation unter Mac OS X

7 Ergebnisse

Dieses Kapitel soll einen Überblick darüber schaffen, welche der beschriebenen Anforderungen des Systems umgesetzt werden konnten. Darüber hinaus werden Erweiterungsmöglichkeiten bzw. Verbesserungen für den zukünftigen Ausbau des Systems thematisiert.

7.1 Zusammenfassung der Arbeit

Ausgehend von der Tatsache, dass der Betrieb aktueller Sensornetzwerke verschiedene Herausforderungen birgt, war es Aufgabe dieser Arbeit, Lösungsansätze zu entwickeln. Die Herausforderungen bestehen insbesondere in:

- Nutzung diverser Programmiersprachen
- Automatisierung
- Konnektivität

Die in der vorliegenden Arbeit vorgestellten Lösungsansätze gestalten sich wie folgt:

Nutzung diverser Programmiersprachen
Durch die Weiterentwicklung von Programmiersprachen und die Schaffung der Laufzeitumgebung *Node.js* war es möglich, mit *JavaScript* auf eine Ressource zurückzugreifen, die es ermöglicht, fast alle Bereiche der benötigten Programmierung zu bedienen. Dadurch wird die Anzahl benötigter Programmiersprachen auf ein Minimum reduziert, was das Verständnis des gesamten Systems erheblich vereinfacht.

Automatisierung
Die zunehmende Verbesserung der Internetanschlussmöglichkeiten - auch in entlegenen Bereichen - begünstigt die Automatisierung des Arbeitsablaufes. Vom Sensor können über die Basisstation Daten zum Backend gereicht werden, ohne dass ein Eingreifen durch den Menschen notwendig ist. Voraussetzung dafür ist eine Netzwerkverbindung aller Komponenten.

Konnektivität
Durch die Entwicklung netzwerkfähiger Mikrocontroller kann auf die Nutzung proprietärer Anschlüsse verzichtet werden. Somit stehen für die Sensoren ein einheitlicher Anschluss und ein einheitliches Kommunikations-Protokoll (*Firmata*) zur Verfügung.

Die in der Arbeit entwickelte Softwarelösung unterteilt sich in vier Komponenten.

Sensorik
Die für die Sensorik genutzten Skripte basieren auf der Programmiersprache C. Das begründet sich darin, dass diese Skripte bereits durch die *Arduino IDE* gegeben sind. Zudem würde die Nutzung der gewählten Laufzeitumgebung *Node.js* die Leistung der Mikrocontroller erheblich einschränken.
Nach dem Start der Mikrocontroller erwarten sie Befehle über das *Firmata*-Protokoll.

Basisstation
Die Befehle an die Mikrocontroller werden von der Basisstation abgesetzt.
Sie enthält eine Installation von *Node.js* und betreibt ein Skript, in dem diese Befehle mit Hilfe diverser Konfigurationsdateien beschrieben werden können. Die Befehle veranlassen die Abfrage der Messdaten der Sensoren.
Nachdem die Basisstation die Messdaten erhalten hat, werden diese mit Hilfe eines POST-Requests an eine Web-Schnittstelle weitergeleitet.

Web-Schnittstelle
Nach Eingang der Messdaten in der Web-Schnittstelle, veranlasst sie die Speicherung der Daten in der verbundenen Datenbank. Diese Verbindung kann wiederum mit Hilfe von Konfigurationsdateien definiert werden.
Nach der Speicherung der Daten können diese vom Nutzer bzw. vom Frontend über von der Web-Schnittstelle bereitgestellte Routen abgefragt werden.

Frontend
Das Frontend bietet eine Nutzeroberfläche. Auf dieser können Parameter, wie z.B. Messstation oder Beobachtungszeitraum, festgelegt werden. Nach der Auswahl dieser Parameter fragt das Frontend die entsprechenden Werte von der Web-Schnittstelle per GET-Request ab und visualisiert diese in verschiedenen Diagrammen.

7.2 Ausblick auf zukünftige Verbesserungsmöglichkeiten

Im Folgenden werden Erweiterungsmöglichkeiten für die Systemkomponenten vorgestellt.

