Abteilung: Höhere Lehranstalt für Informatik

Diplomarbeit

im Studiengang Informatik

Thema:	Integration von Sensoren in ein bestehendes Netzwerk für Hausautomation
eingereicht von:	Mathias Praher <mathias.praher@gmail.com> Hannes Lumetzberger <hannes.lumetzberger@gmail.com>
eingereicht am:	22. Oktober 2015
Betreuer:	Prof. Dipl.-Ing. Dr. Michael Buchberger
In Zusammenarbeit mit:	FH Hagenberg

Eidestaatliche Erklärung

Die unterfertigten Kandidaten / Kandidatinnen haben gemäß § 34 (3) SchUG in Verbindung mit § 22 (1) Zi. 3 lit. b der Verordnung über die abschließenden Prüfungen in den berufsbildenden mittleren und höheren Schulen, BGBl. II Nr. 70 vom 24.02.2000 (Prüfungsordnung BMHS), die Ausarbeitung einer Diplomarbeit mit der umseitig angeführten Aufgabenstellung gewählt.

Die Kandidaten / Kandidatinnen nehmen zur Kenntnis, dass die Diplomarbeit in eigenständiger Weise und außerhalb des Unterrichtes zu bearbeiten und anzufertigen ist, wobei Ergebnisse des Unterrichtes mit einbezogen werden können.

Die Abgabe der Diplomarbeit hat bis spätestens 08.04.2016 beim zuständigen Betreuer / der zuständigen Betreuerin zu erfolgen.

Die Kandidaten / Kandidatinnen nehmen weiters zur Kenntnis, dass gemäß § 9 (6) der Prüfungsordnung BMHS nur der Schulleiter bis spätestens Ende des vorletzten Semesters den Abbruch einer Diplomarbeit anordnen kann, wenn diese aus nicht beim Prüfungskandidaten (bei den Prüfungskandidaten) gelegenen Gründen nicht fertiggestellt werden kann.

Kandidaten / Kandidatinnen inkl. Unterschrift:

Ort, Datum	Hannes Lumetzberger
Ort, Datum	Mathias Praher

Inhaltsverzeichnis

1 Einleitung 7
1.1 Einführung 8
1.2 Danksagung 9

2 Grundlagen 11
2.1 JavaScript Framework 12
2.2 HTML 5 12
2.2.1 Ziele 12
2.2.2 Technische Neuerungen 12
2.3 SASS 13
2.3.1 Unterschiede zu CSS 13
2.4 AngularJS 14
2.4.1 Konzept 15
2.4.2 Struktur 15
2.5 Ionic 17
2.6 NoSQL-Datenbanken 19
2.6.1 Unterschiede zu SQL 19
2.6.2 Ansätze für NoSQL-Datenbanken 19
2.6.3 CouchDB 22
2.7 Maven 26
2.7.1 Konfigurationsdatei 26
2.7.2 Lebenszyklus 26
2.7.3 M2Eclipse 27
2.8 Java Persistence API 29
2.8.1 Aufbau 29
2.8.2 Verwendung 30
2.8.3 Ektorp 31
2.9 NodeJS 33
2.9.1 Programmaufbau 33
2.9.2 Node Package Manager 33
2.9.3 Serialport 34
2.10 MeshSSN 35
2.10.1 Das Mesh-Protokoll 35
2.10.2 Datenverarbeitung 36
2.11 HomeMatic 37
2.11.1 CCU2-Zentrale 37
2.11.2 HomeMaticScript 41
2.12 XMLRPC 42
2.12.1 Ziele der Schnittstelle 42
2.12.2 Protokoll 42
2.12.3 HomeMatic-XMLRPC-Schnittstelle 43
2.13 SSH 46
2.13.1 WinSCP 46
2.14 Tool Command Language 47

3 Benutzung **49**
3.1 Sensordaten-Server – Konfiguration 50
3.1.1 HomeMaticServer 50
3.1.2 SensorDataServer 50
3.1.3 CouchDBServer 50
3.1.4 HomeMaticWriteMethod 51
3.1.5 CouchDBLogSkips 51
3.1.6 CouchDBMaxEntries 51
3.1.7 DataLoadingTimeout 51
3.2 App Oberflächen 52
3.2.1 Home-Ansicht 53
3.2.2 Übersicht der Komponenten 53
3.2.3 Detailansicht für Komponenten 53
3.2.4 Konfiguration der Verbindung 53

4 Schnittstellen **55**
4.0.5 Sensoren der MeshSSN einlesen 56
4.0.6 Daten in CouchDB schreiben 58
4.0.7 Daten in HomeMatic schreiben 59

5 Implementierung **61**
5.1 Aufbau des HomeMatic-Systems 62
5.1.1 Geräte zur CCU2 hinzufügen 62
5.2 App 63
5.2.1 Implementierung GUI 63
5.2.2 App-Routing 63
5.2.3 Persistierung der Einstellungen 63
5.2.4 Verbindung mit XML-RPC Service 64
5.2.5 Dynamisches Generieren der Detailansicht 65
5.2.6 Struktur 66
5.3 Server 67
5.3.1 Installation 67
5.3.2 Erstellung des Maven-Projektes 69
5.3.3 Programmaufbau 72
5.3.4 Start des Programms 73
5.3.5 Laden der Daten aus der Konfigurationsdatei 74
5.3.6 Einlesen der Sensordaten von MeshSSN 75
5.3.7 Schreiben der Sensordaten in CouchDB 76
5.3.8 Schreiben der Sensordaten in HomeMatic 79
5.4 Umschreiben der HomeMatic-Systemdateien um ein Gerät hinzuzufügen 84
5.4.1 Erzeugen eines System-Geräts mit CUxD 84
5.4.2 Ausführen eines Kommandozeilenbefehls mit dem System-Gerät von CUxD 85
5.4.3 “homematic.regadom” 86
5.4.4 TCL-Script zum Umschreiben der “homematic.regadom”-Datei schreiben 90
5.4.5 Das Verhalten des neuen Geräts 93

6 Beurteilung **95**
6.1 App 96
6.2 Server 96

Literaturverzeichnis **96**

Abbildungsverzeichnis **99**

Quellcodeverzeichnis **102**

KAPITEL 1

Einleitung

1.1 Einführung

Ziel unserer Diplomarbeit ist es, selbstgebaute Sensoren in ein bestehendes Hausautomationssystem von HomeMatic zu integrieren. Die Sensoren wurden im Rahmen einer Masterarbeit an der FH-Hagenberg von fünf Studenten konstruiert und uns für unsere Diplomarbeit zur Verfügung gestellt.
Durch die Integration der selbstgebauten Sensoren in das HomeMatic-System soll es ermöglicht werden, dass beispielsweise ein Rollladenaktor des Systems die Jalousien herunterfährt, wenn einer der selbstgebauten Sensoren einen gewissen Schwellenwert bei seiner gemessenen Temperatur übersteigt. Des Weiteren sollen die Daten der Sensoren in einer Datenbank archiviert werden. Um das alles zu gewährleisten, schreiben wir einen Server, der die Daten von den selbstgebauten Sensoren einliest und diese anschließend an eine CouchDB-Datenbank und an die HomeMatic-Zentrale weiterleitet. Zum Schluss wird auch noch eine App implementiert, die auf die HomeMatic-Geräte zugreift.

1.2 Danksagung

Allen voran möchten wir Herrn Prof. Dipl.-Ing. Dr. Michael Buchberger danken, da er diese Diplomarbeit durch seinen Kontakt mit der FH Hagenberg erst möglich gemacht hat und uns als Betreuungslehrer dieser Diplomarbeit immer mit Rat und Tat zur Seite gestanden ist. Des weiteren bedanken wir uns bei der Studentengruppe, die die Masterarbeit durchgeführt und geschrieben hat. Diese haben uns nicht nur die Sensoren ihres Sensornetzwerks überlassen, sondern haben uns auch bei allen unseren Fragen zu diesen Sensoren bereitwillig weitergeholfen. Außerdem möchten wir der HTL Perg unseren Dank aussprechen, denn diese hat uns die Aktoren, Sensoren und Zentralen von HomeMatic zur Verfügung gestellt. Hierbei möchten wir zusätzlich noch Herrn Professor Dipl.-Ing. Richard Kainerstorfer danken, denn er ist für die Bestellung dieser Artikel zuständig gewesen.

Grundlagen

2.1 JavaScript Framework

JavaScript Frameworks sind Programmiergerüste, die den Rahmen zur Verfügung stellen, in dem ein Programmier seine Applikation erstellen kann. Ralph E. Johnson und Brian Foot definierten den Begriff Framework in ihrem Artikel „Designing Reusable Classes“ vom Jahr 1988 folgendermaßen: „Ein Framework ist eine semi-vollständige Applikation. Es stellt für Applikationen eine wiederverwendbare, gemeinsame Struktur zur Verfügung. Die Entwickler bauen das Framework in ihre eigene Applikation ein, und erweitern es derart, dass es ihren spezifischen Anforderungen entspricht. Frameworks unterscheiden sich von Toolkits dahingehend, dass sie eine kohärente Struktur zur Verfügung stellen, anstatt einer einfachen Menge von Hilfsklassen.“ Frameworks unterscheiden sich im Allgemeinen dadurch von Klassenbibliotheken, dass Frameworks den selbst programmierten Quelltext steuern beziehungsweise benutzen, während Klassenbibliotheken lediglich Klassen und Funktionen zur Verfügung stellen, die bei Implementierungen verwendet werden können.

2.2 HTML 5

HTML 5 ist die fünfte Version der Hypertext Markup Language. Die Spezifikation der Auszeichnungssprache wurde 2014 vom „World Wide Web Consortium“, dem Gremium zur Standardisierung von Techniken im World Wide Web, vorgelegt. Damit wurde der 1997 spezifizierte Vorgänger HTML 4.0 vom neueren Standard abgelöst.

2.2.1 Ziele

- Bestehende Inhalte müssen weiterhin kompatibel sein und neue Elemente sollten die alten nicht negativ beeinflussen.
- Teile aus XML sollten auch in HTML erlaubt werden.
- Bei der Entwicklung neuer Funktionen müssen Sicherheitsaspekte berücksichtigt werden.
- Das Verhalten sollte so genau wie möglich definiert und die Komplexität so gering wie möglich sein.
- HTML soll auf allen Endgeräten und mit Inhalten in allen Weltsprachen verwendbar sein.

2.2.2 Technische Neuerungen

Dokumenttypangabe

Um den Browsern mitzuteilen, den Quelltext im standardkonformen Modus zu verarbeiten wird folgende Dokumenttypangabe am Beginn von HTML-Dokumenten verwendet:

```
1 <!DOCTYPE html>
```

Neue Elemente

HTML 5 brachte eine Vielzahl von neuen Elementen mit sich. Dazu zählen beispielsweise strukturierende Elemente, bei denen im Vergleich zum konventionellen „div“-Element auch die Art des Inhalts definiert werden kann. Außerdem wurden Gruppierungselemente, für Bildunterschriften bei Abbildungen, oder Multimedia-Elemente zur Einbindung von Video- und Audiodateien zu HTML hinzugefügt.

Geänderte Bedeutung für manche Elemente

Manchen darstellenden Elementen wurde eine semantische Bedeutung gegeben, während andere Elemente umdefiniert bzw. aus dem Vokabular gelöscht wurden. Auch bei diesen Neudefinitionen wurde darauf geachtet, dass diese relativ weit gefasst sind, um garantieren zu können, dass diese auch bei existierenden Webseiten noch funktionieren.

2.3 SASS

Syntactically Awesome Stylesheets ist wie der Name bereits verrät eine Stylesheet-Sprache. SASS dient allerdings nur dazu die Erzeugung von CSS zu erleichtern indem die SASS-Datei in eine CSS-Datei kompiliert wird, die vom Browser gelesen werden kann. SASS verfügt mittlerweile außerdem über eine neuere Syntax, die SCSS (Sassy CSS) genannt wird. Der große Unterschied zwischen der SASS- beziehungsweise der SCSS-Syntax ist, dass bei SCSS geschwungene Klammern als Verschachtelungselemente dienen. Bei SASS wird dies rein über die Einrückungen des Quelltextes umgesetzt.[Sas15]

2.3.1 Unterschiede zu CSS

- Geschachtelte Regeln
  - In SASS ist es möglich geschachtelte Selektoren zu verwenden. Diese dienen dazu, die Lesbarkeit komplexerer Selektoren zu erhöhen.

```
1  nav {
2     ul {
3        margin: 0;
4        padding: 0;
5        list-style: none;
6  }
```

Listing 2.1: Verschachtelungen in SCSS

- Variablen
  - Um in großen Stylesheets mehr Übersicht zu schaffen, können in SASS Variablen genutzt werden in denen zum Beispiel Stylefarben gespeichert werden, die immer wieder relevant sind.

```
1        $font-stack:  Helvetica, sans-serif;
2        $primary-color: #333;
3
4        body {
5           font: 100% $font-stack;
6           color: $primary-color;
7        }
```

Listing 2.2: Definition von Variablen in SCSS

- Mixins
  - Mixins ähneln den Funktionen in anderen Programmiersprachen und auch Mixins können Variablen übergeben werden. Außerdem können auch Mixins geschachtelte Selektoren enthalten.

```
1     @mixin border-radius($radius) {
2        -webkit-border-radius: $radius;
3        -moz-border-radius: $radius;
4        -ms-border-radius: $radius;
5        border-radius: $radius;
6     }
7     .box { @include border-radius(10px); }
```

Listing 2.3: Definition vonm Mixins in SCSS

2.4 AngularJS

AngularJS bezeichnet ein clientseitiges JavaScript-Webframework. Das Framework wird hauptsächlich für dynamische HTML-Webanwendungen angewandt, die nach dem "Model-View-ViewModel" Entwurfsmodell aufgebaut sind.

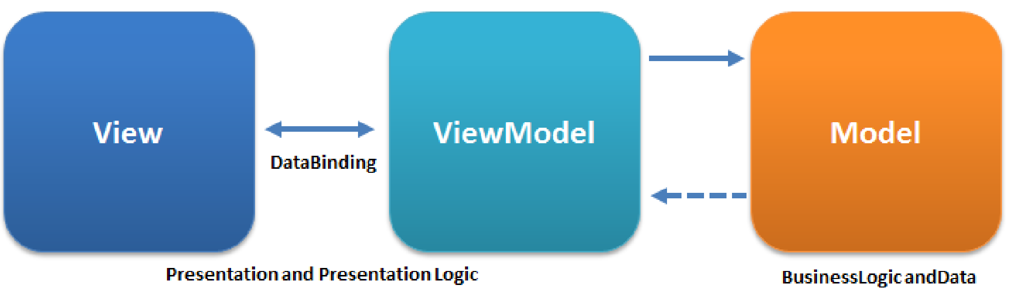


Abbildung 2.1: Model-View-ViewModel Entwurfsmodell [Uga12]

Dieses Entwurfsmuster basiert auf dem Model-View-Controller-Muster und ermöglicht die Trennung von Logik und Darstellung der Benutzeroberfläche. Das *Model* enthält das Datenmodell der

Applikation und dient dazu, dem Benutzer den Zugriff auf die Daten zu ermöglichen. Außerdem wird die Geschäftslogik der Anwendung im Model implementiert. In der *View* werden alle Elemente der grafischen Benutzeroberfläche angezeigt sowie die Datenbindung auf das ViewModel programmiert. Das *ViewModel* ist das Bindeglied von Model und View und implementiert die Logik der Benutzeroberfläche. Es holt sich Daten aus dem Model und bereitet diese für die View auf. Bei der Entwicklung kann dadurch eine Rollenunterscheidung von Designern, die spezialisiert auf die Benutzeroberfläche der Anwendung sind, und Entwicklern, die beispielsweise die Geschäftslogik programmieren, vorgenommen werden. Angular wird vom amerikanischen Softwareriesen Google entwickelt und ist ein Open-Source-Projekt.[Goo16]

2.4.1 Konzept

Angular dient zur clientseitigen Generierung von HTML-Ansichten und Erweiterung des HTML-Vokabulars. Dadurch kann Funktionalität in die Anzeige der Web-Applikation eingebunden werden, ohne DOM-Manipulationen mit Javascript Frameworks vornehmen zu müssen. DOM ist die Abkürzung für das Document Object Model, einer Schnittstellenspezifikation für den Zugriff auf HTML oder XML Dokumente, die den Aufbau solcher Dokumente beschreibt. Sobald der Browser eine Webseite lädt, erstellt er das DOM dieser Seite, also einen Baum von Objekten. Mit JavaScript kann man auf diese Objekte zugreifen, diese verändern, löschen oder auch neue hinzufügen. Diesen Zugriff nennt man DOM-Manipulation.
Auch Datenvalidierungen werden in Angular im Rahmen von Eingabeformularen als Funktionalität in der View umgesetzt.

2.4.2 Struktur

Controller

Im Controller werden in Angular das clientseitige Modell und die Logik der Applikation implementiert. Diese Controller werden in Modulen zusammengefasst, welche wiederum in die Webanwendung eingebunden werden. Dadurch können die Benutzeroberfläche und das Modell verbunden werden.

Direktiven

Angular bietet die Möglichkeit benutzerdefinierte HTML-Attribute zu definieren. Mit benutzerdefinierten HTML-Attributen sind Attribute gemeint, die nicht vordefiniert sind, sondern vom Programmierer angelegt und deshalb auch ihre Funktion definiert werden. Diese Elemente werden in Angular Direktiven genannt. Angular stellt auch bereits vordefinierte Direktiven zur Verfügung, welche daran erkennbar sind, dass sie mit der Silbe „ng“ beginnen.

Erstellen von Direktiven

```
1 angular.module('docsSimpleDirective', [])
2 .controller('Controller', ['$scope', function($scope) {
3    $scope.customer = {
4       name: 'Naomi',
5       address: '1600 Amphitheatre'
6    };
7 }])
8
```

```
9  .directive('myCustomer', function() {
10    return {
11       template: 'Name: {{customer.name}} Address: {{customer.address}}'
12    };
13 });
14 
15 <div ng-controller="Controller">
16 <div my-customer></div>
17 </div>
```

Listing 2.4: Erstellen neuer Direktiven

In obigem Beispiel wird zuerst ein Controller angelegt in dem ein Kunde gespeichert wird. Über das *$scope* kann man in der angelegten Direktive *myCustomer* auf den gespeicherten Kunden zugreifen. Im HTML-Beispielcode wird lediglich der Controller geladen und die angelegte Direktive in ein div-Element geschrieben.

Interpolation

Um Javascript-Ausdrücke im HTML-Code zu verwenden, wird in Angular sogenannte „double-curly-syntax“ verwendet. Das heißt, die Ausdrücke werden in doppelte geschwungene Klammern eingebettet. Zusätzlich können mit dem Pipe-Operator verschiedene Filter auf das Ergebnis angewandt werden. In folgendem Beispiel wird im Controller ein Model definiert. Im Fall des Beispiels eine Liste von Telefonnamen mit einer kleinen Beschreibung. Im HTML-Abschnitt wird diese Liste mithilfe der *ng-repeat* Direktive durchgelaufen und diese in einer Liste angezeigt. Mithilfe des Pipe-Operators kann nach einem bestimmten Schlüsselwort gesucht werden, das in das Eingabefeld darüber eingegeben wurde.

```
1  var phonecatApp = angular.module('phonecatApp', []);
2  
3  phonecatApp.controller('PhoneListCtrl', function ($scope) {
4     $scope.phones = [
5        {'name': 'Nexus S',
6        'snippet': 'Fast just got faster with Nexus S.'},
7        {'name': 'Motorola XOOM with Wi-Fi',
8        'snippet': 'The Next, Next Generation tablet.'},
9        {'name': 'MOTOROLA XOOM',
10       'snippet': 'The Next, Next Generation tablet.'}
11    ];
12 });
13 
14 <div class="col-md-2">
15     Search: <input ng-model="query">
16 </div>
17 <ul class="phones">
18    <li ng-repeat="phone in phones | filter:query">
19       {{phone.name}}
20       <p>{{phone.snippet}}</p>
21     </li>
22 </ul>
```

2.5 Ionic

Ionic ist ein Frontend-Framework zur Implementierung mobiler Smartphone Applikationen. Frontends sind der benutzernahe Teil einer Software, das heißt in den meisten Fällen bezieht sich Frontend auf die Benutzeroberfläche der Software. Ionic ist ein Open-Source-Framework und basiert auf HTML 5. Die Hauptziele des Frameworks sind einerseits die Erhöhung der Benutzerfreundlichkeit und andererseits die Geschwindigkeit mobiler Applikationen zu verbessern. Um die Struktur von Ionic Applikationen regeln zu können, wird AngularJS verwendet. Zudem zählt Ionic zu den mobilen Hybrid-Frameworks. Hybrid-Applikationen werden sowohl auf Smartphones als auch auf Tablets betrieben, wobei Web-Technologien wie HTML, Javascript, CSS und SASS zur Implementierung verwendet werden. Für den Benutzer vermitteln Hybrid-Applikationen das Gefühl von nativen Smartphone Anwendungen, allerdings werden diese in einer Browser-Umgebung betrieben. Ein großer Vorteil von Hybrid-Applikationen ist zudem, dass diese für sowohl Android, als auch für iOS entwickelt werden können, ohne irgendwelche Code-Änderungen machen zu müssen. [Ion16]

Ionic installieren

Um Ionic-Projekte erstellen zu können, muss Node JS 4 bereits vorinstalliert sein. Mehr zu Node JS 4 können Sie im Grundlagenkapitel 2.9 NodeJS lesen. Mit den folgenden Kommandozeilenbefehlen kann Ionic installiert und ein Projekt angelegt werden:

```
1 $ npm install -g cordova ionic
2 $ ionic start myApp blank
3 $ cd myApp
4 $ ionic platform add ios [android]
5 $ ionic build ios
6 $ ionic emulate ios
```

Listing 2.5: Kommandozeilenbefehle zur Installation von Ionic[Ion15]

Mit dem ersten Befehl werden die neuesten Versionen der Kommandozeilenapplikationen Apache Cordova und Ionic installiert. Cordova ist ein Framework zur Entwicklung von mobilen Applikationen in HTML 5, JavaScript und CSS. Durch die Verwendung von Cordova entstehen sogenannte Hybrid-Applikationen, da die Benutzeroberfläche über Web-Ansichten und nicht über die Geräte-nativen Benutzeroberflächen-Frameworks gerendert wird. Mithilfe des zweiten Befehls wird das Ionic Projekt erzeugt. Ionic bietet dabei 3 Konfigurationsmöglichkeiten: *blank*, *tabs* oder *sidemenu*. Bei blank wird eine leere Applikation erstellt, tabs erzeugt eine kleine Applikation mit 3 Tabs und mithilfe von sidemenu kann ein neues Projekt implementiert werden, das bereits ein Menü mit mehreren Menüpunkten enthält. Dies ist vor allem zu Beginn sehr hilfreich, um sich mit Ionic vertraut zu machen. Die Befehle ab Zeile 4 fügen dem Projekt die gewünschte Plattform hinzu und wandeln das Ionic-Projekt in eine Smartphone-App um und ermöglichen das Emulieren der App auf dem Computer. Zum Testen der App kann mit folgendem Befehl auch ein anderer laufender Emulator verwendet werden:

```
1 $ ionic run android
```

CSS Komponenten

Ionic stellt ein großes Sortiment von CSS-Komponenten zur Verfügung. CSS-Komponenten sind vordefinierte Benutzeroberflächen-Elemente wie Buttons, Formulare oder Listen. Alle diese Kom-

ponenten sind übersichtlich aufgelistet und jeweils mit Beispielen beschrieben, um den Einstieg in das Framework zu erleichtern. Außerdem sind vorgefertigte Farbvarianten in Ionic inkludiert, die mithilfe von SASS an eigene Bedürfnisse angepasst werden können, allerdings reichen die Standardfarben meistens für eine einfache Applikation aus. Durch diese Auswahl von Komponenten und Anpassungsmöglichkeiten werden Erweiterungen in zusätzlichen CSS Dateien auf ein Minimum beschränkt und sind normalerweise nur für Sonderwünsche nötig.

2.6 NoSQL-Datenbanken

Der Begriff "NoSQL" hat zwei Bedeutungen, welche zu aller erst klar unterschieden werden müssen. Zum einen ist "NoSQL" der Name einer Datenbank, welche 1998 erschienen ist. Das Besondere an dieser Datenbank ist, dass sie für den Zugriff auf ihre Daten und ihr Schema komplett auf eine SQL-Sprache verzichtet. Seit 2009 hat der Begriff auch eine andere Bedeutung. Bei einem offiziellen Treffen von Datenbanktechnikern ist "NoSQL" erstmals als Überbegriff für Datenbanken, welche sich vom üblichen relationalen Datenbankmodell entfernen, eingeführt worden.

2.6.1 Unterschiede zu SQL

Bei NoSQL-Datenbanken handelt es sich somit um Datenbanken, welche nicht nach einem relationalen Modell konzipiert sind. Ein solches Modell hat zwar den Vorteil, dass alle Daten, also auch administrative Daten der Datenbank, über die Abfragesprache SQL abgefragt werden können und es ist auch die Struktur der Daten und deren Relationen vollständig mathematisch definiert. Andererseits hat ein relationales Modell einige Nachteile, wie dass man bei einem objektorientierten Programm einzelne Datensätze der Datenbank nicht sofort als Objekte verwendet werden können, sondern zuerst umgeschrieben werden müssen. Relationale Datenbanken haben zudem oft das Problem, dass sie beim Arbeiten mit großen Datenmengen sehr langsam werden. Um eine bessere Performanz in einem System zu erreichen oder um das System logisch übersichtlicher aufbauen zu können, benötigt man somit manchmal auch eine nicht-relationale Datenbank.

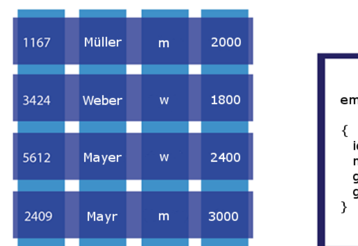
NoSQL-Datenbanken haben in den letzten Jahren ständig mehr an Bedeutung gewonnen und obwohl relationale Datenbanken immer noch den Markt beherrschen, sind NoSQL-Datenbanken aufgrund ihrer unterschiedlichen Architekturen und Anwendungsgebiete ebenfalls unvermindert am Markt vertreten.

2.6.2 Ansätze für NoSQL-Datenbanken

Für die Umsetzung einer NoSQL-Datenbank gibt es mehrere Ansätze zwischen denen man sich entscheiden kann. In diesem Kapitel werden die wichtigsten dieser Ansätze erklärt. [DOL16]

Dokumentenbasierte Datenbank

Bei solchen Datenbanken werden Daten in einer Liste von Dokumenten beziehungsweise Dateien gespeichert. Je nach Datenbank kann es sich bei diesen Dokumenten um unterschiedliche Arten von Daten handeln. Es können einerseits einfache “binary large objects” (BLOBs) wie zum Beispiel Video-Dateien sein, oder Text-Dateien mit vorgegebener Struktur. Bei dieser vorgegebenen Struktur kann es sich beispielsweise um ein JSON-Format oder ein XML-Format handeln. Die vorgesehene Struktur von Dokumenten definiert letztendlich die Datenbank. Bei der nebenstehenden Abbildung sieht man, wie eine relationale Tabelle in einer dokumentenbasierten Datenbank gleich zu setzen ist mit einer Liste von Dokumenten. In diesem Beispiel wird jeder Datensatz einer Tabelle zur Speicherung von Mitarbeitern einer Firma in ein Dokument mit JSON-Format umgewandelt



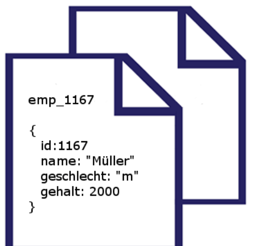


Abbildung 2.2: Vergleich einer relationalen und Dokumentenbasierten Datenbank [Cou11]

Graphen-Datenbank

Diese Datenbanken werden verwendet, wenn man mit stark vernetzten Daten arbeitet. Sie bestehen aus einer Reihe von Knoten, welche mittels Kanten wie in einem Netzwerk-Modell miteinander verbunden sind. Je nach Datenbank können sowohl Knoten als auch Kanten Daten in Form von sogenannten “Properties” speichern. Der große Vorteil bei Graphen-Datenbanken sind ihre auf Graphen spezialisierten Algorithmen für Datenbankabfragen.

Key-Value-Datenbank

Diese Datenbanken zeichnen sich vor allem durch ihren einfachen Aufbau aus und gehören zu den ältesten NoSQL-Datenbanken. Hierbei werden Daten in “Werten” gespeichert, welche immer einem Schlüssel zugeordnet sind. Die Schlüssel sind eindeutig und können in Namensräume oder andere Datenbanken unterteilt werden. Werte können abhängig von der Datenbank auch Listen oder Sets enthalten. Der große Nachteil von Key-Value-Datenbanken ist, dass keine komplexen Datenstrukturen abgebildet werden können. Sie eignen sich hingegen besonders für kleinere Systeme da der Datenzugriff aufgrund seiner Einfachheit sehr schnell ist.

Objekt-Datenbank

Diese Datenbanken speichern Daten direkt als Objekte. Das erleichtert das einlesen der Daten in ein objektorientiertes Programm, denn die Daten können sofort als Objekte instanziert werden und müssen vorher nicht erst umgeschrieben werden. Vor allem werden bei Objekt-Datenbanken keine komplexen Datenbank-Abfragen über mehrere Tabellen benötigt, da weitere Daten direkt über die Referenzen der Objekte geladen werden können. Das erhöht wiederum die Verstehbarkeit der Programme. Größter Nachteil von Objekt-Datenbanken ist die Dauer der einzelnen Datenbankoperationen. Diese ist viel länger als bei relationalen Datenbanken. Grund dafür sind die besonderen Eigenschaften von Objekten. Beispielsweise müssen bei Objekten sowohl Vererbungsbeziehungen als auch die Assoziationen zu anderen Objekten gespeichert und beim Schreiben von Daten überprüft werden.

Spaltenorientierte Datenbank

Während man bei relationalen Datenbanken immer einen Datensatz nach dem anderen speichert, wird bei spaltenorientierten Datenbanken immer jede Spalte einzeln abgespeichert. Mehrere Spalten, die logisch zusammen gehören, werden zudem zu einer "Column Family" zusammengeschlossen, was im relationalen Schema einer Tabelle entsprechen würde. Aufgrund der geänderten Speicherform der Daten ändern sich Geschwindigkeiten von Datenbankoperationen. Zum einen wird das Lesen von einzelnen Spalten beschleunigt, da man nicht mehr alle Datensätze traversieren muss sondern nur noch die konkrete Spalte. Andererseits müssen beim Schreiben von mehrspaltigen Datensätzen alle Spalten einzeln traversiert werden.

2.6.3 CouchDB

Bei CouchDB handelt es sich um ein dokumentenbasiertes Datenbankmanagementsystem welches seit 2005 von der Apache Software Foundation als freie Software entwickelt wird. Freie Software bedeutet, dass die Software und auch der Quellcode, auf dem sie basiert, unbeschränkt verwendet werden darf. [Cou16]

Vorteile:

- Ein simples aber trotzdem skalierbares Datenbankmodell mithilfe des dokumentenorientierten Ansatzes.
- Ein leistungsfähiges Datenbankmanagementsystem im Hintergrund.
- Einfacher Zugriff auf die Daten über zwei verschiedene Web-Schnittstellen.
- Sowohl auf Linux- und Unix-Systemen, als auch auf Mac OS X und Windows installierbar.

Speicherung der Daten

CouchDB speichert Daten in Form von Dokumenten. Diese Dokumente sind selbst in einer JSON-Struktur aufgebaut, deren Form und Größe beliebig gewählt werden kann. In einer Datenbank wird jedes Dokument eindeutig durch eine Identifikationsnummer und eine Sequenznummer identifiziert. Diese beiden Nummern werden automatisch als JSON-Felder mit dem Namen "_id" und "_rev" zum Dokument hinzugefügt. Wenn ein Dokument neu erstellt wird, wird eine eindeutige Identifikationsnummer generiert. Wird ein Dokument bearbeitet bekommt es eine neue Sequenznummer zugewiesen. Auf diese Weise können Änderungen in der Datenbank nachvollzogen werden. [Cou12]

Es gibt noch weitere besondere JSON-Felder, welche von CouchDB verwendet werden um die Dokumente besser zu verwalten:

- **_attachments**: Dieses Feld kann verwendet werden, um andere Dateien wie zum Beispiel Bilder an ein Dokument anzuhängen. Anhänge können bei CouchDB entweder direkt im Dokument gespeichert werden, oder er kann über eine URL auf die Datei referenziert werden.
- **_deleted**: Wird ein Dokument aus der Datenbank gelöscht, wird lediglich dieses Feld zum Dokument hinzugefügt. Um ein Dokument endgültig aus dem Speicher zu entfernen muss der Befehl "purge" verwendet werden. In Futon, der Browserbasierten Web-Schnittstelle von CouchDB, gibt es auch die Möglichkeit alle "gelöschten" Daten in einer Datenbank auf einmal aus dem Speicher zu entfernen.
- **_revisions** und **_revs_info**: Diese beiden Felder geben Auskunft über den Bearbeitungsverlauf eines Dokuments.
- **_conflicts** und **_deleted_conflicts**: Sollte es mehrere Versionen eines Dokuments geben die miteinander in Konflikt stehen, werden diese Konflikte hier vermerkt. Dies kann passieren wenn man eine verteilte Version von CouchDB verwendet.

Zudem gibt es auch noch standardmäßig zwei Datenbanken, welche für administrative Tätigkeiten zuständig sind. Diese haben folgende Namen:

- **_replicator**: CouchDB kann ganze Datenbanken kopieren und an eine bestimmte URL senden. Um eine Replizierung einzuleiten, muss lediglich in dieser Datenbank ein Dokument mit bestimmten Feldern erzeugt werden, die genau definieren welche Datenbank wo hin kopiert werden soll. CouchDB kümmert sich dann automatisch um den Rest.
- **_users**: CouchDB hat auch eine eigene Benutzerverwaltung. Benutzer können hier mit Benutzername, Passwort und einer Liste von Rollen definiert werden.

Zugriff auf die Daten

Views: Da in CouchDB Daten nur in einer einzigen Liste von Dokumenten gespeichert werden, verwendet CouchDB Views, damit Daten trotzdem strukturiert eingesehen werden können. Views sind ebenfalls Dokumente, werden aber gesondert als sogenannte "design-documents" mit einem eindeutigen Namen und eigenem Zugriffspfad gespeichert.
Views speichern eine einzige Methode, welche ein Dokument als Parameter bekommt. In der Methode wird darauf hin entschieden ob und wie das Dokument zurückgegeben werden soll. Diese Methode kann zum Beispiel anhand der Felder, die ein Dokument enthält, entscheiden ob es dieses zurück gibt. Oder die Methode sucht nach einem extra Feld welches den Typ des Dokuments als Wert speichert. Zurückgegeben wird ein Dokument mit einem Schlüssel und einer Liste von Werten. Der Schlüssel kann zudem auch verwendet werden um die Dokumente schon beim Aufruf der View zu sortieren und zu filtern.
Wenn eine View zum ersten mal aufgerufen wird, iteriert diese über alle Dokumente und wendet für jedes Dokument die gespeicherte Methode an. Um Zeit und Rechenleistung zu sparen, wird sofort für alle Dokumente, welche zurückgegeben wurden, ein Verweis gespeichert. Bei späteren Aufrufen der View müssen somit nur mehr die Dokumente, bei denen der Verweis gespeichert wurde, durchlaufen werden und nicht mehr alle.

REST-API: Eine Möglichkeit um auf die Daten von CouchDB zuzugreifen ist über eine RESTful-API. Bei REST handelt es sich um eine standardisierte Web-Schnittstelle, welche den Zugriff auf die Daten der CouchDB über den Aufruf einer URL regelt. Daten können hierbei nicht nur eingesehen, sondern auch bearbeitet, gelöscht und hinzugefügt, werden. Bei CouchDB können mithilfe dieser Schnittstelle alle Dokumente und die Datenbanken selbst verwaltet werden. Die Rest-API ist vor allem für die Kommunikation zwischen zwei Maschinen gedacht, da nur über die URL definiert werden kann, worauf zugegriffen werden soll. Die Antwort im JSON-Format enthält die gewünschten Daten oder gibt Auskunft über den Erfolg der Anfrage. Es wird somit komplett auf eine übersichtliche Präsentation der Daten und auf eine verbindungsorientierte Interaktion mit dem Benutzer verzichtet.
Um auf den Server zugreifen zu können, belegt CouchDB den Port 5984. Diesen Port hat CouchDB offiziell von der IANA, der Internet Assigned Numers Authority, zugewiesen bekommen.

Um die Schnittstelle zu nutzen, kann man auch den Kommandozeilenbefehl “curl” verwenden. Dieser Befehl wird dazu verwendet, Daten über eine bestimmte URL herunter- oder hochzuladen. In unserem Fall werden die Daten über das HTTP-Protokoll übertragen. Je nachdem, welche Operation mit “curl” ausgeführt werden soll, verwendet man eine andere zusätzliche Option:

- **PUT**: Erstellt eine neue Datenbank, View oder ein neues Dokument.
- **POST**: Dieser Befehl wird zum Hochladen von Daten verwendet. Damit kann man ebenfalls Datenbanken, Views oder Dokumente erstellen, verschiedene Werte von diesen setzen oder administrative Prozesse starten. Wenn man ein existierendes Dokument ändern möchte, muss zusätzlich auch die Revisionsnummer des Dokuments eingegeben werden.
- **GET**: Holt sich eine bestimmte Datenbank-Information.
- **DELETE**: Löscht ein bestimmtes Dokument, View oder Datenbank.

Um beispielsweise eine Datenbank namens “demo” zu erstellen muss folgender Befehl eingegeben werden:

```
curl -X PUT http://<CouchDB-IP>:5984/demo
```

Listing 2.6: Ein curl-Aufruf mit dem man in CouchDB eine Datenbank erstellt

Wenn die Datenbank erfolgreich erstellt wurde, bekommt man eine JSON-Struktur mit dem Inhalt “ök”:true” zurück. Aber wenn die Datenbank nicht erstellt werden konnte, weil sie beispielsweise bereits existiert, wird eine Fehlermeldung in der JSON-Struktur zurückgegeben.
Will man ein Dokument in der Datenbank erstellen, muss man eine eigene JSON-Struktur zu CouchDB senden, das den Inhalt des Dokuments beschreibt. Die JSON-Struktur muss man dazu mit der Option “-d” kennzeichnen. Außerdem kann man mit der Option “-H” den HTTP-Header umändern, um sicher zu gehen, dass man eine JSON-Struktur bei dem Aufruf zurück bekommt.

```
curl -H 'Content-Type: application/json' \
     -X POST http://127.0.0.1:5984/demo \
     -d '{"company": "Example, Inc."}'
```

Listing 2.7: Ein curl-Aufruf mit dem man in einer Datenbank von CouchDB ein Dokument erstellt

Wenn das Dokument erfolgreich erstellt wurde, wird dessen ID und Revisionsnummer zurückgegeben.

Futon: Bei Futon handelt es sich um eine Web-Schnittstelle über die man CouchDB verwalten kann. Futon wird grundsätzlich über den Browser aufgerufen und bietet für den Zugriff auf die Daten der CouchDB eine grafische Benutzeroberfläche. Mit Futon kann man alle Dokumente aller Datenbanken einsehen und man kann auch entscheiden ob man diese über eine View filtern will. Weiters kann man die Konfigurationseinstellungen der CouchDB ändern und ganze Datenbanken schnell und einfach replizieren. In der unten stehenden Abbildung sieht man beispielsweise die Oberfläche von Futon in der die Dokumente einer Datenbank angezeigt werden.

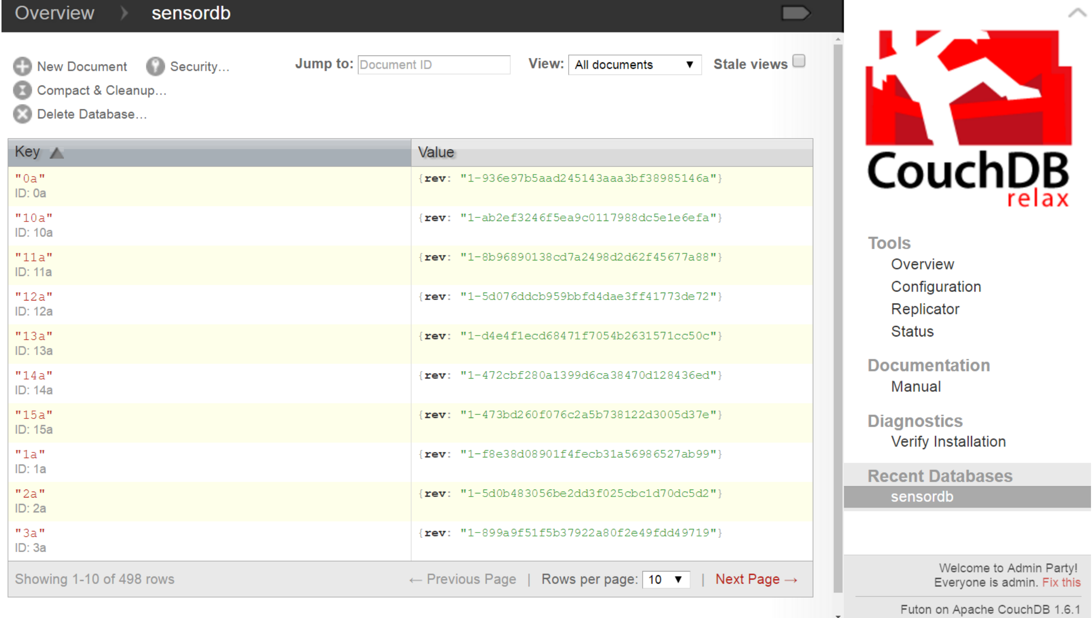


Abbildung 2.3: Futon ist die Webbasierte Schnittstelle zu CouchDB

Ektorp: Dabei handelt es sich um eine Java-Library, welche auf der Java Persistence API basiert und den Zugriff eines Java-Programms zur CouchDB regelt. Ektorp verwendet für den Zugriff auf CouchDB zudem die oben beschriebene REST-API. Nähere Informationen gibt es im Kapitel **2.8 Java Persistence API**.

2.7 Maven

Dabei handelt es sich um ein Software- und Projekt-Verwaltungswerkzeug, welches die Entwicklung von Java-Programmen erleichtern soll. Konkret wird es dazu verwendet um automatisch Bibliotheken zu einem Java-Programm hinzufügen zu können. Es entlastet Entwickler, da man lediglich durch die Angabe einer Konfigurationsdatei alle benötigten Dateien über das Internet beziehen und zu seinem Programm hinzufügen kann. [AMP16]

2.7.1 Konfigurationsdatei

Diese Datei ist der zentrale Punkt eines jeden Maven-Projekts und hat standartmäßig den Namen pom.xml". In ihr werden alle Bibliotheken, welche im Projekt verwendet werden, und alle Konfigurationsdaten des Projekts im XML-Format aufgelistet. Wichtig ist hierbei, dass alle Bibliotheken, welche Maven verwendet, in einem Internet-Repository - also einer Art öffentlichen Datenbank - gespeichert sind. Das hat zur Folge, dass Maven nur Bibliotheken verwalten kann, welche offiziell Teil des Maven-Repositories sind. Diese Zentrale Speicherung hat aber wiederum den Vorteil, dass man auf die einzelnen Bibliotheken schnell und ohne großen Suchaufwand zugreifen kann.
In der Konfigurationsdatei gibt es zum einen Einträge, die das Projekt beschreiben zu dem die Datei gehört und welche Maven-Version verwendet wird. Im zweiten Teil werden alle verwendeten Bibliotheken als sogenannte "Dependencies" aufgelistet. Diese Dependencies werden mit drei Werten eindeutig im Maven-Repository identifiziert:

- **GroupID**: Es kann sein, dass ein bestimmtes System auf mehrere Bibliotheken aufgeteilt wird. Wenn man zum Beispiel nur einen HTTP-Client erstellen möchte, will man nicht das ganze Server-Modell einer HTTP-Anfrage ebenfalls zum Projekt hinzuzufügen. Solche aufgeteilten Bibliotheken können mit der GroupID trotzdem gruppiert werden, damit erkennbar ist, dass sie zum selben System gehören.
- **ArtifactID**: Dieser Wert identifiziert eine konkrete Bibliothek im Repository.
- **Version**: Gibt an welche Version der Bibliothek verwendet werden soll.

2.7.2 Lebenszyklus

Mit Maven kann man aber nicht nur Bibliotheken zu einem Projekt hinzufügen und entfernen. Es ist auch für das Kompilieren des Projekts zuständig. Es hält sich hierbei an den Standart-Lebenszyklus eines Software-Projekts, kann aber beliebig angepasst werden. Man braucht nämlich für unterschiedliche Abschnitte des Lebenszyklusses unterschiedliche Versionen des kompilierten Projekts. Beispielsweise werden Test-Klassen nur in der Testphase verwendet, nicht aber wenn das fertige Produkt kompiliert wird. Diesen Lebenszyklus kann man konfigurieren, indem man die "pom.xml" ändert. Man kann im Grunde einstellen Maven-Plugins bei welcher Phase des Lebenszyklusses gestartet werden.

2.7.3 M2Eclipse

Dies ist eine Erweiterung der Entwicklungsumgebung Eclipse und wird verwendet um Maven direkt in diese integrieren zu können. Der Grundgedanke hierbei ist, dass man in Eclipse ein Maven-Projekt erzeugen und die Funktionen von Maven möglichst uneingeschränkt nutzen kann. Dafür muss Eclipse zuerst wissen wie ein Maven-Projekt aufgebaut ist. Zudem muss es eine Möglichkeit bereitstellen, das Projekt zu konfigurieren. Schließlich ist es auch dafür zuständig, die einzelnen Maven-Phasen wie zum Beispiel das Beziehen der Bibliotheken vom Repository durchzuführen.

In Folgender Liste werden die wichtigsten Funktionen von M2Eclipse Beschrieben:

Die Erstellung eines Maven-Projekts:

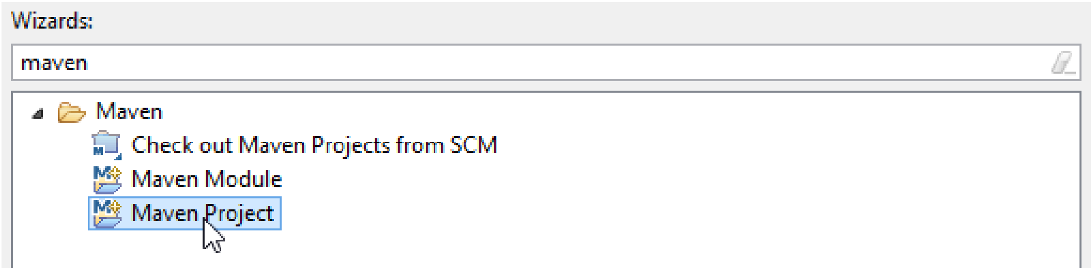


Abbildung 2.4: Auswahl des Wizards zur erstellung eines Maven-Projekts

M2Eclipse stellt einen Wizard zur Verfügung mit dem man direkt ein Maven-Projekt erzeugen kann. Bei diesem Wizard kann man angeben, ob man eine existierende "pom.xml"-Datei verwenden will, oder ob eine neue erstellt werden soll. Weiters ist auszuwählen, welchen "Archetype" man für das Projekt verwenden möchte. Ein Archetype stellt im Grunde eine Vorgehensweise zur Verfügung, nach der das Maven-Projekt aufgebaut wird. Das hat den Vorteil, dass man auch den Projektaufbau an ein bestimmtes Projekt anpassen kann.

Oberfläche für die Konfiguration der pom.xml:
Um den Entwicklern das Konfigurieren der pom.xml zu erleichtern hat man in Eclipse mehrere eigene Ansichten für diese Dateien erstellt. Dabei ist es dem Entwickler überlassen welche der Ansichten er verwenden möchte. Hierbei gibt es zum einen Ansichten, mit denen man direkt den XML-Text der Datei ändern kann und zum anderen gibt es Ansichten welche eine grafische Benutzeroberfläche bieten. Beim Zweiteren ist zudem die Funktion integriert, dass man direkt im Maven-Repository nach Bibliotheken suchen und als Dependency zur Konfigurationsdatei hinzufügen kann (siehe Abbildung). Verschiedene Metadaten des Projekts (wie Name, Beschreibung usw.) können ebenfalls sowohl grafisch als auch im XML-Format eingesehen und angepasst werden.

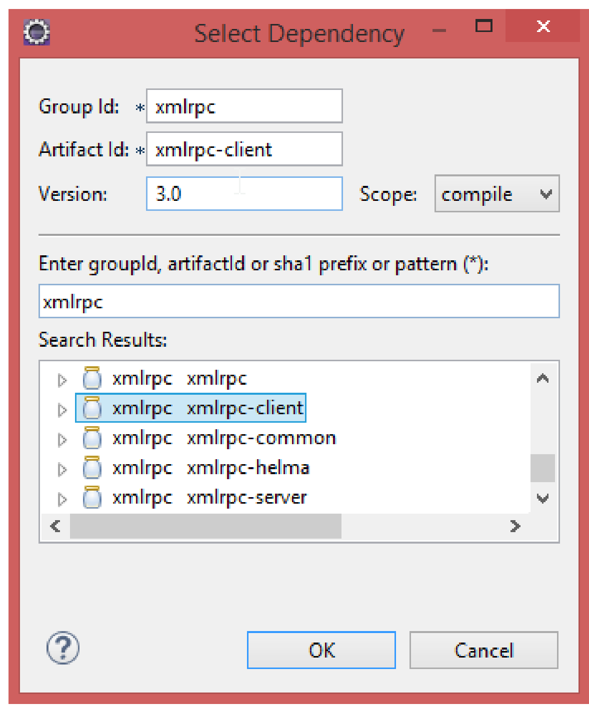


Abbildung 2.5: Suche nach einer Maven Bibliothek

Durchführen der einzelnen Maven-Phasen:

Bei Maven gibt es Phasen, welche jeweils für einen anderen Teil des Lebenszyklusses eines Projekt zuständig sind und somit unterschiedliche Prozesse starten. Bei jeder Phase können noch unterschiedliche Ziele gesetzt werden mit denen es hierbei möglich ist, genauer zu definieren, wie eine Phase durchgeführt werden soll.
Bei der Phase “install”, werden beispielsweise alle Bibliotheken aus der Konfigurationsdatei ausgelesen und vom Repository ins Projekt geladen. Dann gibt es noch Phasen welche das gesamte Projekt mit oder ohne Testklassen kompilieren und das Programm anschließend ausführen.

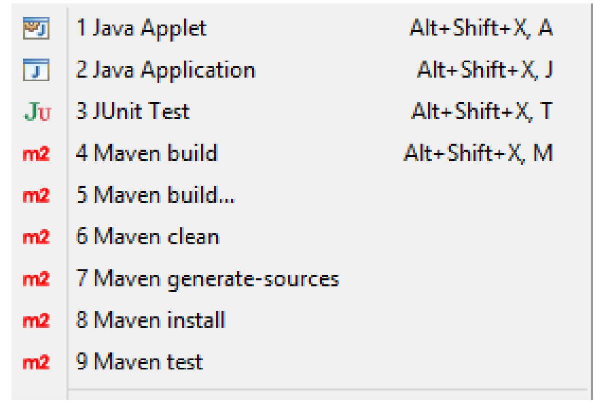


Abbildung 2.6: Liste der in Eclipse standartmäßig verfügbaren Buildprozesse

2.8 Java Persistence API

Die Java Persistence API oder JPA wird verwendet um eine einheitliche Schnittstelle für die Datenbankkommunikation zu schaffen. Es wird zwar immer noch für alle unterschiedlichen Datenbanken eine eigene Implementierung dieser Schnittstelle benötigt, aber wie mit der Datenbank vom Java-Programm aus kommuniziert wird über die JPA definiert. Es gibt sie seit 2006 und JPA 2.1 ist seit April 2013 die aktuellste Version.

2.8.1 Aufbau

Will ein Java-Programm auf eine Datenbank zugreifen, so greift sie eigentlich nur auf die ORM-Schicht (bzw. ORM-Layer in der Abbildung **??**) zu. In der ORM-Schicht verwendet das Programm die JPA unabhängig auf welches Datenbanksystem diese zugreift immer gleich. Wie in der Abbildung zu sehen ist, kommuniziert JPA auch nicht direkt mit der Datenbank. Das wird von der Implementierung von JPA geregelt da JPA nur eine Struktur vorgibt. Für ein bestimmtes Datenbanksystem muss es eine eigene Implementierung von JPA geben, damit dieses verwendet werden kann. Der große Vorteil von JPA ist es, dass das Datenbanksystem schneller gewechselt werden kann als üblich, da nur die Implementierung von JPA ausgetauscht werden muss und somit nur minimale Änderungen im Programm nötig werden.