Sensorik
Durch die Entwicklung weiterer Mikrocontroller, wie z.B. dem *ESP32*, besteht auch die Möglichkeit des Anschlusses mehrerer analoger Sensoren an einem Gerät. Zudem kann die Konfiguration digitaler Sensoren in das Skript integriert werden.

Basisstation
Durch die Implementierung einer Funktion in JavaScript, die das Versetzen des Gerätes in den Ruhemodus unterstützt, könnte die Basisstation sowohl an Langlebigkeit gewinnen als auch den Stromverbrauch verringern. Eine solche Funktion wäre auch auf Ebene der Sensorik wünschenswert (z.B. Erweiterung durch Wake-on-LAN (Siehe auch: Tang; Yomo; Kondo; Obana, 2012)).
Des Weiteren besteht die Möglichkeit der Einbindung der *station.json*-Datei, in welcher Daten über die jeweilige Basisstation spezifiziert werden, zur Authentifizierung bzw. automatischen Registrierung neuer Basisstationen.

Backend
Auf Ebene des Backends wurden in diesem Projekt kaum Sicherheitsmaßnahmen betrachtet. Diese gewinnen außerhalb eines Testaufbaus eine primäre Bedeutung. Dazu gehören unter anderem *HTTPS*, Authentifizierung und ein Rechte- und Rollen-System.
Außerdem können je nach Anzahl der Nutzer die Anforderungen an die Hardware variieren. Somit können bei Bedarf die Möglichkeiten des Clusterings ausgeschöpft werden.

Frontend
Bei Umsetzung eines Rechte- und Rollen-Systems im Backend ist die Anpassung des Frontends dahingehend notwendig. Dies kann einen Login-Bereich sowie eine Nutzerverwaltung für den Systemadministrator umfassen.
Zudem wäre die Optimierung der Benutzeroberfläche in Betracht auf die Datennutzung wünschenswert. Dazu gehören unter anderem die Möglichkeit der Auswahl eines Betrachtungszeitraums (bisher nur ein Tag) sowie der Vergleich diverser Datensätze untereinander.

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis von Codebeispielen

Literatur

BRINKSCHULTE, Uwe; UNGERER, Theo, 2010. *Mikrocontroller und Mikroprozessoren*. 3. Aufl. Berlin, Heidelberg: Springer-Verlag. ISBN 978-3-642-05398-6.

BULL, Gabriel, 2018. *React Desktop: React UI Components for macOS High Sierra and Windows 10* [online]. GitHub, Inc. [besucht am 1. März 2018]. Abgerufen unter: `https://github.com/gabrielbull/react-desktop`.

CARLSON, Brian. *node-postgres* [online] [besucht am 5. Feb. 2018]. Abgerufen unter: `https://node-postgres.com/`.

CARLSON, Brian, 2018. *pg* [online]. npm, Inc. [besucht am 5. Feb. 2018]. Abgerufen unter: `https://www.npmjs.com/package/pg`.

CHAI.JS, 2017. *Chai* [online] [besucht am 9. Feb. 2018]. Abgerufen unter: `http://chaijs.com`.

CHEN, Tom, 2018. *react-google-maps: React.js Google Maps integration component* [online]. GitHub, Inc. [besucht am 1. März 2018]. Abgerufen unter: `https://github.com/tomchentw/react-google-maps`.

CHOUDHARY, Himanshu, 2012. *Difference between Microprocessor and Microcontroller* [online] [besucht am 19. Jan. 2018]. Abgerufen unter: `https://www.engineersgarage.com/tutorials/difference-between-microprocessor-and-microcontroller`.

CLARK, Tim, 2013a. HTTP. In: *Encyclopedia of Systems Biology*. Hrsg. von DUBITZKY, Werner; WOLKENHAUER, Olaf; CHO, Kwang-Hyun; YOKOTA, Hiroki. New York, NY: Springer New York, S. 925–926. ISBN 978-1-4419-9863-7.

CLARK, Tim, 2013b. HTTPS. In: *Encyclopedia of Systems Biology*. Hrsg. von DUBITZKY, Werner; WOLKENHAUER, Olaf; CHO, Kwang-Hyun; YOKOTA, Hiroki. New York, NY: Springer New York, S. 926. ISBN 978-1-4419-9863-7.