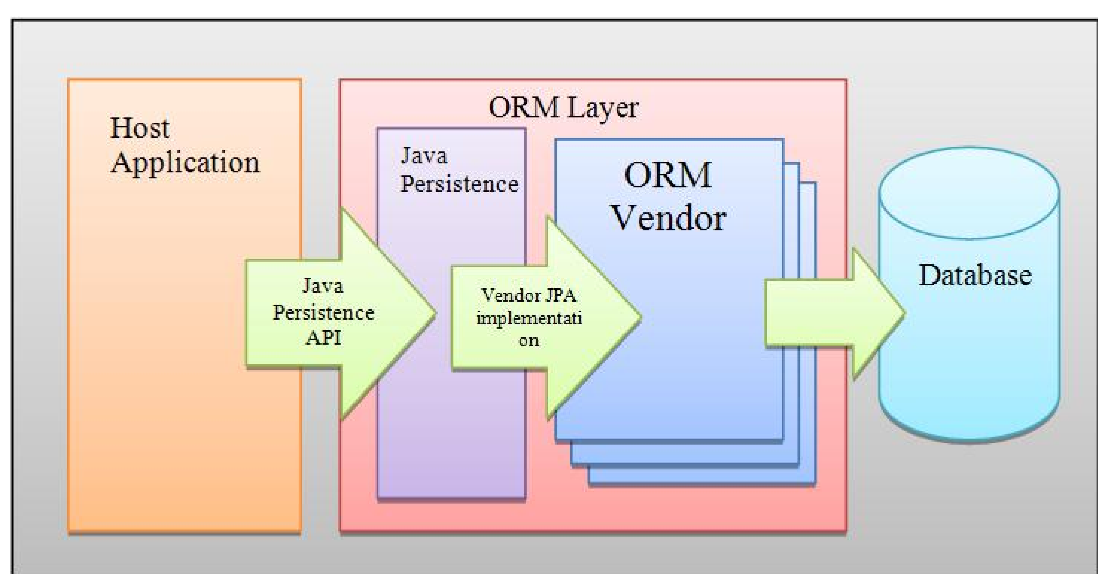


Abbildung 2.7: Der Aufbau der Java Persistence API und wie diese die Verbindung zur Datenbank regelt. [NPA12]

Wird die JPA für ein bestimmtes Datenbanksystem implementiert, soll es möglich sein das Schema einer Datenbank komplett als eine Klassenstruktur abzubilden. JPA definiert ihre Schnittstelle vor allem für relationale Datenbanken. Daher ist die Klassenstruktur im grundsätzlich ein Object-Relational-Mapper. Aber auch für NoSQL-Datenbanksysteme wie CouchDB gibt es eine entsprechende JPA-Implementierung.

2.8.2 Verwendung

Bei JPA gibt es sogenannte Entitäts-Klassen welche auf Datenbankebene normalerweise eine Tabelle abbilden. Während der Laufzeit spricht man bei Entitäten von "Plain Old Java Objects" (POJOs), also primitive Java-Objekte, welche jeweils genau einen Datensatz einer Tabelle speichern. Bei JPA sollen die POJOs nur zum Speichern der Daten verwendet werden und bei der Klassendefinition wird angegeben, wie und welche Daten geladen werden. Zusätzlich kann ein sogenanntes Repository erstellt werden, welches dann eine Liste von Entitäten beziehungsweise POJOs speichert und Methoden für dessen Verwaltung zur Verfügung stellt.
Um zu definieren wie die einzelnen Methoden der Repositories und die Entities mit der Datenbank kommunizieren gibt es bei JPA zwei Möglichkeiten:

Java Persistence Query Language (JPQL)

Dabei handelt es sich um eine eigene Abfrage-Sprache, die entwickelt wurde um die Entitäten einer JPA zu verwalten. Diese Sprache ähnelt zudem stark der Abfrage-Sprache SQL. Die JPA übersetzt dann alle Kommandos, die während der Laufzeit erstellt werden so, dass sie das Datenbanksystem für das sie Implementiert wurde versteht und gibt die jeweiligen Antworten wieder weiter.
Wenn man beispielsweise in einer relationalen Datenbank alle Datensätze einer Tabelle auslesen will, wird zuerst der JPQL-Befehl in einen SQL-Befehl umgeschrieben und zur Datenbank gesendet. Als Ergebnis bekommt man eine Entität mit den jeweiligen POJOs.

JPA Metadaten

Um bei JPA zu definieren wie und welche Daten der Datenbank die Entitäten abbilden, kann man direkt zu den Klassendefinitionen der Entitäten zusätzliche Metadaten hinzufügen. Diese Metadaten, oder Annotationen, ermöglichen es beispielsweise, dass nur durch den Aufruf einer Methode automatisch eine Abfrage zum entsprechenden Datenbanksystem gesendet wird und auch die Antwort sofort wieder in eine Entität oder ein POJO umgewandelt wird. Dadurch ist es nicht mehr nötig einen JPQL-Befehl zu schreiben, sondern der gesamte Ablauf der Datenbankkommunikation wird über einen Methodenaufruf geregelt.
Es gibt in JPA aber nicht nur Annotationen zum Definieren von Entities und wie deren Attribute in der Datenbank gespeichert sind. Es gibt auch Annotationen, die bei einer Klasse definieren, wie diese in das JPA-Modell einzuordnen ist. Denn im JPA-Modell gibt es auch noch andere Bestandteile als Entitäten und Repositories.
Beispielsweise definiert JPA zwei weitere Klassenarten, die direkt in der Datenbank persistiert werden können. Die beiden Annotationen MappedSuperclass und Embedded kann man verwenden um Vererbungsbeziehungen im Datenbankmodell abzubilden, die ja in einer relationalen Datenbank nur angedeutet werden können.
Mit anderen Annotationen kann man wiederum die relationen zwischen den Attributen einzelner Entities bestimmen. Außerdem können Zusatzinformationen für JPQL-Abfragen und Zugriffsmodi der einzelnen Attribute für Entitäten definiert werden. Unter Zugriffsmodus versteht man in diesem Kontext, ob man auf die Attribute der Klasse direkt zugreifen kann weil sie öffentlich sind, oder ob man Getter und Setter verwenden muss.
Dies sind nur einige Beispiele für Annotationen in JPA. Für eine Liste von allen Annotationen sollte man die offizielle Dokumentation von Oracle zu dem Thema lesen.

2.8.3 Ektorp

Ektorp ist eine JPA für CouchDB welche besonders mithilfe von Annotationen eine einfache Schnittstelle zu einer CouchDB-Datenbank schafft. Man kann mit Ektorp nicht nur eine existierende Datenbank abbilden, man auch sofort die Datenbank samt Schema, das man mithilfe von Entitäten bzw. Repositories in der Applikation abbildet, sofort in der Datenbank generieren. Dadurch entfällt nicht nur das Erstellen des Datenbankschemas, sondern man muss auch nicht mehr das Entitäten-Schema an das Datenbankschema anpassen. [Lun11]

Verbindung zur Datenbank

Ektorpt stellt einen eigenen HTTP-Client zur Verfügung, der die Verbindung sowie die Anmeldung bei der Datenbank regelt. Um sich zur CouchDB zu verbinden, wird die REST-Schnittstelle von dieser verwendet. Der Client ist auch SSL-Fähig und es können optional verschiedene Einstellungen wie zum Beispiel das Caching von Dokumenten konfiguriert werden.

Repositories und Entitäten

Zuerst ist es wichtig zu wissen, dass es sich bei CouchDB um eine Dokumentenbasierte Datenbank handelt. Das bedeutet, dass man zum eindeutigen Identifizieren eines Dokuments nur die Datenbank, zu der das Dokument gehört, und dessen Identifikations- und Revisionsnummer angeben muss. Die Datenbank speichert nur eine Liste von Dokumenten und keine Tabellen. Um Dokumente Strukturiert anzuzeigen, werden Views verwendet. Das bedeutet, dass man bei der Klassendefinition von den Entitäten und Repositories nur angeben muss welche Datenbank man abbildet und welche View zum Laden der Dokumente verwendet werden soll.
Beim Laden von Dokumenten als Entitäten gibt es die Möglichkeit, automatisch Views zu generieren, indem man eine entsprechende Methode mit Annotationen in einer Repository-Klasse erstellt. Hierfür ist es aber nötig, dass man die Entitäts-Klasse mit Annotationen definiert, woran man die jeweiligen Dokumente erkennt.

Annotationen

In folgender Liste werden die grundlegenden Annotationen erklärt, die man für eine einfache Ektorp-Applikation benötigt:

- **@JsonProperty**: Diese Annotation schreibt man zu einem Attribut einer Entität und man schreibt auch noch einen String als Parameter dazu. Dieser String entspricht dann einem Schlüssel in einem CouchDB-Dokument. Diese Annotation, wird somit verwendet um den Attributen die richtigen Schlüssel-Wert-Paare zuzuordnen.

- **@JsonIgnore**: Hat man in einer Entität eine Methode oder ein Attribut, das nichts mit der Persistierung der Daten zu tun haben soll, wird diese Annotation verwendet.

- **@DocumentReferences**: Mit dieser Annotation kann eine Referenz zu einer anderen Entität erstellt werden. Mit Parametern kann man definieren auf welches Attribut der anderen Entität genau referenziert wird oder wie die referenzierten Entitäten während der Laufzeit geladen werden sollen. Es ist nicht nötig bei der Annotation zu schreiben auf welche Entität referenziert wird, weil das bereits an der Klasse des Attributs erkennbar ist.

- **@TypeDiscriminator**: Diese Annotation kann entweder zu einem Attribut oder zu einer Klasse geschrieben werden. Sie ist dazu da, um zu ermitteln, ob es sich bei einem Dokument in der Datenbank um eines dieser Entität handelt. Mit dieser Annotation können automatisch Views, für die Entität generiert werden.

Integrierte CRUD-Operationen

Wenn man ein Repository von der Klasse org.ektorp.support.CouchDbRepositorySupport ableitet, sind standartmäßig alle nötigen CRUD-Operationen im Repository integriert. Will man eigene Datenbank-Abfragen erzeugen, so muss man sie wie oben erklärt als Methoden definieren. Um auf die CRUD-Operationen zuzugreifen, müssen zudem auch die entsprechenden Annotationen bei der Entitäts-Klasse, die das Repository verwaltet, gesetzt sein.
CouchDbRepositorySupport beinhaltet folgende Methoden, die die wichtigsten CRUD-Operationen abdecken:

- public void add(Sofa entity);
- public void update(Sofa entity);
- public void remove(Sofa entity);
- public Sofa get(String id);
- public Sofa get(String id, String rev);
- public List<Sofa>getAll();
- public boolean contains(String docId);

2.9 NodeJS

NodeJS ist ein JavaScript-Framework welches speziell dafür entwickelt wurde, um Netzwerkanwendungen zu implementieren. Es werden damit hauptsächlich serverseitige Anwendungen erstellt, welche vor allem über Events gesteuert werden. Obwohl NodeJS in JavaScript geschrieben wird, können Erweiterungen auch in C oder C++ geschrieben sein.

2.9.1 Programmaufbau

JavaScript, welches eigentlich für clientseitige Webanwendungen erstellt wurde, hat den Vorteil, dass man Callback-Methoden zu verschiedensten Events hinzufügen kann. Aufgrund dieses Aufbaus, bei dem eine Programmlogik nur ausgeführt wird, wenn ein bestimmter Event ausgelöst wird, braucht ein NodeJS-Server während der Laufzeit weniger Ressourcen als andere Server mit gleicher Funktionalität. Das hat den Grund, dass bei anderen Programmiersprachen Threads für jede Verbindung erzeugt werden, was speichertechnisch einen Mehraufwand bedeutet. NodeJS wird somit vor allem verwendet, wenn man nur einen kleineren aber dafür umso effizienteren Server benötigt.
Damit das eigentlich clientseitige Javascript als Server funktioniert, stellt NodeJS verschiedene Bibliotheken zur Verfügung mit denen die Events, die Serverseitig passieren, ebenfalls mit Callback-Methoden versehen werden können.
Im unten stehenden Beispiel sieht man eine minimale Konfiguration für einen HTTP-Server der bei einer Anfrage nur "Hello World!" als String zurückgibt. Im ersten Schritt holt man sich hier das http-Modul und erstellt dann mithilfe von diesem einen Server, dem man als Parameter eine Callback-Methode mitgibt. Diese Methode wird dann ausgeführt, wenn von einem Client eine http-Anfrage kommt. In der letzten Anweisung sagt man dem neu erstellten Server, dass dieser am Port 8080 auf Anfragen warten soll.

```
1 var http = require('http');
2 var server = http.createServer(
3    new function (request, response){
4       response.end('Hello World!');
5    }
6 );
7 server.listen(8080);
```

Listing 2.8: Ein einfaches NodeJS-Programm in dem ein http-Server erstellt wird.

2.9.2 Node Package Manager

Der Node Package Manager oder auch "npm" wird verwendet, um die Module beziehungsweise Pakete von NodeJS verwalten zu können. Es handelt sich dabei um ein Kommandozeilen-Programm, das im Hintergrund auf ein Repository mit über 240.000 Paketen zugreift. Mithilfe von npm können einerseits Pakete über einen einzigen Befehl installiert werden. Zum anderen kann man installierte Pakete einsehen und auch deinstallieren.
Der Node Package Manager wird über die Kommanodzeile ausgeführt. Ein Befehl zum installieren eines neuen Moduls würde dann so aussehen:
npm install <Modulname>

Man kann bei der Installation von Paketen auch angeben wo sie installiert werden sollen oder welche Version installiert werden soll.

2.9.3 Serialport

Damit ein NodeJS-Server auf eine serielle Schnittstelle wie zum Beispiel einen USB-Port zugreifen kann, muss man das Modul Serialport installieren. Dieses Modul ist wie das restliche NodeJS-Framework Eventgesteuert. Das bedeutet, dass man einfach eine Callback-Methode zu Serialport hinzufügt, die dann aufgerufen wird sobald ein serieller Port angeschlossen wird. Dann fügt man noch eine Callback-Methode hinzu, die aufgerufen wird, sobald von dem Port Daten empfangen werden. Man kann natürlich auch Daten an den Port senden oder auch alle seriellen Ports auflisten. [Wil16]
Wenn man sich mit einem Port verbindet, gibt es verschiedene Einstellungen, die man ändern kann, um die Kommunikation mit dem Port an seine Wünsche anzupassen. Beispielsweise kann man die Übertragungsrate, die Puffergröße oder die Kodierung der Daten ändern.

2.10 MeshSSN

MeshSSN oder lang Mesh-Sensornetzwerk ist der Titel einer Masterarbeit, bei der es sich fünf Studenten der FH-Hagenberg zum Ziel gemacht haben, Sensoren zu entwickeln, welche Daten wie Luftfeuchtigkeit und Helligkeit messen und über eine USB-Verbindung an einen Computer weitergeben und anzeigen können.
Wie in der Abbildung zu sehen ist, ist diese Masterarbeit in drei große Teile gegliedert:
Der erste Teil besteht aus einer Reihe von selbst gebauten Sensoren, welche die Daten, die sie erheben, periodisch zum nächsten Sensor weitergeben, bis alle Daten beim sogenannten Coordinator landen. Dieser Art der Datenweiterleitung ist genau vom sogenannten Mesh-Protokoll definiert. Die Sensoren kommunizieren außerdem über den Funkstandard IEEE 802.15.4, welcher unter anderem die Übertragungsfrequenz und benötigte Übertragungsleistung definiert.
Ab hier startet dann der zweite Teil der Arbeit, denn der Coordinator, welcher ebenfalls selbstgebaut ist, gibt nun die Daten über eine Mikro-USB-zu-USB-Verbindung an einen Rechner weiter, wo sie von einem NodeJS-Server eingelesen und aufbereitet werden. Dieser Server stellt die Daten über einen Netzwerk-Port im JSON-Format zur Verfügung.
Im dritten und letzten Schritt werden die aufbereiteten Daten noch mit einem C#-Programm dargestellt.

Abbildung 2.8: Der Aufbau der Masterarbeit MeshSSN [FH16]

2.10.1 Das Mesh-Protokoll

Dieses Protokoll, welches die Weiterleitung der Daten über Funk bei mehreren Sensoren definiert, ist speziell dafür konzipiert, möglichst energiesparend und simpel alle erhobenen Daten an einer zentralen Stelle zu sammeln. Das ermöglicht es nämlich, dass die Sensoren nur wenige Hardware-Bestandteile und fast keine Software benötigen um dieses Protokoll durchführen zu können.
Bei diesem Protokoll gibt es Grundsätzlich drei Typen von Knoten:

Endknoten

Dieser Knoten sendet Daten periodisch an den Coordinator oder Router. Er kann selbst keine Daten empfangen, verbraucht aber dadurch weniger Strom, da er nicht ständig den Kanal abhören muss, ob Daten gesendet werden.

Router

Dieser Empfängt zum einen Daten von anderen Knoten und sendet diese dann zusammen mit seinen selbst erhobenen Daten an den Coordinator oder an einen anderen Router wenn er diesen nicht direkt erreichen kann.

Coordinator

Der Coordinator sammelt nun alle Daten und sendet diese über eine serielle Schnittstelle zu dem Rechner, an den er angeschlossen ist.

Energieverwaltung

Das Mesh-Protokoll ermöglicht es, dass die eingesetzten Sensoren möglichst wenig Energie verbrauchen, denn da die Sendezeiten periodisch sind, kann in den Pausen zwischen diesen das System abgeschaltet werden. Bei den Router-Knoten muss der Empfangskanal zwar immer noch offen bleiben, aber man erspart sich trotzdem eine menge Energie, was sich natürlich sehr positiv auf die Laufzeit der Sensoren auswirkt. Die Laufzeit eines MeshSSN-Sensors beträgt somit mehrere Jahre.

2.10.2 Datenverarbeitung

Der NodeJS-Server

Nachdem der Coordinator die Daten über eine serielle Schnittstelle zu einem Rechner gesendet hat, lest sie dort ein NodeJS-Server mithilfe des Serialport-Plugins ein. Als nächstes werden die Daten, welche in einer einzelnen Bit-Kette stehen, umgeschrieben, sodass sie von anderen Programmen verwendet werden können. Der Server belegt außerdem den Port 8080 und stellt dort die Daten für andere Programm im JSON-Format zur Verfügung.

Datenvisualisierung mit C#

Im letzten Schritt werden nun die Daten der Sensoren über ein C#-Programm visualisiert. Genauer gesagt holt sich das Programm die Daten vom NodeJS-Server und stellt diese strukturiert als einen Baum dar.
Um zu erklären wie das Programm zu der Baum-struktur kommt, muss man nochmals bedenken, dass die Daten von den Endknoten entweder direkt ober über eine Anzahl von Routern zum Coordinator gesendet werden. Jetzt werden bei jedem Router die Daten nochmal extra verpackt bevor sie weitergesendet werden. Das hat zur Folge, dass die Daten verschachtelt beim Coordinator ankommen. Durch diese Verschachtelung kann man wiederum für jeden Sensor-Knoten feststellen über welche Router dieser zum Coordinator gekommen ist. Dadurch ergibt sich im Endeffekt eine Baumstruktur wie sie auch in der Abbildung unten zu sehen ist. Voraussetzung dafür, dass die Daten so angezeigt werden können, ist natürlich auch, dass der NodeJS-Server die Daten so verschachtelt weitergibt.

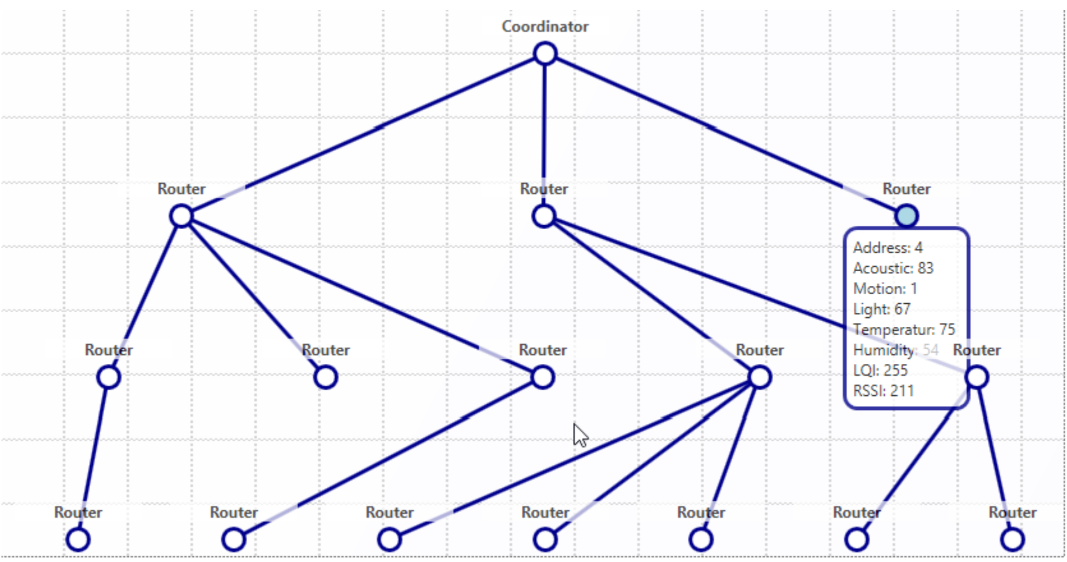


Abbildung 2.9: In dem C#-Programm werden die Sensoren in einer Baumstruktur angezeigt.

2.11 HomeMatic

HomeMatic is ein Hausautomationssystem für private Haushalte. Hausautomationssysteme dienen der automatischen Überwachung von Parametern, wie zum Beispiel der Temperatur, und der Steuerung von Komponenten, zum Beispiel der Heizung im Eigenheim. Dies dient als Erweiterung für private Wohnungen und Wohnhäuser zur Erhöhung des Wohnkomforts. Außerdem können mehrere Wohnsitze überwacht und mehr Sicherheit gewährleistet werden. Auch HomeMatic bietet ein derartiges System mit einem breiten Angebot von Sensoren und Aktoren. Im Sortiment des Hausautomationssystems sind Licht- bzw Beschattungssteuerungen, Heizungselemente, wetterabhängige Steuerungen oder auch sicherheitstechnische Maßnahmen.

2.11.1 CCU2-Zentrale

Die CCU2 ist die Zentrale des HomeMatic-Systems und verwaltet alle vorhandenen Sensoren. Der Benutzer kann Daten der Sensoren über das Webinterface oder eine bereits vorhandene Smartphone-App abfragen beziehungsweise neue Einstellungen festlegen. Die benötigte Software ist dabei im Lieferumfang enthalten.
Auch das Hinzufügen neuer Komponenten wird von der CCU2 übernommen. Es muss lediglich die Anlern-Taste gedrückt werden um die Zentrale in den Anlernmodus zu versetzen und schon können neue Komponenten in das System aufgenommen werden. Statusabfragen und andere wartungstechnische Maßnahmen werden zusätzlich von der CCU2 übernommen. Zusätzlich wird auf der Zentrale ein XML-RPC-Server ausgeführt, welcher über das Netzwerk mit anderen solchen Servern kommuniziert, damit auch externe Anwendungen auf die Zentrale und deren Geräte zugreifen können.
Das gesamte System kommuniziert per Funk mit der Zentrale, weshalb die Komponenten ohne Kabel installiert und sehr einfach an anderen Orten im Haus aufgestellt werden können. Die einzelnen Komponenten ermöglichen auch die Kommunikation untereinander.

Add-Ons

Add-Ons stellen zusätzliche Funktionen für die HomeMatic-Zentrale zur Verfügung, die nicht in der offiziellen Firmware enthalten sind. Sollten bestimmte Add-Ons nicht mehr gebraucht werden, so können diese rückstandslos deinstalliert werden. Mithilfe dieser Add-Ons können beispielsweise neue Weboberflächen, verschiedene Dienste zur Übertragung der Daten in anderen Formaten oder Schnittstellen zur Anbindung an Fremdsysteme installiert werden. Allerdings ist es auch möglich, vollkommen neue Funktionen, wie das Ausdrucken von Programmen über die Weboberfläche, auf die CCU2 zu installieren.
In der Abbildung 2.10 ist die Benutzerschnittstelle von HomeMatic zu erkennen, mit der die Installierten Add-Ons verwaltet und weitere Add-Ons hinzugefügt werden können. Wie in der Abbildung ebenfalls zu erkennen ist, haben die einzelnen Add-Ons die auch Möglichkeit zusätzliche Buttons mit eigener Funktionalität direkt in die Benutzeroberfläche von HomeMatic zu integrieren.

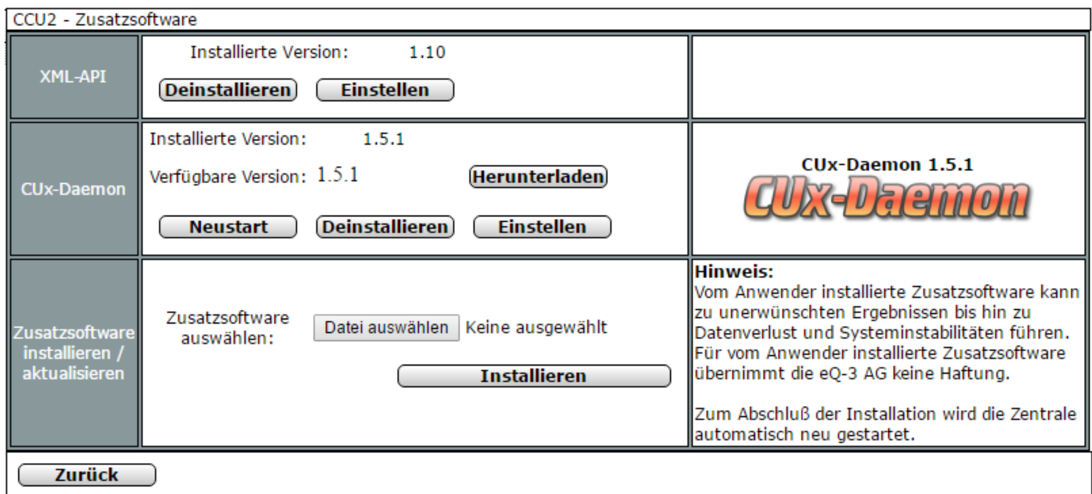


Abbildung 2.10: Ansicht der installierten Add-Ons von Homematic

XML-API Hierbei handelt es sich um ein Add-On für die HomeMatic-Zentrale, welches eine Zusätzliche Schnittstelle für externe, über das Netzwerk verbundene Geräte zur Verfügung stellt. Ähnlich wie bei der XML-RPC-Schnittstelle der Zentrale können alle Daten von Geräten, Programmen oder einige Daten vom System selbst abgefragt werden. Zusätzlich können aber auch Systemvariablen aufgelistet und geändert werden.
Zum Aufrufen des Add-Ons muss lediglich eine bestimmte URL mit entsprechenden Parametern aufgerufen werden und man bekommt eine Antwort von der Zentrale im XML-Format. Die Abfrage der Geräte wird hierbei stark vereinfacht, da der gesamte Aufruf nur über die URL Erfolgt und es somit nicht mehr nötig ist über das XML-RPC-Protokoll komplexe anfragen zu versenden.

CUxD Normalerweise kann die HomeMatic-Zentrale nur mit Sensoren und Aktoren von HomeMatic kommunizieren, aber das Add-On CUxD erweitert die Funktionalität der Zentrale so, dass auch Geräte von vielen anderen Systemen zum System von HomeMatic hinzugefügt werden können. Aufgerufen und konfiguriert wird CUxD über den Browser. [Lan15]
CUxD hat aber noch einige weitere Verwendungsmöglichkeiten:

- Man kann das Gesamte Dateisystem der Zentrale einsehen und erkennt sofort ob die Systemdateien der Zentrale fehlerhaft sind.
- Man kann die Log-Dateien von dem CUxD Add-On und von der Zentrale einsehen.
- Man kann neben den normalen Geräten auch spezielle Geräte zur Zentrale hinzufügen, mit denen man Kommandozeilenbefehle direkt auf der Zentrale ausführen kann.

Programme

HomeMatic ermöglicht es eigene Ablaufsteuerung im Bezug auf bestimmte Ereignisse zu erstellen. Diese Programme funktionieren nach einem bekannten "if-else"-Prinzip. Das heißt, wenn beispielsweise die Außenhelligkeit unter einen bestimmten Wert fällt und eine bestimmte Uhrzeit schon überschritten wurde, sollen die Rollläden im ganzen Haus heruntergefahren werden. Bei Programmen unterscheiden HomeMatic außerdem unter fünf verschiedenen Auslösetypen:

- Auslösen auf Änderung
  - Änderungen sind Zustandsänderungen der Komponenten verglichen mit der letzten Abfrage.
- Auslösen auf Aktualisierung
  - Aktualisierungen sind Benachrichtigungen der Komponenten an die Zentrale, bei denen sich der Messwert nicht zwingendermaßen verändert haben muss.
- Nur Prüfen
  - Dieser Typ kann nicht verwendet werden um Programme auszulösen. Es kann lediglich der aktuelle Zuständ eines Geräts abgefragt werden.
- Auslösen beim Start der Zentrale
- Manuelles Ausführen

Verknüpfungen

Direktverknüpfungen sind die zweite Art Sender mit Aktoren in HomeMatic zu verbinden. Der große Unterschied zu den Programmen ist bei Verknüpfungen, dass diese auch funktionieren, wenn die HomeMatic Zentrale ausgeschaltet ist. Alle benötigten Daten werden innerhalb der Sender und Aktoren gespeichert und ausgeführt. Außerdem ist die Ausführungsgeschwindigkeit höher als bei Programmen, da die Daten nicht über ein weiteres Organ geschickt und verarbeitet werden müssen. Für komplexere Automationen sind Direktverknüpfungen allerdings nicht bestimmt, da diese durch ihren Aufbau zu Einfach und daher eher für Ein- beziehungsweise Ausschalt-Prozesse geeignet sind.

BidCoS-Protokoll

BidCoS bezeichnet das im HomeMatic-System eingesetzte Funkprotokoll zur Steuerung der Komponenten. Die Abkürzung steht für "Bidirectional Communication". Bidirektionale Kommunikation bedeutet, dass alle Komponenten eine Empfangsbestätigung an den Sender senden, sobald dieser die Kommunikation aufgebaut hat. Folge dieser Eigenschaft ist sowohl die erhöhte Sicherheit, als auch verbesserte Funktionalität des gesamten Systems.
Der Standard wurde vom HomeMatic-Produzenten eq-3 entwickelt und wird auch nur von diesem Anbieter verwendet. Das heißt Kunden können sich bei Fragen nur an den Hersteller wenden und müssen auch seine Produkte verwenden. Mithilfe von BidCoS kann die gesamte HomeMatic-Haussteuerung realisiert werden. Dazu zählen sicherheitsrelevante Aspekte, wie beispielsweise Einbruchsschutz, aber auch die Absicherung gegenüber anderer Gefahren, wie zum Beispiel Hausbrände denen man mit einem Rauchmelder vorbeugt. Auch die Klimatisierung des Hauses, die Lichtquellen sowie alle anderen elektrischen Verbrauchen können mit BidCoS angesteuert werden.
Durch Auflagen der Netzagentur ist die Sendeleistung für Funksender auf 25 Milliwatt, sowie die Sendedauer auf 36 Sekunden pro Stunde begrenzt. Diese Beschränkungen bereiten der Hausautomation keine Probleme, Audio- beziehungsweise Bildübertragungen sind dadurch aber nicht möglich. Während viele Funkprotokolle wie Bluetooth oder W-LAN auf einer Frequenz von 2400 MHz senden, arbeitet HomeMatic auf einer Frequenz von 868 MHz, da bei dieser Frequenz die Wellen besser durch Hauswände übertragen werden.[Sch15]

2.11.2 HomeMaticScript

HomeMatic hat auch eine eigene Programmier-Sprache entwickelt, welche besonders an das HomeMatic-System angepasst ist. Von der Syntax lässt sich die Programmiersprache am ehesten mit JavaScript vergleichen, wobei der größte Unterschied ist, dass es in HomeMatic-Script nicht nur Container-Variablen ohne bestimmten Datentyp gibt. Variablen mit primitiven Datentypen können nämlich eindeutig deklariert werden. [Wik15]
In unten stehender Abbildung ist ein Teil des Klassenmodells von HomeMatic-Script (oder HM-Script) zu sehen. HMScript arbeitet nicht direkt mit den Systemdateien oder auf Kommandozeilen-Ebene sondern vielmehr handelt es sich um ein Werkzeug, welches dazu verwendet wird, beispielsweise Geräte (OT_DEVICE), deren Kanäle (OT_CHANNEL) sowie deren Datenpukte (OT_DP) zu verwalten. Die Klassenstruktur beinhaltet aber auch andere Elemente des Hausautomationssystems, wie Favoriten, Räume oder Programme. Die Abbildung zeigt des Weiteren, dass grundsätzlich alles von "OT_OBJECT" abgeleitet wird, welches Methoden hat mit denen man unter anderem ID, Name, Typ und Status eines Objekts abfragen kann. Die beiden letzten Klassen, welche in der Abbildung zu sehen sind, sind "OT_VARDP" für Systemvariablen und "OT_ENUM" für Aufzählungen. Jedes Gerät enthält beispielsweise eine Aufzählung von Kanälen und jeder Kanal enthält eine Aufzählung von Datenpunkten.

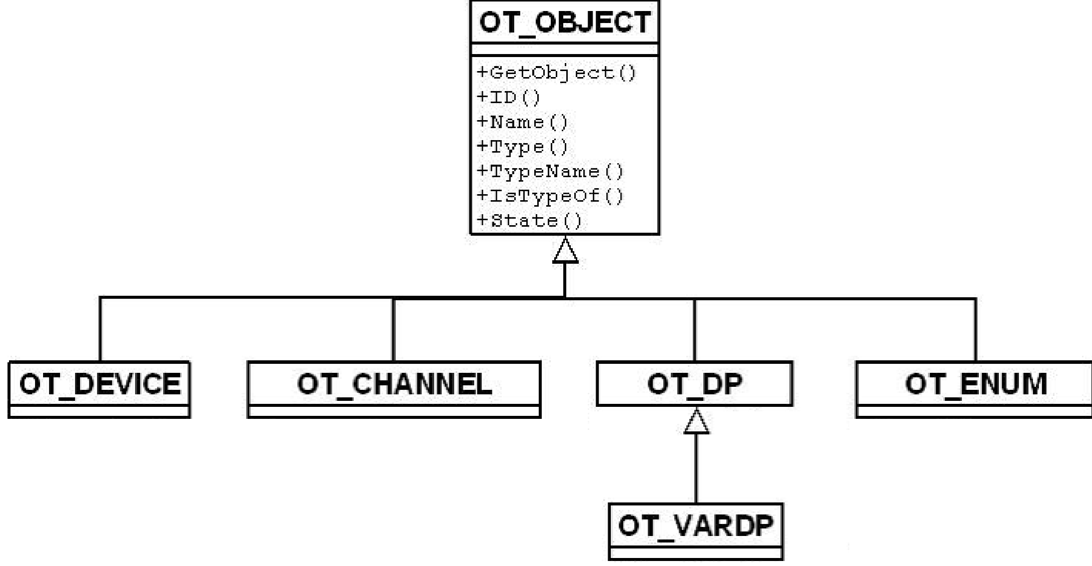


Abbildung 2.11: Das Grundsätzliche Klassenmodell von HomeMatic

Die eigentliche Objektstruktur ist übrigens in der "homematic.regadom"-Datei gespeichert. Diese Datei speichert grundsätzlich alle Daten zum aktuellen Hausautomationssystem, mit allen Geräten, Räumen, Programmen sonstigen Konfigurationen.
Während der laufzeit hat man außerdem Zugriff auf zwei weitere Objekte, namens "system" und "dom". Mit "dom" kann man sich ein Objekt über dessen ID oder Namen holen, oder auch selbst Objekte erzeugen. Mit "system" kann man einerseits auf die Systemzeit zugreifen oder prüfen ob eine Container-Variable ein Objekt enthält oder nicht.
Es gibt bei HMScript viele weitere Objekt-Typen und Methoden, welche aber alle nicht offiziell dokumentiert sind. Es gibt zwar Auflistungen der Objekt-Typen aber leider keine nähere Beschreibung bezüglich ihrer Methoden und Verwendung.

2.12 XMLRPC

XML-RPC (Extensible Markup Language Remote Procedure Call) ist eine Schnittstelle mit verschiedenen Implementierungen, welche der Kommunikation verteilter Systeme über das Internet sorgt. Verteilte Systeme bezeichnen Zusammenschlüsse unabhängiger Computer. Diese unabhängigen Computer verfügen daher nicht über einen gemeinsamen Speicher und kommunizieren miteinander mittels Nachrichten. Für den Benutzer präsentieren sich verteilte Systeme als ein einziges Gesamtsystem. Zur Kommunikation beziehungsweise dem Transport der Daten nutzt XML-RPC das HTTP-Protokoll und XML zur Darstellung der Daten. Ziel der Schnittstelle ist es die Verarbeitung komplexer Datenstrukturen zu ermöglichen, diese allerdings so einfach wie möglich zu gestalten. Dadurch, dass eine große Anzahl von Entwicklern weltweit gemeinsam an dem Projekt arbeitet, gibt es für alle Betriebssysteme, sowie alle häufigen Programmiersprachen Implementierungen der Schnittstelle. Diese Implementierung bietet folgende Basisfunktionalitäten:

- Darstellung der Datentypen
- Übertragung und Empfang von Datenpaketen
- Generierung und Analyse der Datenpakete

2.12.1 Ziele der Schnittstelle

- Firewalls
  - Als Datenformat für das Protokoll wurde XML gewählt, um für Entwickler auf verschiedensten Systemen eine funktionierende und gleichbleibende Basis zu schaffen. Auch Firewalls müssen nur den Datentyp der POST-Aufrufe auf "text/xml" überprüfen um XML-RPC Pakete zu empfangen.
- Klarheit
  - Die Struktur der Aufrufe sollte möglichst eindeutig und erweiterbar sein, sodass Programmierer diese schnell verstehen können.
- Einfachheit
  - Das Protokoll sollte einerseits einfach implementierbar sein und andererseits auch problemlos für andere Umgebungen abgeändert werden. [Win03]

2.12.2 Protokoll

Nachrichten werden in XML-RPC mittels HTTP-POST Requests versendet. Die Sendedaten, als auch die Antwort sind in XML formatiert. Als Parameter können einerseits Skalare, Nummern, Zeichenketten, Daten, andererseits aber auch komplexe Typen wie Listen übertragen werden.

Request Beispiel

```
1 POST /RPC2 HTTP/1.0
2 User-Agent: Frontier/5.1.2 (WinNT)
3 Host: betty.userland.com
4 Content-Type: text/xml
5 Content-length: 181
6
```

```

<?xml version="1.0"?>
<methodCall>
   <methodName>examples.getStateName</methodName>
   <params>
      <param>
         <value><i4>41</i4></value>
      </param>
   </params>
</methodCall>
```

Listing 2.9: Beispiel für einen Request mit XML-RPC

Das Format der ersten Zeile des Headers ist nicht spezifiziert und wird deshalb zum Routen von Anfragen genutzt. Alle anderen Angaben wie der *Host* oder die *Content-Length* müssen spezifiziert werden.
Die Nutzdaten bestehen aus einer einzigen <methodCall>-Struktur, die wiederum ein untergeordnetes Element <methodName>hat. Wenn die aufgerufene Methode bestimmte Parameter benötigt, können diese mit dem <params>-Feld mitgegeben werden.

Response Beispiel

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 158
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:08 GMT
Server: UserLand Frontier/5.1.2-WinNT


<?xml version="1.0"?>
<methodResponse>
   <params>
      <param>
         <value><string>South Dakota</string></value>
      </param>
   </params>
</methodResponse>
```

Listing 2.10: Beispiel für ein Antwort mit XML-RPC

Falls es bei der Anfrage keinen Fehler gegeben gab, wird im Header der Statuscode *200 OK* zurückgegeben. Die Nutzdaten bestehen aus einem einzelnen <methodResponse>Element, gefolgt von einem einzelnen <params>- und einem <value>-Element. Alternativ kann statt dem Parameter auch ein Fehlercode zurückgeliefert werden.

2.12.3 HomeMatic-XMLRPC-Schnittstelle

Diese Schnittstelle wird von HomeMatic verwendet damit externe Server auf die Geräte, die von der Zentrale verwaltet werden, zugreifen können, indem man direkt Methoden von der Zentrale über das Netzwerk aufruft. Zum Aufruf der Methoden wird das Standartisierte XMLRPC-Protokoll verwendet. Mit dieser Schnittstelle kann man aber nicht nur auf die Geräte des HomeMatic-Systems zugreifen. Man kann ebenfalls die Links zwischen den Geräten einsehen oder diverse, nur in der XMLRPC-Schnittstelle sichtbare, Metadaten speichern und einlesen.

[Hom16]

Zugriff auf die Schnittstelle

Zu Beginn der Diplomarbeit gab es drei Separate Ports, unter denen der Zugriff auf die Schnittstelle ermöglicht wurde:

- **Port 2000**: Mit diesem Port werden alle Drahtgebundenen Geräte (genannt HomeMatic-Wired) der HomeMatic-Zentrale angesprochen.
- **Port 2001**: Mit diesem Port kann man alle auf Drahtlosen Geräte, welche an der Zentrale angebunden sind, zugreifen.
- **Port 2002**: Über diesem Port erreicht man nur die Systeminternen Geräte der Zentrale.

Anfang März 2016 hat es aber ein Update für die HomeMatic CCU2 gegeben, bei dem die Schnittstelle für interne Geräte deaktiviert wurde. Somit sind nur noch die Ports 2000 und 2001 verfügbar. HomeMatic stellt außerdem eine JSON-Schnittstelle zur Verfügung, welche aber weitaus weniger dokumentiert ist, als die HomeMatic-Schnittstelle. [Hom13]

Datenmodell

Bezüglich Geräten gibt es bei HomeMatic ein paar logische Unterschiede zwischen dem systeminternen und dem äußeren Modell, welches man beispielsweise bei XMLRPC findet.

Im internen Modell gibt es, wie in der Abbildung **??** zu sehen ist, nämlich Geräte, die jeweils eine Liste von Kanälen speichern. Jeder dieser Kanäle speichert wiederum eine Liste von Datenpunkten. Ein Datenpunkt ist nun ein einzelner Wert, welcher einen Zustand des Geräts speichert und eine genaue Beschreibung dieses Wertes (Einheit, Obergrenzen, Untergrenzen, usw.).

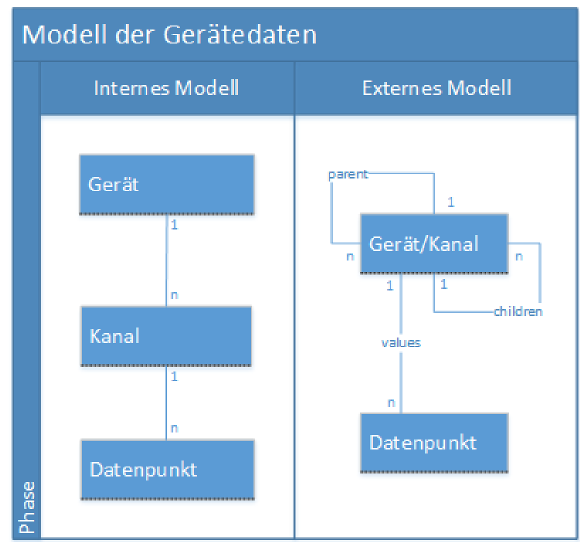


Abbildung 2.12: Das interne und externe Modell von HomeMatic-Geräten

Bei dem externen Modell also der XMLRPC-Schnittstelle ist es nun so, dass kein Unterschied zwischen Geräten und Kanälen gemacht wird, sondern man bekommt beispielsweise bei der Methode “listDevices()” beides zusammen in einer Liste zurück. Jeder Listeneintrag ist hierbei eine HashMap, welche eine Beschreibung des Gerätes bzw. des Kanals darstellt. Der einzige eindeutige Unterschied zwischen Gerät und Kanal ist, dass ein Device beim Schlüssel “CHILDREN” eine Liste seiner Kanäle speichert und ein Kanal beim Schlüssel “PARENT” das Gerät speichert zu dem er gehört. Es ist somit immer ein Schlüssel leer, je nachdem ob es sich um einen Kanal oder ein Gerät handelt. Ein Kanal verweist zusätzlich auf eine Liste von Datenpunkten.

Ein weiterer Unterschied ist, dass die XMLRPC-Schnittstelle zwischen dem Geräte-Zustand und der Gerätebeschreibung eindeutig Unterscheidet. Ein Gerät oder ein Kanal hat einerseits ein Parameter-Set sind alle Zustandsinformationen dieses Gerätes bzw. Kanals als eine HashMap gespeichert. Hierbei gibt es drei mögliche Paramsets:

- **LINK**: In diesem Set werden alle direkten Verknüpfungen zwischen Geräten aufgelistet.
- **MASTER**: Eine Auflistung von administrativen Informationen zum Gerät bzw. Kanal (beispielsweise, ob der Kanal verschlüsselt ist).
- **VALUES**: Listet den Zustand aller Datenpunkte eines Kanals auf.

Parallel dazu kann man auch die Gerätebeschreibung, Parameter-Set-Beschreibung und eine Parameter-Beschreibung über die XMLRPC-Schnittstelle einsehen. Die Gerätebeschreibung ist auch gleichzeitig eine Kanalbeschreibung, da ja hier kein Unterschied zwischen den Beiden gemacht wird. Die Parameter-Set-Beschreibung enthält außerdem eine Liste von Parameter-Beschreibungen. Wenn sie einen Parameter aus dem "VALUES"-Parameter-Set beschreibt, entspricht diese Beschreibung der eines bestimmten Datenpunktes aus dem systeminternen Modell. Die Beschreibung eines "LINK"- oder "MASTER"-Parameters wird im internen Modell direkt beim Gerät oder Kanal gespeichert.

Java-Bibliothek

Da es sich bei XMLRPC um ein standardisiertes Protokoll handelt, wird keine besondere Bibliothek für die HomeMatic-XMLRPC-Schnittstelle benötigt. In Java gestaltet sich der Ablauf eines Programms mit XMLRPC-Aufrufen dann so, dass zuerst die Verbindung einmalig aufgebaut werden muss. Anschließend können verschiedene Methodenaufrufe abgesendet werden, wobei über einen String der Name der Aufzurufenden Methode mitgegeben wird. Da die Methode nur als String gekennzeichnet ist und somit auf der Clientseite nichts über die Methoden-Parameter und den Rückgabewert bekannt ist, müssen auch die nötigen Parameter generisch übergeben werden können. Java Löst dies mit einer Liste von generischen Objekten, welche beim Methodenaufruf ebenfalls übergeben werden. Zu guter Letzt ist auch der Rückgabewert vom Typ "Object" und muss manuell in den Richtigen Wert umgewandelt werden.

2.13 SSH

Bei SSH, oder Secure Shell, handelt es sich um ein Netzwerk-Protokoll welches das Aufrufen eines Terminals von einem entfernten Rechner ermöglicht. SSH bietet zudem eine Verschlüsselungsfunktion um die Verbindung zum entfernten Rechner abzusichern. Genauso wird es aber zum sicheren Übertragen von Dateien verwendet, oder um eine andere Netzwerkverbindung verschlüsselt weiterzuleiten. Aufgrund seiner Vielseitigen Einsatzfähigkeit, wird das Protokoll auf fast allen etablierten Betriebssystemen unterstützt. Auf einigen Linux-Distributionen ist sogar standardmäßig ein SSH-Client installiert, welcher von einem Terminal aus aufgerufen werden kann.

2.13.1 WinSCP

Auf Windows wird eine eigene Anwendung benötigt wenn man einen SSH-Client verwenden will, aber Windows selbst stellt eine Lösung für das Problem gratis zur Verfügung. WinSCP, was ausgeschrieben Windows-Secure-Copy bedeutet, ist eine Anwendung, welche sich vor allem auf die Verwaltung eines entfernten Dateisystems spezialisiert. SSecure Copy" bezieht sich hierbei auf das Protokoll, welches zum sicheren Kopieren von Daten aus einem entfernten Dateisystem verwendet wird. WinSCP beinhaltet aber zusätzlich auch einen SSH-Client mit dem auf ein Terminal des entfernten Rechners zugegriffen werden kann.

2.14 Tool Command Language

Bei "Tool Command Language" bzw. TCL handelt es sich um eine Skript-Sprache dessen Quellcode vom TCL-Interpreter sofort in Byte-Code übersetzt werden kann. Somit muss das Programm nicht kompiliert werden um ausgeführt werden zu können.
TCL-Skript hat eine sehr einfache Syntax. Es gibt grundsätzlich keine Wörter denen ein besonderer Zweck zugeordnet ist, sondern nur bestimmte Zeichen mit besonderer Bedeutung. Beispielsweise wird mit einer Eckigen Klammer ein verschachteltes Kommando gestartet und mit einem Semikolon oder Zeilenende wird ein Kommando vom nächsten getrennt.
Ein Kommando wird immer durch einen String identifiziert. Obwohl es keine Schlüsselwörter gibt, hat man in TCL vorgefertigte Kommandos, die in C oder einer anderen Kompilierten Sprache geschrieben sind und als Bibliothek in einem Programm eingebunden werden können. Mit dem Kommando "proc" ist es auch Möglich eigene Kommando in TCL zu erzeugen.
TCL-Skript ist besonders darauf ausgelegt mit Strings zu arbeiten. Alle Variablen sind grundsätzlich vom Datentyp String und Kommandos werden mit einem String identifiziert. TCL-Skript konzentriert sich vor allem darauf, leistungsfähige Funktionen zur String-Bearbeitung zur Verfügung zu stellen.
In folgendem Code-Ausschnitt ist ein Kommando zu sehen, das den Wert einer Variable "var" auf den Rückgabewert eines zweiten Kommandos setzt. Bei dem zweiten Kommando wird der Wert an der 0-ten Stelle eines Arrays zurückgegeben. Bei dem Array handelt es sich um eine Variable, weshalb es mit einem "$"-Zeichen gekennzeichnet werden muss. Bei der Variable "var" muss man das nicht machen, weil es sich in diesem Kontext nur um einen String-Parameter für das Kommando "set" handelt. Erst innerhalb des Kommandos "set" wird dann die Variable erzeugt auf die mit "$var" zugegriffen werden kann.

```
1 set var [lindex $arr 0]
```

Listing 2.11: Ein kleines Beispiel für die Syntax von TCL-Skript

KAPITEL 3

Benutzung

3.1 Sensordaten-Server – Konfiguration

Der Server besteht aus einer einzigen jar-Datei, in die bereits alle benötigten Bibliotheken integriert sind. Somit kann man den Server starten, ohne ihn erst installieren zu müssen.
Beim erstmaligen Ausführen des Servers wird eine Konfigurationsdatei generiert, mit der man einerseits die URLs, zu denen sich der Server verbindet, angeben und andererseits das Verhalten des Servers beeinflussen kann.