CLARK, Tim, 2013c. Uniform Resource Identifier (URI). In: *Encyclopedia of Systems Biology*. Hrsg. von DUBITZKY, Werner; WOLKENHAUER, Olaf; CHO, Kwang-Hyun; YOKOTA, Hiroki. New York, NY: Springer New York, S. 2319–2320. ISBN 978-1-4419-9863-7.

DEMBOWSKI, Klaus, 2015. *Raspberry Pi - Das technische Handbuch: Konfiguration, Hardware, Applikationserstellung*. 2. Aufl. Wiesbaden: Springer Vieweg. ISBN 978-3-658-08711-1.

ECMA INTERNATIONAL, 2017. *Standard ECMA-262: ECMAScript 2017 Language Specification* [online] [besucht am 23. Feb. 2018]. Abgerufen unter: `https://www.ecma-international.org/publications/standards/Ecma-262.htm`.

ENGEL, Philipp, 2013. *Entwicklung eines autonomen Low-Cost-Messclients für geodätische Überwachungsmessungen*. Masterarbeit. Hochschule Neubrandenburg.

FACEBOOK INC., 2018. *React* [online] [besucht am 7. Feb. 2018]. Abgerufen unter: `https://reactjs.org`.

FIELDING, Roy Thomas, 2000. *Architectural styles and the design of network-based software architectures*. Dissertation. University of California, Irvine.

FORSCHUNGSZENTRUM JÜLICH GMBH, 2017a. *Ausstattung mit Messgeräten* [online] [besucht am 24. Jan. 2018]. Abgerufen unter: `http://teodoor.icg.kfa-juelich.de/overview-de/tereno-instumentation`.

FORSCHUNGSZENTRUM JÜLICH GMBH, 2017b. *Durchführung* [online] [besucht am 24. Jan. 2018]. Abgerufen unter: `http://teodoor.icg.kfa-juelich.de/overview-de/tereno-implementation`.

FORSCHUNGSZENTRUM JÜLICH GMBH, 2017c. *Ziele* [online] [besucht am 24. Jan. 2018]. Abgerufen unter: `http://teodoor.icg.kfa-juelich.de/overview-de/tereno-goals`.

GELLERT, Laurence, 2012. *What is a Full Stack developer?* [online] [besucht am 6. Feb. 2018]. Abgerufen unter: `https://www.laurencegellert.com/2012/08/what-is-a-full-stack-developer/`.

GITHUB, INC., 2018. *Electron: Plattformübergreifende Desktop-Anwendungen mit JavaScript, HTML und CSS entwickeln* [online] [besucht am 19. Jan. 2018]. Abgerufen unter: `https://electronjs.org`.

GOOGLE LLC, 2018. *Chromium: The Chromium Projects* [online] [besucht am 22. Jan. 2018]. Abgerufen unter: `https://www.chromium.org/Home`.

HABLICH, Michael, 2017. *What is V8?* [online]. GitHub, Inc. [besucht am 24. Jan. 2018]. Abgerufen unter: `https://github.com/v8/v8/wiki`.

HAHN, Evan, 2018. *helmet* [online]. npm, Inc. [besucht am 5. Feb. 2018]. Abgerufen unter: `https://www.npmjs.com/package/helmet`.

HARTMANN, Nils, 2015. *React Anwendungen mit Webpack entwickeln* [online] [besucht am 7. Feb. 2018]. Abgerufen unter: `http://reactjs.de/posts/react-anwendungen-mit-webpack-entwickeln`.

HOEFS, Jeff, 2018. *Firmata Protocol Documentation*. GitHub.

ISFAN, Petru, 2017. *supervisor* [online]. npm, Inc. [besucht am 7. Feb. 2018]. Abgerufen unter: `https://www.npmjs.com/package/supervisor`.

LEE, Mark, 2018. *electron-packager* [online]. npm, Inc. [besucht am 7. Feb. 2018]. Abgerufen unter: `https://www.npmjs.com/package/electron-packager`.

LINDSTRÖM, Sebastian, 2017. *Getting to know asynchronous JavaScript: Callbacks, Promises and Async/Await* [online]. Medium [besucht am 28. Feb. 2018]. Abgerufen unter: `https://medium.com/codebuddies/getting-to-know-asynchronous-javascript-callbacks-promises-and-async-await-17e0673281ee`.