```
1 HomeMaticServer: http://172.16.14.8:2001
2 SensorDataServer: http://localhost:8080
3 CouchDBServer: http://localhost:5984
4 HomeMaticWriteMethod: CreateSysvar
5 CouchDBLogSkips: 60
6 CouchDBMaxEntries: 5000
7 DataLoadingTimeout: 10000
8
```

Abbildung 3.1: Die Konfigurationsdatei des Servers.

In den folgenden Kapiteln werden die einzelnen Einträte beschrieben, mit denen man den Server konfigurieren kann:

3.1.1 HomeMaticServer

Bei diesem Eintrag muss man die URL des XMLRPC-Servers von der HomeMatic-Zentrale eingeben. Wichtig ist hierbei, dass der Port nicht angegeben werden soll, da je nach der Methode, welche zum Schreiben der Sensordaten verwendet wird, ein anderer benötigt wird.

3.1.2 SensorDataServer

Dieser Eintrag gibt die URL des NodeJS-Servers an, mit dem die Sensordaten im JSON-Format zur Verfügung gestellt werden.

3.1.3 CouchDBServer

Mit diesem Eintrag kann man dem Server sagen, wo er den Server für die CouchDB findet, in der die Sensordaten persistiert werden sollen.

3.1.4 HomeMaticWriteMethod

Mit diesem Eintrag kann man beim Server einstellen welche Methode er zum Schreiben der Sensordaten in die HomeMatic-Zentrale verwenden soll. Es gibt hier drei Methoden, die man verwenden kann:

- Bei "Metadata" werden die Sensordaten als Metadaten hinzugefügt. Hierzu wird die XML-RPC-Schnittstelle verwendet.

- Bei "Sysvar" werden die Sensordaten als Systemvariablen in der HomeMatic-Zentrale gespeichert. Dazu muss aber das Add-On "XML-API" auf der Zentrale installiert sein. Jede Systemvariable speichert dann einen bestimmten Wert eines Sensors, wie beispielsweise die aktuelle Helligkeit. Außerdem müssen die Systemvariablen von Hand erstellt werden. Damit sie der Server findet, muss ihnen dazu ein bestimmter Name gegeben werden:
  Der erste Teil lautet "MeshSSNNode_", dann muss die Nummer des Sensors eingegeben werden, die er beim JSON-Server bekommt. Nach der Nummer kommt ein weiterer "_" und als letztes muss man noch angeben, welchen Wert des Sensors die Systemvariable speichern soll.
  Für den letzten Teil der Namensgebung gibt es folgende Möglichkeiten: *"battery"*, *"acoustic"*, *"motion"*, *"light"*, *"temperatur"*, *"humidity"*, *"lqi"*, *"rssi"*
  Ein fertiger Name für eine Systemvariable könne zum Schluss so aussehen: "MeshSSNNode_1_light".
  Für nähere Informationen wie man eine Systemvariable bei der CCU hinzufügt bitte zum Kapitel **?? Systemvariablen erstellen** gehen.

- Um die Methode "CreateSysvar" verwenden zu können, muss auf der Zentrale eine eigene Version des "XML-API"-Add-Ons installiert werden, welches im Laufe der Diplomarbeit erstellt wurde. Dadurch ist es dann nicht mehr nötig die Systemvariablen auf der Zentrale von Hand zu erstellen, wie es bei der Methode "Sysvar" der Fall ist, sondern diese werden automatisch vom Add-On generiert.

3.1.5 CouchDBLogSkips

Der CouchDB-Server muss nicht unbedingt jedes mal, wenn er die Sensordaten einlest, diese in der Datenbank persistieren. Mit diesem Eintrag kann man einstellen, wie oft der Server den Persistierungsschritt auslassen soll bevor er ihn durchführt.

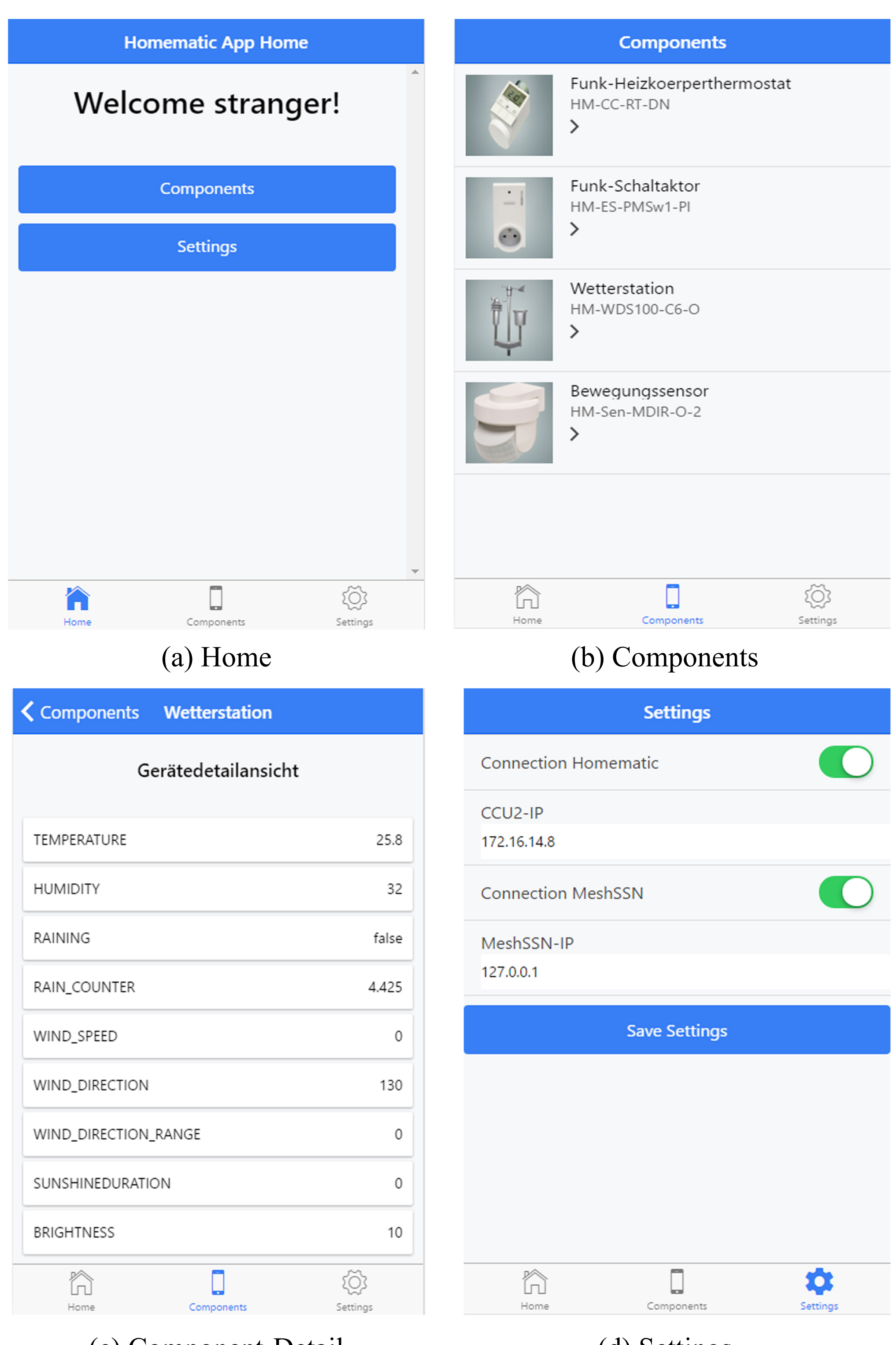
3.1.6 CouchDBMaxEntries

Damit die Datenbank nicht irgendwann zu groß wird, bietet der Server die Möglichkeit, die ältesten Dokumente automatisch zu löschen. Dieser Eintrag gibt hierfür an, wie viele Dokumente maximal in der Datenbank sein dürfen. Nach dem Einfügen der neuen Sensordaten, werden dann genau so viele Dokumente aus der Datenbank gelöscht, wie zu viele drinnen sind. Gibt man hier einen Wert von null oder kleiner an, so wird die Funktion deaktiviert.

3.1.7 DataLoadingTimeout

Mit diesem Eintrag gibt man dem Server vor, wie lange er in Millisekunden warten soll, bevor er den nächsten Zyklus beginnt, in dem er die aktuellen Sensordaten in die Datenbank und die HomeMatic-Zentrale lädt.

3.2 App Oberflächen



(a) Home (b) Components

(c) Component-Detail (d) Settings

Abbildung 3.2: Überblick der App-Oberflächen

Die App ist aufgeteilt in 3 Tabs, wodurch eine übersichtliche Benutzeroberfläche entsteht in der sich auch unerfahrene Benutzer schnell zurecht finden. Zwischen den einzelnen Seiten eines Tabs kann mithilfe des Zurück-Buttons in der blauen Kopfzeile gewechselt werden.

3.2.1 Home-Ansicht

Auf der Startseite der Applikation (a) befindet sich keine Funktionalität, es gibt lediglich zwei Buttons, die neben der Tab-Leiste auf die verschiedenen Menüs innerhalb der Anwendung verlinken.

3.2.2 Übersicht der Komponenten

Im Tab *Components* (b) wird eine Übersicht der verbundenen Geräte aufgelistet. Dazu wird der Gerätetyp, sowie ein kleines Vorschaubild angezeigt.

3.2.3 Detailansicht für Komponenten

Nach Klick auf eine einzelne Komponente kann eine Detailansicht (c) dieser Komponente angezeigt werden, in der genauere Sensordaten für den Benutzer zur Verfügung gestellt werden. Bei jedem Aufruf der Detailansicht werden die angezeigten Daten erneuert und dargestellt.

3.2.4 Konfiguration der Verbindung

Nach dem Start der Smartphone-App kann man mithilfe der vorgegebenen Buttons auf der Startseite oder mithilfe der Tab-Leiste in das Einstellungsmenü (d) navigieren. Dort kann mithilfe des Toggle-Buttons die Verbindungs-IP-Adresse zum Homematic- beziehungsweise MeshSSN System angegeben werden.

Schnittstellen

Der Server hat im Grunde drei Schnittstellen. Eine Schnittstelle ist ein JSON-Server von dem er die Sensordaten einliest und dann gibt es noch zwei Schnittstellen an die diese Sensordaten weitergeleitet werden. Bei der ersten Schnittstelle handelt es sich um eine CouchDB-Datenbank und bei der zweiten um die HomeMatic-Zentrale. Wo diese Schnittstellen zu finden sind, kann dem Server über die Konfigurationsdatei mitgeteilt werden, indem man dort bei den Entsprechenden Feldern eine URL hinzufügt.

4.0.5 Sensoren der MeshSSN einlesen

Bei MeshSSN handelt es sich um ein System von Sensoren, die jeweils kontinuierlich verschiedene Werte, wie beispielsweise Temperatur oder Helligkeit messen. Diese werden dann an einen benachbarten Sensor weitergeleitet und letztendlich an einem Zentralen Sensor gesammelt. Bei diesem Sensor spricht man dann vom Coordinator und er ist über eine seriellen Port an einen Rechner angebunden. Auf der Rechner-Seite gibt es nun zum einlesen der Daten einen NodeJS-Server. Dieser Server stellt die Daten aber nicht nur über eine Netzwerk-Schnittstelle zur Verfügung, sondern wandelt auch das sehr komprimierte Format des Coordinators ein leicht zu verwendendes JSON-Format um. Unser Server kann somit die Sensordaten einfach vom NodeJS-Server holen und weiterverwenden.
Dieser NodeJS-Server wird standardmäßig hinter dem Port 8080 ausgeführt und um die Sensordaten zu bekommen muss man somit eine Anfrage an die URL "<Server-IP>:8080/getSensorData" absenden. Der NodeJS-Server speichert immer die letzte Version der Sensordaten, welche er vom Coordinator bekommen hat um diese bei einem Aufruf weiterleiten zu können. Grund dafür ist, dass der Coordinator selbst keine Anfragen entgegen nimmt, sondern nur periodisch die gesammelten Daten zum seriellen Port weiterleitet. Der NodeJS-Server muss deswegen immer die letzte Version der Sensordaten abspeichern, um auf sofort Anfragen von Clients reagieren zu können, ohne vorher auf den Coordinator warten zu müssen. Der NodeJS-Server wandelt die Daten des Coordinators in folgendes JSON-Format um:
Zuerst werden alle Sensoren in eine Liste geschrieben, die von "0" bis zur Anzahl der Nodes durchnummeriert ist. Jede Nummer bildet einen Schlüssel und der zugehörige Wert ist ein Weiteres JSON-Objekt mit allen gemessenen Werten und einer Beschreibung des Sensors.

Das würde dann also folgendermaßen aussehen:
{"0":{<Sensordaten>},
"1":{<Sensordaten>}}

Folgende Werte werden für die Sensordaten gespeichert:

- **caption**: Mit diesem Wert wird festgehalten, um welche Art von Sensor es sich handelt. Die beiden möglichen Werte sind hierbei "Coordinator" oder "Router".
- **nodetype**: Genau wie bei "caption" wird hier die Art des Sensors gespeichert, aber nicht mit einem Namen, sondern mit einem Zahlenwert. "Coordinator" entspricht hierbei einer "0" und "Router" einer "1", wenn er auch selbst Daten von anderen Knoten weitergeleitet hat oder einer "2", wenn er nur als Randknoten seine eigenen Daten weitergegeben hat.
- **parent**: Dieser Wert sagt welcher Knoten der erste Empfänger dieses Datenpakets war. Wenn beispielsweise der Knoten "3" seine Daten zu Knoten "2" sendet und dieser das Empfangene Datenpaket und sein eigenes zum Coordinator "0" sendet, dann ist der "parent" des Datenpakets von "3" der Knoten "2". Der Parent von Knoten "2" ist direkt der Coordinator "0". Das führt dazu, dass man Schritt für Schritt herausfinden kann über welche Knoten ein Datenpaket zum Coordinator gelangt ist.
  Da der Coordinator seine gesammelten Daten nirgendwo hinschickt, steht bei ihm beim Schlüssel "parent" 65535 als Wert.
- **battery**: Der Wert informiert über den aktuellen Batteriestand des Sensors
- **acoustic**, **motion**, **light**, **temperatur**, **humidity**: Mit diesen Werten werden die verschiedenen gesammelten Sensor-Daten gespeichert.
- **lqi**: Dieser Wert gibt an mit welcher Sende-Qualität das Datenpaket empfangen wurde. Dieser Wert wird vom "parent" des Datenpakets erhoben.
- **rssi**: Dieser Wert gibt Auskunft darüber mit welcher Signalstärke das Datenpaket empfangen wurde. Dieser Wert wird vom "parent" des Datenpakets erhoben.
- **timeout**: Dieser Wert gibt Auskunft darüber, wie viel Zeit seit dem letzten Senden von Daten vergangen ist.

4.0.6 Daten in CouchDB schreiben

Bei CouchDB handelt es sich um ein Datenbanksystem, welches zwei verschiedene Schnittstellen besitzt, mit denen man über das Netzwerk auf das System zugreifen kann. Die erste heißt Futon, und stellt eine grafische Benutzeroberfläche zur Verfügung, die von einem Browser aus aufgerufen werden kann. Bei der zweiten Schnittstelle handelt es sich um einen REST-Service, der eher darauf ausgelegt ist, direkt mit anderen Programmen zu kommunizieren. Auch unser Server verbindet sich über den REST-Service zu unserer Datenbank. Dafür verwenden wir aber eine Bibliothek, und den Kommunikationsablauf nicht selbst implementieren zu müssen. Diese Bibliothek ist außerdem eine Implementierung der "Java Persistence API", dadurch wird es nötig auch innerhalb des Programms ein Modell unserer Datenbank anzulegen. Da es aber bei CouchDB eigentlich kein strenges Datenbankmodell gibt, kann man das Modell vom Programm direkt in die Datenbank übertragen, indem man dort entsprechende Views auf die Dokumente generiert.

Bei unserem Datenmodell gibt es genau zwei Arten von Dokumenten, die erste Art symbolisiert einen bestimmten Sensor. Dieser hat nur einen Verweis auf das letzte Sensordaten-Dokument, das in die Datenbank geschrieben wurde. Das Sensordaten-Dokument ist auch die zweite Dokumentart in unserer Datenbank. Dieses speichert die konkreten Werte eines einzigen Sensors zu einem bestimmten Zeitpunkt.

4.0.7 Daten in HomeMatic schreiben

Bei HomeMatic gibt es nun mehrere Schnittstellen die der Server verwenden kann, da man in der Konfigurationsdatei ja insgesamt drei verschiedene Methoden auswählen kann, wie der Server die Daten auf der HomeMatic-Zentrale speichern soll.

- **Daten als Metadaten speichen**: Für diese Methode wird die XMLRPC-Schnittstelle von HomeMatic verwendet. Mit der XMLRPC-Schnittstelle von HomeMatic können hauptsächlich Geräte und Verlinkungen zwischen Geräten verwaltet werden. Es ist auch Möglich Daten zu schreiben, wobei es aber vom Gerät abhängt, ob es die Änderung von Daten zulässt oder nicht. Neben den normalen Systemdaten können über die XMLRPC-Schnittstelle auch Metadaten auf der Zentrale gespeichert werden. Wenn man die Sensordaten mit dieser Methode auf die Zentrale überträgt hat es leider den Nachteil, dass man die Daten nicht von Innerhalb der Zentrale aus sieht. Auf Metadaten kann man nämlich nur von der XMLRPC-Schnittstelle aus zugreifen.

- **Daten in Systemvariablen speichern**: Die zweite Methode mit der man die Sensordaten auf die HomeMatic-Zentrale übertragen kann, hat den großen Vorteil, dass sie auch von innerhalb der Zentrale verwendet werden kann. Systemvariablen haben den großen Vorteil, dass man mit ihnen auch Programme auslösen kann. Für diese Methode wird das Add-On XML-API benötigt, mit dem man ähnlich wie bei der XMLRPC-Schnittstelle Gerätedaten einsehen kann. Zusätzlich kann man sich aber auch alle im System gespeicherten Programme, Räume, Favoriten oder Systemvariablen holen. Systemvariablen kann man mit XML-API auch ändern.
  Der Server benutzt also dieses Add-On um die aktuellen Sensordaten in Systemvariablen zu speichern, aber da das Add-On keine Variablen erzeugen kann, müssen diese vorher erstellt werden. Bei der Erstellung muss man zudem auf die richtige Namensgebung achten, damit der Server die Variablen auch findet. Mehr Informationen hierzu gibt es im Kapitel **3.1.4 HomeMaticWriteMethod**
  Um eine Systemvariable zu ändern ist ein einfacher URL-Aufruf mit bestimmten Parametern nötig. XML-API entscheidet anhand der Parameter welche Systemvariable auf welchen Wert gesetzt wird.

- **Daten in Systemvariablen speichern und diese automatisch generieren**: Die Schnittstelle der dritten Methode ist im grunde Identisch mit der Schnittstelle der zweiten. Der einzige Unterschied ist hierbei, dass man ein spezielles XML-API Add-On benötigt, dessen Methode zum Ändern einer Systemvariable, diese Auch erzeugen kann, wenn es sie noch nicht gibt.

KAPITEL 5

Implementierung

5.1 Aufbau des HomeMatic-Systems

Im Rahmen unserer Diplomarbeit haben wir verschiedene Aktoren, Sensoren und zwei CCU2 Zentralen von HomeMatic von der Schule zur Verfügung gestellt bekommen. Teil unserer Diplomarbeit war es, diese Geräte zu einem Hausautomationssystem zusammen zu schließen. Grundsätzlich werden alle Aktoren und Sensoren bei der Zentrale eingebunden, damit sie über diese gesteuert werden können. Es ist aber auch möglich Geräte direkt zu verbinden, damit die Zentrale einerseits entlastet wird und die Geräte andererseits nicht von dieser abhängig sind.
Um sich mit der Zentrale zu verbinden, kann man diese über Ethernet an ein lokales Netzwerk anschließen oder sich direkt über einer USB-Schnittstelle mit dieser verbinden. Mehr Informationen zur zweiten Variante gibt es im Kapitel 5.3.1 unter der Überschrift "USB-Verbindung der HomeMatic-Zentrale".

5.1.1 Geräte zur CCU2 hinzufügen

Um Geräte mit der Zentrale zu verbinden, müssen sowohl diese als auch die Zentrale in den Anlernmodus gebracht werden. Wie die einzelnen Geräte in den Anlernmodus gebracht werden, steht in deren Gebrauchsanweisung. Um ein Gerät in den Anlernmodus zu bringen, müssen in den meisten Fällen eine gewisse Abfolge von Knöpfen gedrückt werden.
Die Zentrale bringt man in den Anlernmodus, indem man sich über einen Browser mit ihr verbindet und in der Benutzeroberfläche den Button "Gerät anlernen" drückt. Darauf hin startet sich für 60 Sekunden der Anlernmodus und es öffnet sich im Browser ein neues Fenster in dem weitere Möglichkeiten zum Anlernen von Geräten angezeigt werden. Es können einerseits BidCos-Geräte direkt mit ihrer Seriennummer angelernt werden, ohne dass diese in den Anlernmodus gebracht werden müssen. Andererseits kann man in diesem Fenster auch HomeMatic-Wired- und HomeMatic-IP-Geräte angelernt werden. Bei HomeMatic-IP handelt es sich hierbei um den Nachfolger von HomeMatic-BidCos, da eine Anpassung des Funk-Protokolls durch gestiegene technische Anforderungen nötig wurde.

5.2 App

5.2.1 Implementierung GUI

Die Benutzeroberfläche der Applikation ist in verschiedene Tabs aufgeteilt. Diese Tabs sind in einer eigenen HTML-Seite als *ion-tabs* definiert. In Ionic sind Tabs, wie beinahe alle anderen Komponenten stark konfigurierbar. So kann man beispielsweise aus einer großen Anzahl von Icons beziehungsweise Designs auswählen.

5.2.2 App-Routing

Um zwischen den verschiedenen App-Seiten navigieren zu können, gibt es eine Routingdatei, in der alle Navigationswege der gesamten Applikation definiert sind.

```
config(function($stateProvider, $urlRouterProvider) {
   $stateProvider

      .state('tab.components', {
         url: '/components',
         views: {
            'tab-components': {
               templateUrl: 'templates/tab-components.html',
               controller: 'ComponentsCtrl'
            }
         }
      })
      .state('tab.components-detail', {
         url: '/components/:compId',
         views: {
            'tab-components': {
               templateUrl: 'templates/components-detail.html',
               controller: 'ComponentsDetailCtrl'
            }
         }
      })
   });
   $urlRouterProvider.otherwise('/tab/home');
});
```

Listing 5.1: Ausschnitt App-Routing

In dieser Routingdatei wird für jeder Route ein sogenannter *State* definiert, über den sie unterschieden werden. Außerdem wird die URL definiert, über welche der Zugriff auf die einzelnen Seiten ermöglicht wird. Bei der Detailansicht werden mithilfe der *:compId* die einzelnen Komponenten unterschieden. Darüber hinaus wird den verschiedenen Seiten sowohl der passende Tab, als auch der Pfad und der zugehörige Controller hinzugefügt.

5.2.3 Persistierung der Einstellungen

Um Daten wie die Verbindungs-IP auch bis zum nächsten Öffnen der Applikation zu speichern, kann in Ionic der sogenannte *local Storage* verwendet werden. Local Storage ist ein key-value Modul, welches es ermöglicht, Zeichenketten sehr einfach zu speichern.

```
.controller('SettingsCtrl', function($scope, $stateParams, Settings) {
   $scope.settings = {
      HomematicIP: Settings.get('HomematicIP'),
      MeshIP: Settings.get('MeshIP')
```

```
   };

   $scope.updateSettings = function(){
      Settings.set('HomematicIP', $scope.settings.HomematicIP);
      Settings.set('MeshIP', $scope.settings.MeshIP);
   };
})

.factory('Settings', function() {
   return{
      get: function(name) {
         Value = window.localStorage.getItem(name);

         if(Value === "undefined" || Value === undefined ||
            Value === null || Value === 'null'){
            return ("");
         }else{
            return(Value);
         }
      }

      set: function(name,value){
         return window.localStorage.setItem(name, value);
      }
   };
});
```

Listing 5.2: Persistierung der App-Einstellungen mittels Local Storage

Der Controller im Codebeispiel wird aufgerufen, sobald der Benutzer auf den Einstellungen-Tab wechselt. Zu Beginn werden die bereits gespeicherten Einstellungen geladen. Um auf die gespeicherten Daten zugreifen zu können, muss die Funktion *getItem()* des Local Storages aufgerufen werden. Falls der Benutzer keine Eingabe macht und trotzdem speichert, wird der Wert des Eingabefeldes auf *undefinded* gesetzt, deswegen wird das Feld in diesem Fall leer gemacht. *UpdateSettings* ist ein Click-Listener, der ausgelöst wird, sobald der Speicher-Button im Einstellungsmenü geklickt wird. Mithilfe des Settings-Objekts kann auf die zum Controller zugehörige Factory zugegriffen werden. Factories zählen in Angular zu den sogenannten Providern und dienen dazu, Funktionen in den Controller zu injizieren. Damit können die Abhängigkeiten eines Objekts stark reglementiert werden, weil diese an einem zentralen Ort, wie der Factory, hinterlegt sind. Mithilfe der Local Storage-Funktion *setItem* kann ein neuer Wert gespeichert werden.

5.2.4 Verbindung mit XML-RPC Service

```
    var response = null;
    var resp = [];
    var method = "getValue";
    var url = 'http://'+Settings.get('HomematicIP')+':2001';
    var client = new xmlrpc_client(url);
    var msg = new xmlrpcmsg(method, [ new xmlrpcval(device),new
        xmlrpcval(value)]);
    client.setDebug(2);
    var response = client.send(msg);
```

Listing 5.3: Verbindung zur CCU2 Zentrale mithilfe der jsxmlrpc-Bibliothek

Mithilfe der jsxmlrpc-Bibliothek kann mithilfe der Methode *xmlrpcmsg* ein neuer Aufruf an die HomeMatic-Zentrale erstellt werden. Davor wird der Verbindungsstring erstellt. Mithilfe von *Settings.get()* wird dabei die IP-Adresse der CCU2 aus den Einstellungen abgefragt und in die Variable "url" gespeichert. Der Port 2001 wird beim HomeMatic-System für BidCos Funk-Komponenten verwendet. Danach wird die Nachricht an die HomeMatic-Zentrale gesendet und die Antwort in die Variable *response* geschrieben.
Während der Implementierung der Applikation trat ein Verbindungsproblem auf, da die meisten Browser aus Sicherheitszwecken *Cross-Origin Resource Sharing (CORS)* verbieten, sofern der Server dies bei seiner Antwort nicht extra erlaubt. CORS ist ein Mechanismus, der es ermöglicht Anfragen zwischen unterschiedlichen Domänen zu machen. Allerdings gibt es für Google Chrome ein Add-On um CORS-Anfragen, trotz fehlender Bestätigung des Servers zu erlauben. Auf dem eigentlichen Gerät stellt CORS kein Problem mehr dar, da alle Dateien der App direkt auf dem Smartphone selbst liegen und es deswegen keine Quelle gibt.

5.2.5 Dynamisches Generieren der Detailansicht

Um die Detailansichten der verbundenen Geräte anzeigen zu können, gibt es nur eine einzige HTML-Seite, die für jede Komponente dynamisch generiert wird. Alle Komponenten im HomeMatic-System bestehen aus einer Vielzahl von gespeicherten Daten. Diese Daten umfassen beispielsweise Namen, Typ oder Datenpunkte der Sensoren. Datenpunkte sind jene Messwerte, die jedes Gerät durch die installierten Sensoren misst. Da jedoch keine Möglichkeit besteht, über bestimmte Methodenaufrufe die aktiven Datenpunkte eines Sensors abzufragen, müssen diese aus einer Dokumentation entnommen werden. Dadurch ergibt sich folgende Speicherstruktur:

```
var components = [{
   id: 0,
   name: 'LEQ1513317:4',
   displayName: 'Funk-Heizkoerperthermostat',
   type: 'HM-CC-RT-DN',
   thumbnail: 'img/Funk-Heizkoerperthermostat.jpg',
   dataPoints: {'ACTUAL_TEMPERATURE': '', 'BATTERY_STATE': ''}
},{
   id: 1,
   name: 'LEQ0661195:2',
   displayName: 'Funk-Schaltaktor',
   type: 'HM-ES-PMSw1-Pl',
   thumbnail: 'img/Funk-Schaltaktor.jpg',
   dataPoints: {'ENERGY_COUNTER': '', 'POWER': '', 'CURRENT': '', 'VOLTAGE':
       '', 'FREQUENCY': ''}
}];

return{
   get: function(compId) {
      for (var i = 0; i < components.length; i++) {
         if (components[i].id === parseInt(compId)) {
            for (var key in components[i].dataPoints){
               components[i].dataPoints[key] = getValue(components[i].name,
                   key);
            }
            return components[i];
         }
      }
      return null;
   }
}
```

Im darauf folgenden Programmstück wird die Liste von Komponenten iteriert und für jeden spezifizierten Datenpunkt des gewünschten Sensors die zugehörigen aktuellen Messwerte gespeichert. Über die gespeicherte id wird der vom Benutzer angeklickte Sensor erkannt und seine Daten ausgelesen. Mithilfe der *id* wird auch im Controller die ausgewählte Komponente ausgelesen und an die Anzeige weitergegeben.