LORA ALLIANCE™, 2018. *LoRa Alliance™ Technology* [online] [besucht am 29. Jan. 2018]. Abgerufen unter: `https://www.lora-alliance.org/technology`.

MAINWARING, Alan; CULLER, David; POLASTRE, Joseph; SZEWCZYK, Robert; ANDERSON, John, 2002. Wireless Sensor Networks for Habitat Monitoring. In: *Wireless Sensor Networks for Habitat Monitoring. Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*. Atlanta, Georgia, USA: ACM, S. 88–97. WSNA '02. ISBN 1-58113-589-0.

MATTERN, Friedemann; RÖMER, Kay, 2003. Informatik-Spektrum: Drahtlose Sensornetze. 26. Aufl., S. 191.

MEIER, Andreas; KAUFMANN, Michael, 2016. *SQL- & NoSQL-Datenbanken*. 8. Aufl. Berlin, Heidelberg: Springer Vieweg. ISBN 978-3-662-47663-5.

MOCHAJS.ORG, 2018. *Mocha: The fun, simple, flexible JavaScript test framework* [online] [besucht am 9. Feb. 2018]. Abgerufen unter: `https://mochajs.org`.

NEBEL, Rodrigo; AWAD, Abdalkarim; GERMAN, Reinhard; DRESSLER, Falko, 2007. pimoto - Ein System zum verteilten passiven Monitoring von Sensornetzen. In: HOLLECZEK, Peter; VOGEL-HEUSER, Birgit (Hrsg.). *Mobilität und Echtzeit*. Berlin, Heidelberg: Springer Berlin Heidelberg, S. 1–10. ISBN 978-3-540-74837-3.

NPM, INC. *npm* [online] [besucht am 24. Jan. 2018]. Abgerufen unter: `https://www.npmjs.com`.

PICK, Charles, 2015. *Why Babel matters: Why Babel is different from other compile-to-JS systems like CoffeeScript and TypeScript, and how it's going to become the driving force for innovation in JavaScript.* [online] [besucht am 7. Feb. 2018]. Abgerufen unter: `https://codemix.com/blog/why-babel-matters/`.

POMASKA, Günter, 2012. *Webseiten-Programmierung: Sprachen, Werkzeuge, Entwicklung*. Wiesbaden: Springer Vieweg. ISBN 978-3-8348-2485-1.

RASPBERRY PI FOUNDATION. *Setting up a Raspberry Pi as an Access Point in a standalone Network* [online] [besucht am 3. Feb. 2018]. Abgerufen unter: `https://www.raspberrypi.org/documentation/configuration/wireless/access-point.md`.

RECHARTS GROUP, 2018. *Recharts: Redefined chart library built with React and D3* [online]. GitHub, Inc. [besucht am 1. März 2018]. Abgerufen unter: `https://github.com/recharts/recharts`.

SAUER, Uta, 2016. *TERENO* [online]. Helmholtz-Zentrum für Umweltforschung UFZ [besucht am 23. Jan. 2018]. Abgerufen unter: `https://www.ufz.de/index.php?de=37279`.

SPRINGER, Sebastian, 2016. *Node.js: Das Praxisbuch*. 2. Aufl. Bonn: Rheinwerk Verlag GmbH. ISBN 978-3-8362-4003-1.

STENDER, Vivien, 2017. *TERENO: Terrestrial Environmental Observatories* [Präsentation]. Deutsches GeoForschungsZentrum Potsdam.

TANG, Suhua; YOMO, Hiroyuki; KONDO, Yoshihisa; OBANA, Sadao, 2012. Wake-up receiver for radio-on-demand wireless LANs. *EURASIP Journal on Wireless Communications and Networking*. Jg. 2012, Nr. 1.

TILKOV, Stefan, 2011. *REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien*. 2. Aufl. Heidelberg: dpunkt.verlag. ISBN 978-3-898-64732-8.

WILSON, Douglas, 2017. *express* [online]. npm, Inc. [besucht am 5. Feb. 2018]. Abgerufen unter: `https://www.npmjs.com/package/express`.

Anhänge

A Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

08.03.2018

(Datum) (Unterschrift)