```
<div class="card" ng-repeat="(key, value) in component.dataPoints">
   <div class="row">
      <div class="col">{{key}}</div>
      <div class="col text-right">{{value}}</div>
   </div>
</div>
```

Listing 5.4: Dynamisches Generieren der Detailansicht

Mithilfe der *ng-repeat*-Direktive kann die nach Key-Value-System aufgebaute Liste von Datenpunkten durchgelaufen und jeweils in ein neues Element geschrieben werden. Um eine übersichtliche Darstellung der Daten zu erreichen, werden die Daten in sogenannte *Cards* geschrieben.

5.2.6 Struktur

Die Smartphone Applikation ist nach dem Angular MVVM-Modell aufgebaut. Es gibt mehrere *Views*, in denen die Daten angezeigt werden. Diese Views sind HTML-Seiten, zwischen denen mithilfe des App-Routings beziehungsweise der zugewiesenen Tabs gewechselt werden kann. Zur Datenhaltung werden Factories verwendet, die einerseits Sensordaten speichern und andererseits Methoden zur Manipulation dieser Daten bereitstellen. Um in den einzelnen Views Daten anzeigen zu können, beziehungsweise um Funktionalität zu implementieren, ist jeder View ein eigener Controller zugewiesen, der aufgerufen wird, sobald die Seite vom Benutzer ausgewählt wird.

5.3 Server

Der Server ist dafür zuständig die Daten der Sensoren von MeshSSN einzulesen und in eine CouchDB-Datenbank zu schreiben. Außerdem werden sie an die HomeMatic-Zentrale weitergeleitet, damit diese mit den aktuellen Sensordaten arbeiten kann.

5.3.1 Installation

Bei diesem Kapitel geht es um die Installation der drei Schnittstellen des Servers: Einen NodeJS-Server für die Sensordaten von MeshSSN, die CouchDB-Datenbank und der USB-Zugriff auf die HomeMatic-Zentrale.

MeshSSN

Um auf die Sensoren von MeshSSN zugreifen zu können, muss zuerst ein NodeJS-Server installiert werden. Der NodeJS-Server ist für das Einlesen der Daten vom Coordinator des MeshSSN-Netzwerks zuständig und stellt diese im Netzwerk zur Verfügung.
Dazu muss als erstes NodeJS installiert werden. Hierbei ist wichtig, dass man nicht die neueste Version installieren darf, denn der NodeJS-Server, ist inkompatibel mit den neueren Versionen. Entwickelt wurde dieser für Version 0.10, aber auch mit späteren Versionen wie 0.12 kann man ihn noch ausführen.
Nachdem die richtige NodeJS-Version installiert wurde, kann man nun auch das Serialport-Modul installieren. Dieses Modul ermöglicht es dem Programm mit einem seriellen Port zu kommunizieren. Module werden generell über den Paket Manager von NodeJS verwaltet. Vor der Installation des Moduls, muss man noch den NodeJS-Server in ein beliebiges Verzeichnis kopieren. Dann kann man folgenden Befehl ausführen:

```
1 npm install <Verzeichnis-Pfad> serialport@1.7.4
```

Listing 5.5: Kommandozeilenbefehl für die Installation des Serialport-Moduls

Mit dem Verzeichnis-Pfad ist das Verzeichnis gemeint indem sich der NodeJS-Server befindet, da man das Modul direkt dort hinzufügen muss. Man muss hierbei bedenken, dass der Verzeichnispfad, bis zu dem Verzeichnis gehen muss, in dem sich eine Datei mit dem Namen "package.json" befindet. findet npm diese Datei nämlich nicht, so wird die Installation des Moduls abgebrochen. Mit "@1.7.4" sagt man npm welche Version des Moduls installiert werden soll - in unserem Fall die Version 1.7.4.

Nach der Installation des Servers, kann man diesen nun folgendermaßen ausführen:

```
1 node <Verzeichnis-Pfad>/serial.js [COM-Port [http-Port]]
```

Listing 5.6: Kommandozeilenbefehl für die Installation des Serialport-Moduls

Wenn man den Server ausführt, kann man zusätzlich den Namen eines COM-Ports als Kommandozeilen-Argument übergeben. Wenn der Coordinator an einen Rechner angeschlossen wird belegt er automatisch einen COM-Port. Der Name von diesem hängt vom physikalischen Port ab an dem er angeschlossen ist. Mithilfe des Namens ist es dann dem Serialport-Modul möglich mit dem Coordinator zu kommunizieren. Übergibt man keinen Namen als Argument wird automatisch der Port "COM5" verwendet.

Steht hinter dem angegebenen COM-Port kein Coordinator, bekommt man folgende Fehlermeldung:

```
Serial port error: Error: Opening \textbackslash\textbackslash .\textbackslash
    COM5: File not found
INITIATING RECONNECT
RECONNECTING TO COORDINATOR
```

Listing 5.7: Fehlermeldung, wenn man einen unbesetzten COM-Port übergeben hat.

Diese Fehlermeldung wird dann solange wiederholt ausgegeben, bis der COM-Port belegt, oder der Server beendet wird.
Um herauszufinden welchen COM-Port der Coordinator belegt, kann man unter Windows im Geräte-Manager nachsehen. Wie in der Abbildung zu sehen ist, kann man den COM-Port unter "Anschlüsse (COM & LPT)" finden und in den Klammern steht sein Name, mit dem er von Serialport gefunden werden kann.

Kann der Server erfolgreich gestartet werden, so belegt dieser nun den Port 8080 und mit der URL "<Server-IP>:8080/getSensorData" können nun die Aktuellen Sensordaten des MeshSSN-Netzwerkes abgefragt werden.

Wenn man einen COM-Port beim Starten des Servers angibt, kann man zusätzlich auch noch den http-Port angeben, bei dem der Server auf Anfragen horchen soll.
Sollte beim Starten des Servers die Fehlermeldung "Error: listen EADDRINUSE" ausgegeben werden, bedeutet das, dass der verwendete Port bereits von einer anderen Anwendung besetzt wird. Wird kein Port beim Start des Servers angegeben, verwendet dieser seinen Standardport 8080. Oracle-Datenbanken verwenden diesen Port auch standardmäßig für ihre APEX-Schnittstellen. Um diesen Fehler zu beheben, muss man somit entweder alle anderen Anwendungen die diesen Port belegen schließen, oder beim Serverstart einen anderen Port angeben.

CouchDB

Wenn man bei der Installation von CouchDB die Version 1.6.1 verwendet, sollte es grundsätzlich keine Kompabilitätsprobleme geben. CouchDB ist generell sehr kompatibel mit anderen Applikationen, da sie ihre Daten über den REST-Standard zur Verfügung stellt.

USB-Verbindung der HomeMatic-Zentrale

Um sich mit der Zentrale zu verbinden hat man zwei Möglichkeiten. Entweder man schließt sie über ein LAN-Kabel an ein Netzwerk an, oder man verwendet den USB-Port der Zentrale um einen Rechner direkt mit der Zentrale zu verbinden. Will man direkt über USB auf die Zentrale zugreifen, so muss man einen Treiber von HomeMatic installieren, der ein privates Netzwerk simuliert. Um die benötigten Treiber zu installieren, schließt man zuerst die Zentrale per USB an den Rechner an. Dann wird ein neues Laufwerk gemountet, in dem sich ein Setup befindet, das die benötigten Treiber selbstständig installiert.
Die Netzadresse dieses virtuellen Netzwerks ist dann 10.101.82.0/24 und die Zentrale stellt ihre IP-Adresse automatisch auf 10.101.82.51 ein. Der Rechner bekommt die IP-Adresse 10.101.82.52. Sobald die Konfiguration beendet ist, verhält sich die USB-Schnittstelle wie eine normale Netzwerkschnittstelle. Und die Zentrale kann beispielsweise über ihre IP-Adresse im Browser aufgerufen werden.

Wird die Zentrale an einen Rechner angeschlossen, bei dem Windows 8 als Betriebssystem läuft, so wird die Installation ein bisschen komplizierter. Bei Windows 8 hat man nämlich das Problem, dass dieses die Installation von unsignierten Treibern wie den von HomeMatic blockiert. Um das zu ändern muss man einen speziellen Start von Windows herbeiführen, wo solche Treiber akzeptiert werden.
Dazu muss man zuerst die "Windows"-Taste und "R" drücken und beim erscheinenden Fenster folgenden Befehl eingeben, der einen Neustart des Systems herbeiführt:

```
1 shutdown.exe /r /o /f /t 00
```

Listing 5.8: Befehl zum Neustart des Windows-Systems mit zusätzlichen Optionen

Nachdem Windows neu gestartet ist, kommt man nun in ein Menü, wo man nach der Reihe "Problembehandlung", "Erweiterte Optionen" und "Starteinstellungen" auswählen muss. Ist man bei den Starteinstellungen, drück man die "7". Jetzt wird das Betriebssystem ganz normal gestartet, mit dem Unterschied, dass die Treibersignatur nicht mehr erzwungen wird. Somit kann man jetzt das Setup ohne Probleme starten und die Treiber werden installiert.

5.3.2 Erstellung des Maven-Projektes

Ich verwende für die Erstellung des Servers Maven, damit sich dieses um das Generieren einer selbstständig ausführbaren Jar-Datei kümmert. Außerdem kümmert sich Maven um das Importieren aller benötigten Bibliotheken beziehungsweise Dependencies. Jedes Maven-Projekt hat eine eigene Konfigurationsdatei "pom.xml", in der die gesamte Konfiguration vom Maven enthalten ist. Darin werden allgemeine Projektdaten wie Name und Version des Projekts gespeichert, aber auch, welche Bibliotheken das Projekt verwendet und welche Maven-Plugins bei welchen Projektabschnitt verwendet werden.

Installieren der Java-Bibliotheken

Um Bibliotheken mit Maven zum Projekt hinzuzufügen, hat man bei M2Eclipse, also dem Eclipse-Plugin für Maven, zwei Möglichkeiten. Zum einen kann man in einer GUI-Oberfläche das Maven-Repository mithilfe von Stichwörtern durchsuchen, und so Bibliotheken hinzufügen. Zum anderen hat man auch die Möglichkeit direkt den Textinhalt der "pom.xml" zu ändern. Bei der GUI-Variante wird nämlich ebenfalls nur diese Datei umgeschrieben.
Der Server braucht im allgemeinen folgende Bibliotheken:

- **xmlrpc-client**: Diese Bibliothek wird benötigt, damit man mit der HomeMatic-XMLRPC-Schnittstelle kommunizieren kann, wenn man die Sensordaten dort als Metadaten hinzufügen will.
- **org.ektorp**: Mithilfe von dieser Bibliothek greift der Server auf die CouchDB-Datenbank zu
- **org.ektorp.spring**: Diese Bibliothek ist eine Erweiterung der Ektorp-Bibliothek und vereinfacht und beschleunigt das Konfigurieren von Repositories indem es zusätzliche Annotationen zur Verfügung stellt.
- **json**: Um die Sensordaten vom NodeJS-Server einlesen zu können, wird diese Bibliothek verwendet.

Um die Bibliotheken nun ins Projekt zu laden, muss ein Maven-Build mit der Phase “install” gestartet werden. In M2Eclipse ist das ein Standardbefehl, welcher im Menü “Run” im Untermenü “Run As” ausgewählt werden kann. Wird der Befehl ausgeführt, wird das Maven-Install-Plugin gestartet, welches sich um das Laden der Bibliotheken ins Projekt kümmert.

Umschreiben der Maven-Plugins

Bei M2Eclipse gibt es zwei verschiedene XML-Ansichten für die “pom.xml”. Bei der “Effective POM”-Ansicht sieht man die fertige Datei, wie sie aussieht, nachdem M2Eclipse Standard-Plugins und andere Konfigurationen hinzugefügt hat. In der zweiten Ansicht können eigene Konfigurationen im XML-Format hinzugefügt werden. Die “Effective POM” wird dann so abgeändert, dass sie alle benutzerdefinierten Konfigurationen enthält. Fügt man bei der zweiten Ansicht ein in der “Effective POM” existierendes Plugin mit besonderer Konfiguration hinzu, so ersetzt dieses einfach das existierende Plugin.
Wichtig beim Umschreiben der “pom.xml” ist es, dass man auch im M2Eclipse-Plugin die vorgegebene Struktur von Maven einhalten muss. Das bedeutet, wenn man ein Plugin ändern will, muss man zuerst die übergeordneten Tags vom Tag “<plugin>” auch in der benutzerdefinierten Datei hinzufügen, damit das Plugin-Tag dann an der richtigen Position ist. Die übergeordneten Tags werden in der “Effective POM” nicht ersetzt, sondern nur das Plugin-Tag.
Das Umschreiben von Plugins wird bei diesem Projekt benötigt, um vom Server eine ausführbare Jar-Datei zu generieren. Dafür muss das Assembly-Plugin von Maven neu konfiguriert werden, da dieses für das Generieren der Jar-Datei verantwortlich ist. Einerseits muss man dazu angeben, wo sich die Main-Klasse befindet und andererseits muss man dem Plugin ebenfalls sagen, dass alle Bibliotheken zur Jar-Datei hinzugefügt werden sollen.

Mit dem ersten Tag des Plugins “artifactId” wird identifiziert, welches Plugin geändert werden soll. Mit dieser ID weiß dann auch M2Eclipse, welches Tag in der “Effective POM” ersetzt werden muss. Beim nächsten Tag “configuration” kann man nun das Verhalten des Plugins einstellen. In unten stehendem Code-Ausschnitt ist zu sehen welches Tag hier eingefügt werden muss, damit Java beim Ausführen der Jar-Datei die Main-Klasse findet. In diesem Fall befindet sich die Main-Klasse im Paket “writer” und hat den Namen “Main”.

```
1 <archive>
2    <manifest>
3       <mainClass>writer.Main</mainClass>
4    </manifest>
5 </archive>
```

Listing 5.9: Konfiguration zum Generieren einer Ausfürhbaren Jar-Datei

Im “configuration”-Tag kann man auch einstellen, dass alle benötigten Bibliotheken ebenfalls in die Jar-Datei geschrieben werden sollen. Wenn man folgendes Tag hinzufügt, generiert das Plugin schon die richtige Jar-Datei:

```
1 <descriptorRefs>
2    <descriptorRef>jar-with-dependencies</descriptorRef>
3 </descriptorRefs>
```

Listing 5.10: Konfiguration zum Generieren einer Jar-Datei mit integrierten Bibliotheken

Abschließend besteht noch die Möglichkeit, dem Plugin zu sagen es soll auch automatisch gestartet werden, wenn ein Maven-Prozess bei der Phase “install” eine neue Bibliothek ins Projekt

lädt. Dazu muss im Tag “executions”, welcher eine Liste von “execution”-Tags speichert, ein Eintrag geschrieben werden. Ein “execution”-Tag beschreibt hierbei eine Situation, in welcher das Plugin ausgeführt wird. Wie in dem Code-Ausschnitt unten zu sehen ist, hat jedes “execution”-Tag eine ID, mit der verhindert wird, dass beim Zusammenführen mehrerer “pom.xml”-Dateien ein Tag überschrieben wird. Dann muss man noch die Phase und die Ziele angeben, bei denen das Plugin gestartet werden soll. Wenn man ein “execution-Tag” mit einer neuen Phase erstellt, ändert das aber nichts an der Phase bei der es standardmäßig ausgeführt wird.

```
1 <execution>
2    <id>make-assembly</id>
3    <phase>package</phase>
4    <goals>
5       <goal>single</goal>
6    </goals>
7 </execution>
```

Listing 5.11: Tag, welches einstellt, wann das Assembly-Plugin ausgeführt werden soll.

Um dieses Plugin auszuführen, muss ein Maven-build “assembly:single” gestartet werden. Hier bezieht sich “assembly” auf die Projekt-Phase und “single” auf das Ziel. Bei diesem Befehl alleine werden Java-Klassen nicht extra neu kompiliert. Wenn man also sicher die aktuellen Versionen von Java-Klassen zum Projekt hinzufügen möchte, muss man vorher noch die Phasen “clean” und “compile” ausführen. Man kann diese drei Befehle auch im selben Maven-build mit einem Leerfeld als Trennzeichen angeben: “clean compile assembly:single”.
Um in Eclipse einen benutzerdefinierten Maven-Befehl zu verwenden muss man wieder im Menü “Run” das Untermenü “Run As” auswählen und dort “Maven build...” anklicken. Dann öffnet sich ein Fenster, wo man in dem Feld “Goals” alle durchzuführenden Phasen und Ziele eingeben kann. Man sollte bei diesem Fenster auch aufpassen, dass das richtige Projekt beim Feld “Base directory” ausgewählt ist. Abschließend drückt man dann auf “Run” und die fertige Jar-Datei wird erstellt.

5.3.3 Programmaufbau

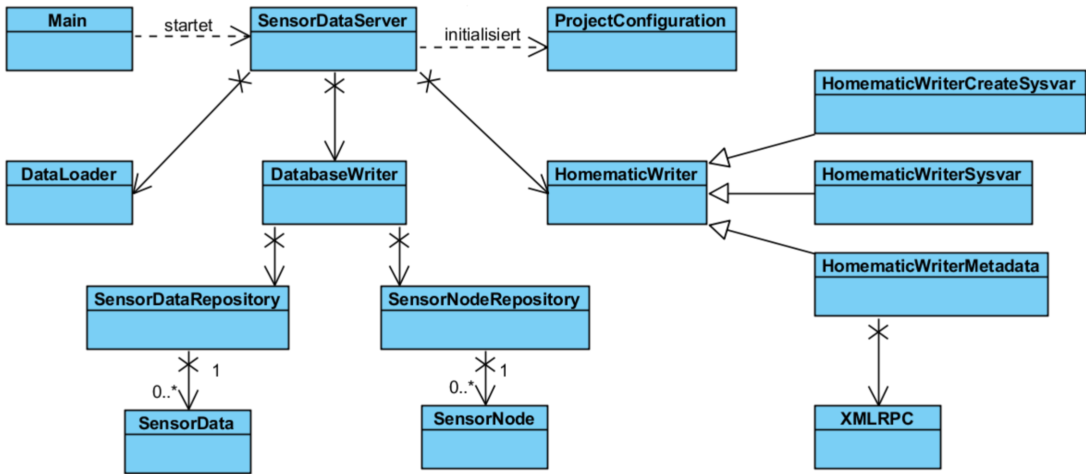


Abbildung 5.1: Das Klassendiagramm für den Server, der die Sensordaten überträgt.

Der Einstiegspunkt des Programms liegt in der Klasse **Main**. Von dort wird die Klasse **SensorDataServer** initialisiert und gestartet. Dieser lädt während der Initialisierung mithilfe der Klasse **ProjectConfiguration** die Konfigurationsdatei und initialisiert die drei Klassen **DataLoader**, **DatabaseWriter** und **HomematicWriter**.
Der **DataLoader** ist für das Laden der Sensordaten vom NodeJS-Server zuständig.
Der **DatabaseWriter** verwendet zwei Repositories namens **SensorDataRepository** und **SensorNodeRepository**, um auf die CouchDB-Datenbank zuzugreifen. Diese Repositories verwalten jeweils eine Liste von Entitäten namens **SensorData** und **SensorNode**.
Bei der abstrakten Klasse **HomematicWriter** erzeugt der **SensorDataServer** je nach Konfiguration ein Objekt der Klasse **HomematicWriterCreateSysvar**, **HomematicWriterSysvar** oder **HomematicWriterMetadata**. Der Unterschied der drei Klassen liegt in der Art, wie sie die Sensordaten in die HomeMatic-Zentrale schreiben. Die Klasse **HomeMaticWriterMetadata** verwendet zusätzlich noch die Klasse **XMLRPC** um über diese auf die gleichnamige Schnittstelle der Zentrale zuzugreifen.

5.3.4 Start des Programms

In der Main-Methode des Programms wird zuerst eine Instanz der Klasse **SensorDataServer** angelegt und anschließend dessen Methode “runServer()” ausgeführt.
Im Konstruktor dieser Klasse werden zuerst die Konfigurationsdaten aus der Konfigurationsdatei geladen und gespeichert, indem man eine Instanz der Klasse **ProjectConfiguration** anlegt. Als nächstes werden jeweils Instanzen der Klassen **DataLoader**, **DatabaseWriter** und **HomematicWriter** angelegt. Da sich je nach Konfiguration die Methode ändert nach der die Sensordaten, nach HomeMatic geschrieben werden, muss für jede Methodenart eine andere Instanz von **HomematicWriter** verwendet werden. In dem Code-Ausschnitt unten ist die Abfrage zu sehen, nach der entschieden wird, welche Instanz von **HomematicWriter** verwendet wird. Wurde keine gültige Methode für das Schreiben der Sensordaten in die HomeMatic-Zentrale angegeben, so wird dieser Schritt einfach ausgelassen und das Attribut leer gelassen.

```
if(methodType.equals("Metadata")){
   this.hw = new HomematicWriterMetadata();

}else if(methodType.equals("Sysvar")){
   this.hw = new HomematicWriterSysvar();

}else if(methodType.equals("CreateSysvar")){
   this.hw = new HomematicWriterCreateSysvar();
}else{
   this.hw = null;
}
```

Listing 5.12: Bei diesem Code wird je nach Konfiguration eine Bestimmte Instanz der Klasse HomematicWriter erstellt.

Außerdem werden vom DatabaseWriter und vom HomematicWriter nur Instanzen erzeugt, wenn sich diese Erfolgreich mit der Datenbank bzw. der HomeMatic-Zentrale verbinden können. Schaffen sie das nicht, werfen die Konstruktoren Exceptions und die Attribute werden leer gelassen.

Nachdem der **SensorDataServer** in seinem Konstruktor das gesamte Projekt initialisiert hat, wird in der Main-Klasse noch seine einzige Methode “runServer()” ausgeführt. Diese besteht hauptsächlich aus einer Schleife, welche sich die aktuellen Sensordaten holt und diese weitergibt, damit sie in die Datenbank bzw. in die HomeMatic-Zentrale geschrieben werden können. Diese Schleife wird aber nur gestartet, wenn entweder der **DatabaseWriter** oder der **HomeMaticWriter** oder beide im Konstruktor erfolgreich initialisiert werden konnten. Es macht keinen Sinn, wenn man zwar Daten laden kann, aber diese nirgends gespeichert werden können.

Zuerst prüft die Methode, ob überhaupt Sensordaten empfangen werden können. Diese Überprüfung passiert bei jedem Schleifendurchlauf, wodurch der Server sofort abgebrochen wird, wenn keine Sensordaten geladen werden konnten. Die Sensordaten werden somit zuerst über den **DataLoader** geladen und als **JSONObject** gespeichert. Ist das Objekt der **JSONObject**-Variable leer, so wird die Schleife verlassen und der Server somit beendet.
Im nächsten Schritt werden die Sensordaten in die Datenbank geschrieben, indem die Methode “writeToDatabase(**JSONObject** obj)” vom **DatabaseWriter** aufgerufen wird. Dieser Aufruf passiert nur, wenn der **DatabaseWriter** erfolgreich Instanziert werden konnte. Nachdem die Sensordaten erfolgreich in die Datenbank geschrieben wurden, werden beginnend mit den ältesten, noch so viele Dokumente aus der Datenbank gelöscht, bis man das erlaubte Max-

imum an Dokumenten erreicht hat. Das erlaubte Maximum wird in der Konfigurationsdatei angegeben. Sollte es sich dabei um einen Wert kleiner als oder gleich null handeln, wird diese Funktion deaktiviert und die Methode nicht aufgerufen.

Sollte der **HomematicWriter** erfolgreich instanziert worden sein, so wird im nächsten Schritt seine Methode “writeToHomematic(**JSONObject** obj())” aufgerufen, um die Sensordaten in die Zentrale zu schreiben. Diese Methode ist abstrakt und wie sie ausimplementiert ist, hängt davon ab, welche Instanz vom **HomeMaticWriter** angelegt wurde.

Im letzten Schritt wird noch eine gewisse Zeit gewartet, bevor der nächste Schleifendurchlauf begonnen wird. Wie lange gewartet wird, steht in der Konfigurationsdatei.

5.3.5 Laden der Daten aus der Konfigurationsdatei

Die Konfigurationsdaten werden von der Klasse **ProjectConfiguration** geladen. Diese Klasse ist die einzige, die direkt mit der Konfigurationsdatei arbeitet.

Instanzieren von ProjectConfiguration

Damit es immer nur eine Instanz dieser Klasse gibt, implementiert sie das Entwurfsmuster eines Singletons. Das bedeutet, dass die Klasse nur einen privaten Konstruktor hat. Um trotzdem noch zu einer Instanz dieser Klasse zu kommen, gibt es, wie in dem Code-Ausschnitt unten zu sehen ist, eine statische Methode “getInstance()”, die eine instanz der Klasse zurückgibt. Bei einem Aufruf der Methode wird dann entweder eine neue Instanz der Klasse erzeugt und in ein statisches Attribut geschrieben, oder dieses statische Attribut zurückgegeben, wenn es bereits eine Instanz enthält. Dadurch wird sicher gestellt, dass nur beim ersten Aufruf der Methode eine Instanz erzeugt wird und bei allen anderen Aufrufen die bereits existierende Instanz zurückgegeben wird.

```
1 private static ProjectConfiguration instance;
2
3 public static ProjectConfiguration getInstance(){
4    if(ProjectConfiguration.instance == null)
5       return new ProjectConfiguration();
6
7    return ProjectConfiguration.instance;
8 }
```

Listing 5.13: Das Singleton-Design-Pattern in der ProjectConfiguration-Klasse

Laden der Konfigurationsdaten

Im Konstruktor der Klasse werden alle Daten aus der Konfigurationsdatei geladen. Gesucht wird die Datei in den App-Daten des aktuellen Benutzers. Konkret sucht der Server in dem Unterordner "SensorData" vom App-Daten-Ordner nach einer Datei namens "config.txt" Findet er die Datei nicht, legt er selbst eine Datei mit Standardwerten an. In unten Stehendem Code-Ausschnitt ist zu sehen wie die Datei aussieht, wenn sie neu angelegt wurde:

```
HomeMaticServer: http://10.101.82.51:2001
SensorDataServer: http://localhost:8080
CouchDBServer: http://localhost:5984
HomeMaticWriteMethod: CreateSysvar
CouchDBLogSkips: 60
CouchDBMaxEntries: 500000
DataLoadingTimeout: 10000
```

Listing 5.14: Die standardmäßige Konfiguration der Konfigurationsdatei des Servers.

Die Daten werden anschließend in eine HashMap geladen, die die Konfigurationsdaten als eine Kombination eines String-Schlüssels und eines String-Wertes speichert.

Weitergeben der Konfigurationsdaten

Zum Weitergeben der Konfigurationsdaten gibt es die beiden Methoden "getValue(String key)" und "getInt(String key)". Beide haben als Parameter einen String als Schlüssel mit dem sie in der HashMap dann nach dem zugehörigen Wert such. Das Ergebnis der Suche wird anschließend zurückgegeben.
Bei der Methode "getInt()" kommt noch hinzu, dass der gefundene Wert noch von einem String in einen Integer umgeschrieben werden muss, damit ein Zahlenwert zurückgegeben werden kann. Ist in dem String keine Zahl im Integer-Format geschrieben, so wird stattdessen null zurückgegeben.

5.3.6 Einlesen der Sensordaten von MeshSSN

Das einlesen der Sensordaten passiert in der Klasse **DataLoader**. Beim initialisieren der Klasse holt sich diese eine Instanz der ProjectConfiguration und speichert diese als Attribut.
Die Methode "loadData()" der Klasse wird vom **SensorDataServer** aufgerufen, damit dieser die aktuellen Sensordaten bekommt. Dazu baut die Methode bei jeder Anfrage eine Verbindung zu der in der Projektkonfiguration angegebenen URL auf, holt sich von dieser die Sensordaten im JSON-Format und speichert sie in einem JSONObject. Das JSONObject gibt er anschließend zurück. Passiert beim Verbindungsaufbau ein Fehler, gibt er "null" zurück.

5.3.7 Schreiben der Sensordaten in CouchDB

Für die Kommunikation mit der Datenbank wird vom Server Ektorp verwendet. Dabei handelt es sich um eine Implementierung der "Java Persistence API", weshalb wir ein Entitäts-Modell brauchen, das das Daten-Modell der Datenbank abbildet.

Datenbankmodell

Der Server arbeitet grundsätzlich mit zwei Arten von Dokumenten in der Datenbank und somit gibt es auch zwei verschiedene Entitätsklassen. Die Klassen heißen **SensorNode** und **SensorData**. Für jeden Sensor in der dem MeshSSN-Netzwerk gibt es ein konkretes Dokument in der Datenbank das einer **SensorNode** entspricht und für jede **SensorNode** gibt es mehrere **SensorData**-Dokumente, welche die Sensordaten von einem bestimmten Zeitpunkt speichern und auf eine bestimmte **SensorNode** verweisen. Aber auch jede **SensorNode** speichert eine Referenz auf eine bestimmte **SensorData**-Entität die die aktuellsten Sensordaten speichert.
Die Entitäts-Klassen speichern eine Reihe von Attributen (mit Gettern und Settern), welche auch in den Dokumenten gespeichert werden. Eine Entitäts-Klasse muss von **CouchDbDocument** erben um als solche angesehen zu werden. Des weiteren werden verschiedene Annotationen benötigt, damit ein Dokument abgebildet werden kann.

Bei einer Entitäts-Klasse wird das Aussehen eines Dokuments über seine Attribute und Methoden bestimmt. Wenn ein Attribut oder eine Methode nichts mit den im Dokument gespeicherten Daten zu tun hat, muss man eine Annotation namens "@JsonIgnore" vor das Element setzen, damit es die Generierung oder das Einlesen eines Dokuments nicht beeinflusst. Beispielsweise wird diese Annotation bei **SensorData** für die Methode "getTimestampDate()" benötigt, da diese nur das Attribut "timestamp" als ein Datumsformat zurückgibt. Ohne die Annotation würde diese Methode als Getter für ein Attribut gehalten werden und beim Schreiben in die Datenbank würde beim Dokument auf einmal ein JSON-Schlüssel "timestampDate" entstehen.

Eine Weitere Annotation, wird unbedingt benötigt, um entscheiden zu können, ob es sich bei einem Dokument in der Datenbank um eine bestimmte Entität handelt: Die Annotation " TypeDiscriminator(<String>)". Diese Annotation fügt man ganz am Anfang vor der Klasse ein. Anhand von dieser Annotation wird es Ektorp ermöglicht eine View zu erzeugen. Wie man aus dem Kapitel **2.6.3 Zugriff auf die Daten** entnehmen kann, besteht eine View-Definition im Endeffekt nur aus einer Callback-Methode, welche für jedes Dokument aufgerufen wird. Ektorp kann diese Methode selbst erzeugen, braucht aber dafür den Inhalt der Bedingung der "if"-Verzweigung. Ist diese Bedingung für das jeweilige Dokument wahr, so wird dieses zurückgegeben. Die Bedingung bekommt sie von dem String-Parameter der "TypeDiscrimpnator"-Annotation.
In dem Code-Ausschnitt unten sieht man ein Beispiel einer solgen generierten Callback-Methode von Ektorp für die Entität **SensorNode**. Um genau diese Callback-Methode zu generieren, muss die Annotation dann "@TypeDiscriminator("doc.type == 'SensorNode' ")" lauten.

```
function(doc) {
   if(doc.type == 'SensorNode')
   {
      emit(null, doc._id)
   }
}
```

Listing 5.15: Die von Ektorp generierte Callback-Methode einer View.

Neben den Entitäten, die die bestimmten Dokument-Arten abbilden, gibt es auch noch Repositories, die den Zugriff auf die Datenbank regeln. Jedes Repository verwaltet eine bestimmte Entität. Somit gibt es auf dem Server zwei Repositories namens **SensorDataRepository** und **SensorNodeRepository**.
Bei der Definition von Repositories kann in Ektorp das Spring-Framework verwendet werden. Das Spring-Frameword ermöglicht es dem Programmierer die Definition von Repositories und Entitäten durch neue Annotationen sehr abzukürzen. Die Annotationen des Springframeworks werden bei Ektorp vor allem bei den Konstruktoren verwendet.

Die erste Annotation ist vor den Konstruktor der Repositories gesetzt und heißt "@Autowired". Diese Annotation führt dazu, dass Java aus der zugehörigen Entität eine Java-Bean erzeugt, also eine Klasse, die nur aus Attributen und entsprechende Gettern und Settern besteht. Diese Java-Bean wird in Ektorp dann zur Erzeugung von Dokumenten verwendet. Zum erstellen der Java-Bean überprüft sie alle Attribute der Entität, ob diese public sind, und alle Methoden, ob diese ein "get" oder "set" am Anfang ihres Namens stehen haben. Diese Überprüfung ist auch der Grund dafür, dass bei **SensorData** die Methode "getTimestampDate" eine extra Annotation "@JsonIgnore" benötigt, mit der man sagt, dass diese Methode bei der Überprüfung übersprungen werden soll.

Die zweite Annotation steht vor einem Parameter des Konstruktors, der den Namen der Datenbank beinhaltet. Bei dem Parameter selbst handelt es sich um einen **CouchDbConnector** welcher die Verbindung mit der Datenbank herstellt. Sollte es die Datenbank aber noch gar nicht geben und der Parameter leer sein, muss die Datenbank vorher noch erstellt werden. Dazu wird über die Annotation der Name der Datenbank mitgegeben, da er diesen sonst nicht wissen würde.

Der Konstruktor ruft somit zuerst den Konstruktor seiner Superklasse auf, dem er noch die Klasse der Entität übergibt, die er verwaltet und den **CouchDbConnector**. Danach lädt er noch alle Dokumente, die zum Repository gehören. In dem Code-Ausschnitt unten sieht man den Konstruktor der Klasse **SensorDataRepository**.

```
1 @Autowired
2 public SensorDataRepository(@Qualifier("sensordb") CouchDbConnector db) {
3    super(SensorData.class, db);
4    initStandardDesignDocument();
5 }
```

Listing 5.16: Der Konstruktor eines Repositories in Ektorp.

Außerdem überschreibt das Repository die Methode "getAll()" die alle Dokumente zurückgibt, die zu der vom Repository verwalteten Entität passen. Das macht sie, indem sie eine Query zur Datenbank absendet, die alle passenden Dokumente in aufsteigender Reihenfolge zurückgibt. Damit bei der Antwort der Query die richtigen Objekte erzeugt werden, wird die Klasse der Entitäten als Parameter mitgegeben.
Die Methode hat außerdem eine Annotation "@GenerateView", mit dem Ektorp gesagt wird es soll das Design-Dokument, das bei dieser Methode temporär erzeugt wird, abspeichern und dann dieses verwenden. Wenn die View als Design-Dokument in der CouchDB gespeichert ist, kann man sie natürlich auch anpassen.
Um später die ältesten Sensordaten aus der Datenbank löschen zu können, muss man die Dokumente in aufsteigender Reihenfolge sortiert nach ihrem Entsehungsdatum zurück bekommen.

Dokumente, die man von CouchDB bekommt, können aber nur nach ihrem Schlüssel sortiert werden. Dieser ist bei der von Ektorp generierten View "null". Um die Dokumente nach ihrem Entstehungsdatum zu sortieren, kann man nun manuell den Schlüssel auf "doc.timestamp" setzen. Wenn man das getan hat, kann nun der Server die ältesten Sensordaten aus der Datenbank löschen.

Programmablauf

Für den Ablauf der Persistierung der Sensordaten in der Datenbank ist der **DatabaseWriter** zuständig. Beim Instanzieren dieser Klasse, wird versucht eine Verbindung mit der Datenbank aufzubauen. Scheitert der Verbindungsversuch, wirft der Konstruktor eine "MalformedURLException". Außerdem wird die Instanz der **ProjectConfiguration** in ein Attribut gespeichert und das Attribut "logCounter", das zählt, wie oft das Archivieren der Sensordaten schon übersprungen wurde, wird mit 0 initialisiert. Der **SensorDataServer** ruft periodisch die Methode "writeToDatabase(**JSONObject** obj)" mit den aktuellen Sensordaten als Parameter auf. Die Methode überschreibt für jeden Sensor die aktuellen Sensordaten und archiviert die aktuellen Sensordaten, sofern dieser Schritt schon oft genug übersprungen wurde. Dafür geht die Methode alle Einträge des **JSONObject**s in einer Schleife durch.
Beim Schreiben der aktuellen Sensordaten eines Sensors gibt es drei Möglichkeiten:

- Wenn sowohl die **SensorNode**-Entität als auch die **SensorData**-Entität auf das diese referenziert existieren, muss nur die **SensorData**-Entität auf die aktuellen Sensordaten aktualisiert werden.

- Wenn zwar die **SensorNode**-Entität existiert, aber die SensorData-Entität auf das diese referenziert nicht, muss eine neue **SensorData**-Entität mit den aktuellen Sensordaten angelegt werden und die Referenz der **SensorNode**-Entität auf die neue **SensorData**-Entität gesetzt werden.

- Wenn die **SensorNode**-Entität noch nicht existiert, müssen beide Entitäten neu angelegt werden.

Sollte das Aktualisieren der Entitäten nicht funktionieren wird eine Fehlermeldung ausgegeben.
Vor der Schleife über alle Einträge vom **JSONObject** wird noch überprüft ob die Sensordaten archiviert werden müssen. Dazu wird die Methode "dbLogNeeded()" aufgerufen, welche als Antwort einen boolean-Wert zurückgibt. Ist dieser "wahr" werden in der Schleife die aktuellen Sensordaten archiviert indem neue **SensorData**-Entitäten erzeugt werden.
Die Methode "dbLogNeeded()" erhöht bei jedem Aufruf das Attribut "logCounter" um 1 und gibt "false" zurück. Sollte aber das Attribut "logCounter" größer werden als die in der Konfigurationsdatei angegebenen Anzahl an zu überspringenden Archivierungen, wird das Attribut "logCounter" stattdessen auf 0 zurückgesetzt und "true" zurückgegeben.
Somit werden bei jedem Aufruf der Methode "writeToDatabase(**JSONObject** obj)" die aktuelllen Sensordaten überschrieben und alle paar Aufrufe werden die Sensordaten archiviert.

5.3.8 Schreiben der Sensordaten in HomeMatic

Um die Sensordaten an die HomeMatic-Zentrale zu senden, gibt es insgesamt drei verschiedene Methoden. Ich habe das gelöst, indem ich eine abstrakte Superklasse **HomematicWriter** erstellt habe, deren abstrakte Methode “writeToHomematic(**JSONObject** obj)” von drei verschiedenen Subklassen implementiert wird. Je nach Projektkonfiguration wird darauf hin während der Laufzeit eine der drei Klassen instanziert und deren Methode ausgeführt. Die Superklasse hat zudem einen “protected” Konstruktor, der somit nur von den Subklassen aufgerufen werden kann. Dieser ist dafür zuständig, die Instanz **ProjectConfiguration** in ein Attribut zu speichern.

Im nächsten Teil werden die drei Möglichkeiten beschrieben, wie die Daten an die HomeMatic-Zentrale gesendet werden können:

Metadaten ändern

Damit die HomeMatic-Zentrale die Daten als Metadaten speichert muss, auf die XMLRPC-Schnittstelle von HomeMatic zugegriffen werden. Dazu verwendet der Server eine Bibliothek zum erstellen eines ganz normalen XMLRPC-Clients. Die HomeMatic-Zentrale hat zwar zwei verschiedene XMLRPC-Schnittstellen, aber der Server muss sich mit dem Port 2001 der HomeMatic-Zentrale verbinden, da der Port 2000 keinen Zugriff auf die Methoden zum Hinzufügen von Metadaten bietet.
Die Klasse die die Methode “writeToHomematic(**JSONObject** obj)” implementiert heißt in diesem Fall **HomeMaticWriterMetatdata**. Die Kommunikation mit der Zentrale bzw. mit dem Schnittstellenprozess erfolgt aber über eine andere Klasse namens **XMLRPC**. Diese Klasse wird im Konstruktor von **HomeMaticWriterMetatdata** erzeugt und regelt im eigenen Konstruktor den Verbindungsaufbau zur Zentrale. Für diesen werden bereits Klassen von der XML-RPC-Bibliothek verwendet und somit muss nur noch die URL als Konfiguration übergeben werden.
Nachdem beide Klassen nun erstellt wurden wird als nächstes periodisch die Methode “writeToHomematic(**JSONObject** obj)” aufgerufen. Dieses ruft nun für jeden Sensor die Methode “setNodeMetadata(String key, **JSONObject** obj)” von der **XMLRPC**-Klasse auf, der es den Namen des zu schreibenden Metadaten-Objekts und die Sensordaten eines bestimmten Sensors übergibt. Diese Methode schreibt dann für jeden Wert des Sensordaten-Objekts einen Eintrag im Metadaten-Objekt auf der Zentrale. Sollte es das Metadaten-Objekt noch nicht auf der Zentrale geben, generiert es die XMLRPC-Methode automatisch.
Der konkrete Ablauf des Methodenaufrufs funkioniert so, dass zuerst alle Parameter der Methode in einen **Vector** geschrieben werden. Die Klasse **Vector** leitet von **AbstractList** ab und speichert somit eine Liste von Werten. Wie in dem Code-Ausschnitt unten zu sehen ist speichert der Vector generische Objects, da die Parameter für den XMLRPC-Methodenaufruf beliebige Typen haben können. Sind alle Parameter hinzugefügt, wird der Vector an die “execute”-Methode vom **XmlRpcClient**, welcher beim Verbindungsaufbau als Attribut erstellt wurde, als ein zweiter Parameter übergeben. Der erste Parameter enthält den Namen der Methode die Aufgerufen werden soll. Anhand von Methodenname und Parametern wird dann eine XML-Struktur erzeugt und an den Server gesendet.

```
Vector<Object> params = new Vector<Object>();
params.add(node_name);
params.add(key);
params.add(obj.get(key));
client.execute("setMetadata", params);
```

Listing 5.17: Ein XMLRPC-Methodenaufruf mit einer Bibliothek von Java.

Ein Metadaten-Objekt speichert eine Liste von Schlüsseln und Werten. Somit muss bei dem Methodenaufruf von "setMetadata" der Name des Metadaten-Objekts, ein Schlüssel als String und der dazugehörige Wert übergeben werden. Der Name ("node_name") des Metadaten-Objekts ist bei dem Code-Ausschnitt oben "MeshSSNNode_" mit der ID des Sensors hinten dran. Der Schlüssel (also der zweite Parameter) ist die Art des gespeicherten Sensor-Datenpunkts, beispielsweise "light". Und der dritte und letzte Parameter ist der aktuelle Wert des Sensor-Datenpunkts.

In der Klasse **XMLRPC** gibt es noch einige weitere Methoden die unterschiedliche XMLRPC-Methoden der Zentrale aufrufen. Mit einem Großteil von ihnen ist es möglich Gerätedaten zu ändern, oder auszulesen. Beim Auslesen von Werten muss man beim Erstellen einer neuen XMLRPC-Methode als erstes herausfinden wie das Rückgabeformat des "execute"-Methodenaufrufs aussieht. Man bekommt nämlich immer nur ein Generisches Object zurück, das man manuell in den richtigen Datentyp umwandeln muss. Hierbei hilft einerseits die offizielle XML-RPC-Schnittstellen-Spezifikation von HomeMatic und der Debug-Modus der Entwicklungsumgebung.

Systemvariablen ändern

Um diese Methode verwenden zu können, wird auf der Zentrale das Plugin XML-API benötigt. Bei der XMLRPC-Schnittstelle ist es nämlich nicht möglich auf Systemvariablen zuzugreifen und diese zu ändern. Außerdem ist es bei XML-API nicht nötig eine XML-Struktur aufzubauen und diese an die Zentrale zu senden. Um auf eine bestimmte Funktion von XML-API zuzugreifen, ist nur ein Aufruf einer URL mit bestimmten Parameter notwendig. Das ermöglicht es dem Benutzer, dass er XML-API nicht nur in einem Programm, sondern auch über den Browser aufrufen kann.

Um eine Systemvariable zu ändern, muss, wie in dem Code-Ausschnitt unten zu sehen ist, "statechange.cgi" aufgerufen werden. Diese URL braucht dann noch zwei Parameter: Den Namen oder die ID der Systemvariable und den Wert auf den diese gesetzt werden soll. Man kann bei den Parametern auch eine Liste von Namen und Werten übergeben, welche dann alle nach der Reihe abgearbeitet werden. Dadurch erspart man sich den Aufwand, die "statechange.cgi" mehrmals aufrufen zu müssen, wenn man mehrere Variablen ändern will.
Bei diesem Konkreten Aufruf wird die Systemvariable "MeshSSNNode_1_light" auf den Wert 20 gesetzt und die Systemvariable "MeshSSNNode_1_humidity" auf den Wert 27.

```
10.101.82.51/config/xmlapi/statechange.cgi?ise_id=MeshSSNNode_1_light,
MeshSSNNode_1_humidity&new_value=20,27
```

Listing 5.18: Ein Aufruf von XML-API, der zwei Systemvariablen ändert.

Der einzige Nachteil von “statechange.cgi” ist, dass es keine Systemvariablen erstellen kann, wenn der übergebene Name noch nicht existiert. Das hat den Grund, dass man mit “statechange.cgi” auf viele unterschiedliche Objekte zugreifen kann, da es ein HMScript erzeugt, das das gesamte System nach einem Objekt mit diesem Namen durchsucht. Die Methode “dom.getObject(<String >)”, die XML-API verwendet, kann beispielsweise auch Geräte oder Kanäle zurückgeben. Zwar würde aber das Setzen eines Wertes, wie es bei der “statechange.cgi” passiert, nicht funktionieren, da ein Gerät oder Kanal keinen bestimmten Wert haben, den man setzen kann. Aber selbst wenn das Programm sicher sein kann, dass man eine Systemvariable setzen möchte, wüsste sie immer noch nicht, welchen Typ diese haben soll. Deshalb erstellt das Programm keine eigenen Systemvariablen wenn ein Name nicht gefunden wird, sondern überspringt diese unbekannten Namen einfach.

Um die Sensordaten als Systemvariablen hinzuzufügen, wird im Java-Programm die Klasse **HomematicWriterSysvar** verwendet. Beim Instanzieren der Klasse überprüft diese, ob sie sich mit dem XML-API-Add-On der Zentrale verbinden kann. Schlägt diese Verbindung fehl, wird eine Exception zum Aufrufer des Konstruktors zurückgesendet. Dadurch weiß dann der **SensorDataServer**, der den HomeMaticWriter instanziert, dass die Daten nicht zur Zentrale geschrieben werden können, da das XML-API-Add-On der Zentrale nicht erreicht werden kann.
In der Methode “writeToHomematic(**JSONObject** obj)” von **HomematicWriterSysvar** wird zuerst die “sysvarlist.cgi” von XML-API aufgerufen, damit man eine Liste aller existierenden Systemvariablen erhält. Die Systemvariablen bekommt man in einem XML-Format zurück, deshalb wird die Antwort im **HomematicWriterSysvar** mit einem XML-Parser eingelesen.
Dann wird in einer Schleife bei jeder Systemvariable überprüft, ob sie einen bestimmten Namen hat. Die genauen Regeln, wie dieser Name auszusehen hat, kann man im Kapitel Benutzung **3.1.4 HomeMaticWriteMethod** nachlesen. In dem Namen der Systemvariable ist die ID des zu speichernden Sensors enthalten und welcher Datenpunkt von diesem Sensor gespeichert werden soll. Mithilfe von diesen zwei Werten kann man sich bei den Sensordaten, die vom Server eingelesen wurden, den aktuellen Wert des bestimmten Datenpunkts von dem bestimmten Sensor holen. Der Name der Systemvariable und der zu schreibende Wert wird dann an eine Methode namens “updateSysVar” weitergegeben. Diese Ruft dann mit den beiden übergebebnen Werten als Parameter “statechange.cgi” auf.
HomematicWriterSysvar liest die Systemvariablen zuerst ein, damit er nur die Systemvariablen ändert die bereits existieren. Man könnte natürlich auch einfach versuchen alle Sensordaten als Systemvariable in die Zentrale zu schreiben, weil es sowieso nur bei denen funktioniert, wo bereits eine Systemvariable existiert. Aber das würde bei vielen Sensoren einen unnötig großen Aufwand an Netzwerkverkehr bedeuten.

Systemvariablen ändern und automatisch generieren

Für diese Methode wird eine modifizierte Version von XML-API benötigt, bei der “statechange.cgi” automatisch eine Systemvariable mit dem gesuchten Namen generiert, wenn zu diesem kein Objekt gefunden wurde. Die neue Systemvariable speichert dann eine Zahl, und hat den Sensordaten entsprechende Ober- und Untergrenzen.

Bei dem modifizierten “statechange.cgi” wird das TCL-Script so abgeändert, dass das HMScript, das von diesem generiert wird automatisch eine Systemvariable erstellt, wenn kein Objekt gefunden wurde.
In dem CodeAusschnitt unten ist das HMScript zu sehen, wie es noch als String im TCL-Script gespeichert wird.

Einerseits müssen in diesem String bestimmte Zeichen, die Teil des fertigen HMScripts sein sollen mit einem "\" vorher geschrieben werden, damit das TCL-Script sie nicht als eigene Operatoren wahrnimmt. Dazu gehören beispielsweise doppelte Hochkommas und geschwungene Klammern. Andererseits soll an bestimmten Stellen der String dynamisch angepasst werden, sodass beispielsweise nach dem übergebenen Parameter für den Namen gesucht wird. Oder damit die Systemvariable auf den übergebenen Wert gesetzt wird. Dazu kann man direkt in den String Methodenaufrufe schreiben und deren Rückgabewert wird dann an die Stelle des Aufrufs gesetzt. Ein Beispiel für eine solche Methode ist "[lindex $rec_ise_id $x]". Diese Methode holt sich von dem Array "$rec_ise_id" die Position "$x" und gibt diese an den Aufrufer zurück. Bei diesem TCL-Script wird hier der Name der gesuchten Systemvariable zurückgegeben und in den String geschrieben.
Das HMScript sucht sich mit der Methode "dom.getObject" nach dem zu ändernden Objekt. Findet es keines, so wird eine neue Systemvariable angelegt und diese so eingestellt, dass sie die Sensordaten speichern kann. Zum Schluss wird der Status des Objekts auf den übergebenen Wert geändert und der geänderte Wert zurückgegeben. Anhand des Rückgabewertes kann das TCL-Script wiederum feststellen ob das HMScript erfolgreich den Wert eines Objektes geändert hat oder nicht.

```
1  object svObj = dom.GetObject(\"[lindex $rec_ise_id $x]\");
2  if (!svObj)\{
3  object svObjects = dom.GetObject(ID_SYSTEM_VARIABLES);
4  svObj = dom.CreateObject(OT_VARDP);
5  svObjects.Add(svObj.ID());
6  svObj.Name(\"[lindex $rec_ise_id $x]\");
7  svObj.ValueType(ivtFloat);
8  svObj.ValueSubType(istGeneric);
9  svObj.DPInfo(\"Speichern von bestimmten Sensordaten\");
10 svObj.ValueUnit(\"\");
11 svObj.ValueMin(0);
12 svObj.ValueMax(65535);
13 svObj.Internal(false);
14 svObj.Visible(true);
15 dom.RTUpdate(false);
16 \}
17 Write(svObj.State([lindex $rec_new_value $x]));
```

Listing 5.19: Der in "statechange.cgi" verwendete String, welcher in ein HMScript umgewandelt wird.

Bei der Installationsdatei von XML-API handelt es sich um ein Archiv, in dem alle TCL-Skripte für beispielsweise "statechange.cgi" unverschlüsselt gespeichert sind. Um das Add-On zu modifizieren, muss somit nur das entsprechende TCL-Script in dem Archiv überschrieben werden.

Wenn nicht existierende Systemvariablen automatisch generiert werden, kann man im Server nun einfach alle Datenpunkte von allen Sensoren an die "statechange.cgi" senden, ohne überprüfen zu müssen, ob diese existieren.
Diese Variante Variablen hinzuzufügen wird in der Klasse **HomematicWriterCreateSysvar** implementiert. Beim Instanzieren der Klasse wird genau wie bei **HomematicWriterSysvar** versucht eine Verbindung mit dem Add-On XML-API auf der HomeMatic-Zentrale herzustellen. Funktioniert das nicht, wird eine Exception zurückgegeben.
Außerdem wird ein String-Array auf Attribut-Ebene erzeugt. Dieses Array speichert die Namen jener Datenpunkte eines Sensors, die auf die Zentrale geschrieben werden sollen. Sensoren haben nämlich einige Datenpunkte, die sich nicht verändern und nur administrativ sind. Zum Beispiel hat jeder Sensor einen Namen wie "Coordinator" oder "Router". Solche Daten sollen nicht auf die Zentrale geschrieben werden und deshalb wird ein Array benötigt, das alle zu schreibnenden Datenpunkte angibt.

Um effektiver zu arbeiten wird für jeden Sensor nur eine Anfrage an "statechange.cgi" gesendet. Wenn man nämlich die Parameter für die Namen der Systemvariablen und deren Werte in der URL jeweils durch Strichpunkte trennt, kann man mehrere Systemvariablen mit nur einem Aufruf von "statechange.cgi" ändern.
Im Programm speichere ich die Parameter für die URL somit in zwei Strings und erweitere den einen in einer Schleife immer um die Namen der Systemvariablen und den anderen um den zu setzenden Wert. Wurden alle zu schreibenden Datenpunkte des Sensors durchlaufen, wird eine Anfrage mit den zwei Strings als Parameter an die Zentrale gesendet. Danach werden die beiden Strings wieder zurückgesetzt und der nächste Sensor wird durchlaufen. Es werden hierbei wie bereits erwähnt nicht alle Datenpunkte durchlaufen, sondern nur die die in dem String-Array enthalten sind.

5.4 Umschreiben der HomeMatic-Systemdateien um ein Gerät hinzuzufügen

Ein Problem, das die gesamte Diplomarbeit begleitet hat, ist, dass es keine Möglichkeit gibt eigene Geräte in der Zentrale zu erzeugen, ohne dass die Zentrale sich direkt mit einem physikalischen Gerät verbindet. Um dieses Problem zu umgehen, werden bei diesem Versuch die Systemdateien von HomeMatic geändert um ein Gerät zu erzeugen, das die MeshSSN-Sensoren anzeigen kann.

5.4.1 Erzeugen eines System-Geräts mit CUxD

Um die Systemdateien ändern zu können, braucht man eine Möglichkeit mit der man Programme auf der Zentrale ausführen kann. Es gibt zwar in HMScript eine Methode "system.Exec", aber diese führt sehr oft zu einem Absturz der Zentrale. Deshalb hat das Add-On CUxD ein eigenes virtuelles Gerät entwickelt, welches man zum Ausführen eines Kommandozeilen-Befehls verwenden kann.
Dieses Gerät kann man wie jedes andere virtuelle Gerät von CUxD über den Browser hinzufügen. Um auf die Benutzeroberfläche von CUxD zu gelangen, wurde in den Einstellungen der HomeMatic-Zentrale bei der Installation von CUxD ein eigener Button "CUxD" eingefügt, der einen zu der Benutzeroberfläche weiterleitet.
Um das Gerät auf der Zentrale hinzuzufügen, muss im Reiter Geräte ein neues Gerät erstellt werden. Wie in der Abbildung unten zu sehen ist, muss man als erstes in dem oberen Dropdown-Menü den Gerätetyp "(28) System" auswählen. Der zweite wichtige Schritt ist, dass man in dem zweiten Dropdown-Menü bei der Funktion "Exec" auswählt. Die restlichen Felder können dann mit beliebigen gültigen Werten ausgefüllt werden. Zuletzt wird mit dem Button "Gerät auf der CCU erzeugen !" das neue Gerät angelegt.

Abbildung 5.2: Die erforderlichen Einstellungen zum Erzeugen eines "(28) System"-Geräts.

Um mithilfe von dem Gerät einen Kommandozeilenbefehl auszuführen, muss man ein HM-Script starten. Dazu wechselt man im Browser bei der HomeMatic-Benutzeroberfläche auf den Menüpunkt "Programme und Zentralenverknüpfungen". Hier drückt man unten den Button "ScriptTesten" und es erscheint ein Fenster, in dem man ein HMScript schreiben kann und zum Schluss auch eine Ausgabe bekommt.
Man könnte den Kommandozeilenbefehl auch direkt über ein Programm starten, oder das Script

in einem Programm speichern und dieses Starten. Leider bekommt man in beiden Fällen kein Ergebnis, wann bzw. ob das Programm erfolgreich durchgeführt wurde. Somit ist es am besten, wenn man das Script über die Test-Funktion von HomeMatic startet.

5.4.2 Ausführen eines Kommandozeilenbefehls mit dem System-Gerät von CUxD

Das HMScript, das den Kommandozeilenbefehl ausführt, ist in dem Code-Ausschnitt unten zu sehen. Der Kommandozeilenbefehl startet ein TCL-Script das in den Systemdateien von HomeMatic ein neues Gerät mit einer bestimmten ID erzeugen soll. Die ID wird hierbei als Argument beim Ausführen des Kommandozeilenbefehls übergeben. Um dieses Argument zu bekommen, muss das HMScript auf eine Systemvariable zugreifen, in der es gespeichert wird. Das passiert beim Ausführen des ersten Befehls vom Code-Auschnitt unten. Mit "dom.GetObject" holt sich das Programm den Wert von der Systemvariable "regadom_script_sensor_id" und schreibt diesen in eine lokale Variable.
Mit dem nächsten Befehl wird der Kommandozeilenbefehl ausgeführt, indem man sich von dem erstellten "System"-Gerät (Es hat hier den Namen "CUxD.CUX2801001") den Datenpunkt "CMD_SETS" holt und dessen Status auf den auszuführenden Kommandozeilenbefehl setzt. Dadurch wird dieser automatisch bereits ausgeführt.
Der nächste Befehl lässt das Programm warten, bis der Kommandozeilenbefehl fertig ist. Zum Schluss wird mit dem letzten Befehl der Rückgabewert des Kommandozeilenbefehls ausgegeben.

```
1 var deviceID = dom.GetObject("regadom_script_sensor_id").State();
2
3 dom.GetObject("CUxD.CUX2801001:1.CMD_SETS").State("tclsh
     /usr/local/addons/sensordata/run.tcl "+deviceID);
4 dom.GetObject("CUxD.CUX2801001:1.CMD_QUERY_RET").State(1);
5 WriteLine(dom.GetObject("CUxD.CUX2801001:1.CMD_RETS").State());
```

Listing 5.20: Das HMScript, welches einen Kommandozeilenbefehl ausführt und dessen Rückgabewert ausgibt.

Erstellen der Systemvariable "regadom_script_sensor_id"

Die Systemvariable "regadom_script_sensor_id" muss händisch erstellt werden, damit sie von dem Kommandozeilenbefehl benutzt werden kann. Dazu muss man sich im Browser mit der CCU2 verbinden und dort unter "Einstellungen" den Menüpunkt "Systemvariablen" auswählen. Hier können nun neue Systemvariablen erstellt werden, oder die Datentypen, Namen oder andere Metadaten von bestehenden Systemvariablen geändert werden. Hier klickt man nun auf den Button "Neu" und legt eine Systemvariable mit dem Namen "regadom_script_sensor_id" als Zeichenkette an. Ich verwende deshalb eine Zeichenkette, damit die Variable als Übergabeparameter im HMScript nicht falsch formatiert wird.
Nachdem die Variable angelegt wurde, muss man noch mithilfe des XML-API-Add-Ons auf den gewünschten Wert setzen. Dazu rufe ich die "statechange.cgi" mit dem Parameterwert "regadom_script_sensor_id" für den Parameter "ise_id" auf und den Parameter "new_value" setze ich auf die ID der Node, die ich hinzufügen will.

5.4.3 “homematic.regadom”

Es wurden bisher öfters die Systemdateien von HomeMatic erwähnt. Bei den Systemdateien geht es eigentlich nur um eine Datei. Diese trägt den Namen “homematic.regadom” und speichert aber die gesamte Konfiguration des Hausautomationssystems. Das Beinhaltet Geräte, Kanäle, Räume, Programme, Favoriten und vieles mehr.
Die Datei wird beim Systemstart geladen und in den Arbeitsspeicher geschrieben. Während die HomeMatic-Zentrale läuft, werden periodisch die Daten aus dem Arbeitsspeicher in die “homematic.regadom” geschrieben. Es gibt auch einen HMScript-Befehl mit dem man die Aktuelle Systemkonfiguration in die Datei laden kann.
Außerdem gibt es für diese Datei ein Backup, welches geladen wird, wenn das Original einen Fehler aufweist. Das kann passieren, wenn die HomeMatic-Zentrale während einem Neustart abstürzt. Denn vor dem Neustart werden noch die aktuellen Konfigurationsdaten des HomeMatic-Systems in die “homematic.regadom” geschrieben. Das hat außerdem zur Folge, dass man die HomeMatic-Zentrale nicht normal beenden darf, nachdem die “homematic.regadom” geändert wurde, weil ansonsten die Änderungen wieder überschrieben werden.

Aufbau der Systemdatei

Die “homematic.regadom” ist in drei große Teile unterteilt:

- Im ersten Teil werden die konkreten Geräte, Programme usw. mit deren Konfiguration in dem Tag “<objmap>” gespeichert.
- Im nächsten Teil werden Administrative Daten in den Tags “<dom-gwcfg>”, “<dom-rtcfg>” und “<dom-webcfg>” gespeichert. Dieser Teil ist für das Hinzufügen von Geräten unrelevant.
- Der letzte teil befindet sich in dem Tag “<dom-root>” In diesem Teil gibt es noch mehrere Listen für alle im ersten Teil stehenden Objekte. Beispielsweise gibt es eine Liste die Referenzen auf alle Geräte vom ersten Teil enthält. Es ist wichtig, dass ein manuell hinzugefügtes Gerät auch in dieser Liste hinzugefügt wird.

Wichtig ist auch noch, dass alle Listen und Maps in der HomeMatic-Zentrale ein Tag “<count>” besitzen, das ebenfalls erhöht werden muss, wenn man einen neuen Eintrag in der Liste erstellt.

Aufbau des "objmap"-Tags

Die "objmap" aus dem ersten Teil enthält selbst mehrere Maps, die alle eine andere Art von Objekt speichern. Beispielsweise speichert das Tag "<devicemap>" alle Geräte, die es im System gibt.
In folgender Liste werden die wichtigsten Objekt-Typen aufgezählt für die es eine eigene Map gibt:

- Kanäle
- Geräte
- Enumerationen, also Listen die Speichern, welche Räume und Gewerke es in dem System gibt. Für jeden Raum und jedes Gewerk wird hier auch noch eine Liste mit den Zugehörigen Geräten gespeichert. Außerdem wird die Liste der Favoriten hier gespeichert.
- Systemvariablen
- Datenpunkte der Kanäle
- Programme
- Interfaces

Die Maps werden außerdem in der aufgezählten Reihenfolge in der "homematic.regadom" gespeichert. Es gibt auch noch weitere Maps, die aber für so spezielle Funktionen verwendet werden, dass sie für das Verständnis des Systems nicht relevant sind.

Jedes Objekt hat zudem eine eigene Objekt-ID. Mit diesen IDs referenziert ein Gerät beispielsweise auf seine Kanäle. Sie werden auch als Referenzen für die Auflistungen im dritten Teil der "homematic.regadom" verwendet. Diese Objekt-ID steht in dem Tag <oid>und steht vor allen Objekten, die in den Maps enthalten sind. Die Objekt-ID muss natürlich eindeutig auf ein Objekt verweisen und es darf somit keine zwei gleichen IDs geben.

Aufbau von einem Kanal-Objekt

Beim hinzufügen eines neuen Kanal-Objekts bzw. eines "channel"-Tags muss dieses folgende Tags enthalten:

- id: Die Objekt-ID, die vor dem Tag steht, wird innerhalb des "channel"-Tags nochmals wiederholt.
- **name**: Name des Kanals, mit dem dieser mit der HMScript-Methode "system.GetObject" gefunden werden kann.
- **type**: Beschreibt Typ des Kanals. – Muss bei unserem Gerät 33 sein.
- **enabled**: Sagt, ob der Kanal läuft. – Muss bei unserem Gerät 1 sein.
- **accessrights**: Sagt, wer Zugriff auf den Kanal hat. - Wird bei allen Kanälen des Systems auf 4294967295 gesetzt.
- **objflgs**: Sichtbarkeit des Kanals. – Muss auf 1 gesetzt werden.
- **metadata**: Hier sind die Gerätebeschreibung, Links mit anderen Kanälen und Parameter-Set-Beschreibungen vermerkt. Es wird auch vermerkt, welche Parameter-Sets es gibt ("LINK", "MASTER" und/oder "VALUES" – Die Beschreibung des "VALUES"-Parameter-Sets wird direkt aus den Beschreibungen der Datenpunkte erzeugt und wird somit nicht extra bei diesem Tag aufgelistet.)
- **channel-info**: bleibt leer
- **chntype**: Eine zweite Typ-einteilung des Kanals. – Muss bei unserem Gerät 22 sein.
- **upnp**: Wird bei allen Kanälen des Systems auf 0 gesetzt.
- **deviceid**: Referenziert zu dem Gerät zu dem der Kanal gehört.
- **channel-dps**: Speichert eine Liste von Datenpunkten, die zu dem Kanal gehören.
- **chnparams**: Ein Parameter-Set welches bei allen Kanälen des Systems leer gelassen wird.
- **chnadr**: Adresse des Kanals, mit der dieser Beispielsweise über die XMLRPC-Schnittstelle gefunden wird.
- **chnlabel**: Ein Name, der auf die Funktion des Kanals hinweist.
- **dpid-read**, **dpid-write** und **dpid-event**: Sind alle drei auf 65535 gesetzt. Bei Manchen Kanälen sind sie auf die Objekt-ID eines bestimmten Datenpunktes gesetzt.
- **ready**: Muss bei unserem Gerät 1 sein. Wird bei ausgeblendeten Kanälen auf 0 gesetzt.

Aufbau von einem Geräte-Objekt

Beim hinzufügen eines neuen Geräte-Objekts bzw. eines “device”-Tags muss dieses folgende Tags enthalten (Tags die bereits beim Kanal-Objekt erklärt wurden werden hier nicht nochmal erklärt):

- **id**
- **name**
- **type**: Muss bei unserem Gerät 17 sein.
- **enabled**
- **accessrights**
- **objflgs**
- **metadata**: Wie beim Kanal, nur dass ein Gerät nur ein “MASTER”-Parameter-Set enthalten kann. Außerdem Speichert zusätzlich ein “property”-Tag “operateGroupOnly”, dessen Wert auf “false” gesetzt ist.
- **device-info**: bleibt leer
- **device-chns**: Speichert eine Liste von Kanälen, die zu dem Gerät gehören.
- **devtype**: Eine zweite Typ-einteilung des Geräts. – Muss bei unserem Gerät 0 sein.
- **devparams**: wie “chnparams”
- **devadr**: wie “chnadr”
- **devifc**: Hier wird das Interface angegeben, das für das Gerät zuständig ist. Bei unserem Gerät schreiben wir hier die Objekt-ID des Interfaces für CUxD-Geräte hin.
- **devlavel**: wie “chnlabel”
- **dev2ready**: Muss bei unserem Gerät 1 sein.

Aufbau von einem Datenpunkt-Objekt

Beim hinzufügen eines neuen Datenpunktes-Objekts bzw. eines “hssdp”-Tags muss dieses folgende Tags enthalten (Tags die bereits beim Kanal-Objekt oder Device-Objekt erklärt wurden werden hier nicht nochmal erklärt):

- **id**
- **name**
- **type**: Muss bei unserem Gerät 393281 sein
- **enabled**
- **accessrights**
- **objflgs**

- **metadata**: In den Metadaten eines Datenpunkts wird beschrieben, welche Werte der Datenpunkt maximal und minimal annehmen darf und welchen Typ und Einheit der Datenpunkt hat. Mit der Eigenschaft "TAB_ORDER" wird außerdem beschrieben, als wievielter dieser Datenpunkt im Browser angezeigt werden soll. Dadurch kann benutzerdefiniert bestimmt werden, in welcher Reihenfolge die Datenpunkte eines einzelnen Kanals in der HomeMatic-GUI angezeigt werden sollen.
- **dp-info**: bleibt leer
- **chnid**: Referenziert zu dem Kanal zu dem der Datenpunkt gehört.
- **valtype**: Muss bei unserem Gerät 4 sein.
- **alvalev**: Muss bei unserem Gerät 1 sein.
- **cachtmout**, **dvi**: Muss bei unserem Gerät 0 sein.
- **op**: Muss bei unserem Gerät 6 sein. Mit dieser Einstellung wird dem System gesagt, dass dieser Datenpunkt editierbar sein soll. Dass ihn also nicht nur das Gerät ändern können soll.
- **subtype**, **polltime**: Muss bei unserem Gerät 0 sein.
- **valdev**: Hier wird der Wert des Datenpunktes gespeichert. Außerdem wird noch ein "type" definiert, der bei unserem Gerät 4 sein muss.
- **hss-adr**: Hier wird die Adresse des Kanals zu dem der Datenpunkt gehört gespeichert.
- **hss-id**: Hier wird der Name des Datenpunkts gespeichert.

5.4.4 TCL-Script zum Umschreiben der "homematic.regadom"-Datei schreiben

Um ein MeshSSN-Gerät in die "homematic.regadom" zu schreiben, muss das TCL-Script drei verschiedene Objekte erzeugen. Zuerst muss ein Gerät erzeugt werden. Dieses Gerät verweist dann auf einen Kanal und dieser Kanal hat wiederum acht Datenpunkte die die einzelnen Datenpunkte der MeshSSN-Sensoren speichern. Im TCL-Script werden aber zuerst der Kanal, dann das Gerät und dann die Datenpunkte geschrieben, da die Kanal-Map als erstes in der "homematic.regadom" steht.
Bevor die "homematic.regadom" aber geschrieben wird, wird sie noch eingelesen. Hierzu wird diese im Lese-Modus geöffnet und alle Daten zeilenweise in ein String-Array geschrieben. Außerdem holt man sich noch ein Kommandozeilenargument, das sagt welche ID das hinzuzufügende Gerät haben soll. Der gesamte schreibvorgang muss zudem zweimal durchgeführt werden, weil es ja auch noch ein Backup von der "homematic.regadom" gibt. Damit die beiden Dateien beschrieben werden können, müssen sie im Schreibmodus geöffnet werden. Beim öffnen wird dann der gesamte Inhalt der Dateien gelöscht.

Grundsätzlich geht das Programm die eingelesende Datei zeilenweise durch und schreibt jede Zeile in die "homematic.regadom". Dies passiert so lange, bis ma zu einer Zeile kommt, an der man etwas ändern muss. Da sich dieser Teil immer wieder wiederholt, wurde dafür eine eigene Methode namens "writeuntilstring" geschrieben.
Wie unten in dem Code-Ausschnitt zu sehen ist, hat die Methode vier Parameter. Der erste ist die Datei in die geschrieben werden soll. Der zweite Parameter ist ein String-Array, welches die eingelesenen Zeilen von der "homematic.regadom" enthält. Der dritte Parameter ist die Zeile, bei der zu schreiben begonnen werden soll und der letzte Parameter ist der String mit dem jede zu schreibende Zeile verglichen wird.
In einer Schleife wird eine Zeile nach der anderen durchlaufen und in die zu schreibende Datei geschrieben. Abgebrochen wird die Schleife, sobald die aktuelle Zeile dem letzten Parameter entspricht oder wenn das Ende der eingelesenen Datei erreicht wurde. Zum Schluss wird die aktuell Zeilennummer an den Aufrufer zurückgegeben, damit dieser die Zeilennummer aktualisieren kann.

```
1 proc writeuntilstring {p_out p_file_data p_line_num p_abort_string} {
2    while {[string match [lindex $p_file_data $p_line_num] $p_abort_string] ==
        0 && $p_line_num < [llength $p_file_data]} {
3       puts $p_out [lindex $p_file_data $p_line_num]
4       incr p_line_num
5    }
6    return $p_line_num
7 }
```

Listing 5.21: Diese Methode Schreibt Zeilen in eine Datei, bis eine Zeile einem bestimmten String entspricht.

Zuerst schreibt man also alle Daten aus der eingelesenen "homematic.regadom" in die neue "homematic.regadom", bis man den Anfang der Kanal-Map erreicht. Dieser ist gekennzeichnet durch das Tag "<channelmap>". In der ersten Zeile der Map befindet sich ein Counter-Tag das man nicht unverändert übernehmen darf. Diesen Counter muss man um 1 erhöhen da wir ja genau einen Kanal hinzufügen. Dann geht man bis zum Ende der Kanal-Map um dort den neuen Kanal hinzuzufügen.

An dieser Stelle ist zu erwähnen, dass das TCL-Script Objekt-IDs im Zahlenraum von 6000 bis 6999 vergibt. Die "6" von den beiden Zahlen wird hierbei statisch vergeben. Die Mittleren beiden Ziffern entsprechen der ID die der Sensor haben soll. Dieser Wert wird als Argument beim Kommandozeilenbefehl übergeben. Die Einerstelle wird anhand des Objekt-Typs definiert. Der Kanal hat an der Einerstelle 0 stehen, das Gerät eine 1 und die Datenpunkte bekommen bei der Einerstelle die Ziffern 2 bis 9 vergeben.

Beim Kanal wird unter anderem die eigene Objekt-ID dynamisch gesetzt. Die Referenzen auf das Gerät, zu dem der Kanal gehört, und die Datenpunkte, die zu dem Kanal gehören, werden ebenfalls anhand der Sensor-ID gesetzt. Außerdem wird der Name des Kanals abhängig von der Sensor-ID gesetzt.
Als nächstes wird der gesamte Vorgang mit der Geräte- und der Datenpunkte-Map wiederholt. Bei beiden werden alle Objekt-IDs, Referenzen und Namen automatisch bezogen.
Bei der Datenpunkte-Map muss zudem der Counter um 8 erhöht werden, weil acht Datenpunkte hinzugefügt werden. Hierzu werden zusätzlich in einem Array die Namen der acht Datenpunkte wie zum Beispiel "HUMIDITY" gespeichert. Dann durchläuft das Programm das Array in einer

Schleife und schreibt in jedem Durchlauf einen neuen Datenpunkt. Somit wird auch die Einerstelle der Object-IDs der Datenpunke dynamisch von der Durchlaufvariable bezogen. Außerdem bekommt jeder Datenpunkt einen eigenen Namen, welcher aus dem Array geschrieben wird. Ein Beispielname für einen Datenpunkt wäre dann "MeshSSNNode00:0.HUMIDITY".

Nachdem alle benötigten Objekte hinzugefügt wurden, muss man noch Referenzen auf diese Objekte beim dritten Teil der "homematic.regadom" hinzufügen. Dazu schreibt man zuerst alle Zeilen, bis man zu einer Zeile kommt die mit dem String "<root>" übereinstimmt. Dann durchläuft man die ersten drei Aufzählungen, welche mit einem "<enel>"-Tag gekennzeichnet sind durch, erhöht dort wieder die Counter und fügt neue Einträge mit den jeweiligen Referenzen auf die neuen Objekte ein. Bei der ersten Aufzählung werden die Geräte aufgelistet, bei der zweiten die Kanäle und bei der dritten die Datenpunkte.
Zum Schluss schreibt man noch den Rest der eingelesenen Datei in die "homematic.regadom" bzw. deren Backup. Damit ist das Ändern der Systemdateien abgeschlossen. Jetzt muss man die Zentrale noch zum absturz bringen, damit sie die Dateien nicht überschreibt und beim erneuten Starten des Programms wird nun auch das neue Gerät geladen und wie in der Abbildung unten zu sehen ist, auch in der Zentrale angezeigt.

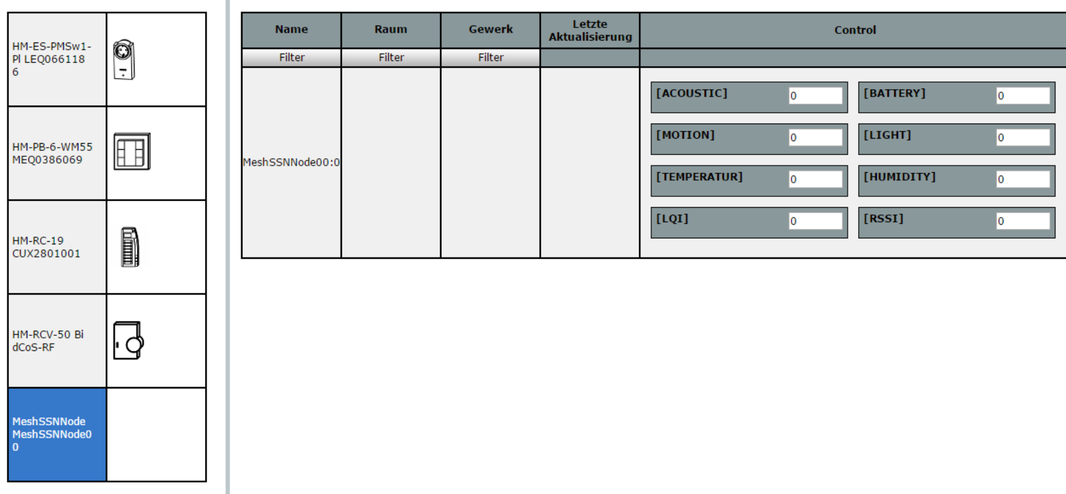


Abbildung 5.3: Anzeige des hinzugefügten Geräts in der HomeMatic-Benutzeroberfläche

5.4.5 Das Verhalten des neuen Geräts

Obwohl das Gerät zwar in der Zentrale angezeigt wird, ist es während der Laufzeit leider nicht möglich die Daten des Geräts zu ändern.
Das Funktioniert deshalb nicht, weil das Gerät keinem kompatiblen Interface zugeordnet ist. Ein Interface ist für die Verbindung von dem Gerät auf der Zentrale mit dem physikalischen Gerät zuständig. Außerdem verweist es auf einen XMLRPC-Server der zum Ändern von Gerätedaten verwendet werden soll.
Es gibt beispielsweise ein Interface für die standard-Geräte von HomeMatic und es gibt ein Interface von CUxD für externe Geräte. Wenn man das Interface von CUxD verwendet, wird das neue Sensor-Gerät zumindest angezeigt. Bei anderen Interfaces wird es gar nicht erkannt.
Das Problem mit den Interfaces ist, dass man kein eigenes implementieren kann, weil es sich um einen systeminternen Server handelt bei dem man nicht weiß, auf welche Daten er zurgreift und für was er alles verantwortlich ist. Dazu müsste man herausfinden, wo die Implementierung dieses Servers liegt.

Es ist auch gar nicht Sicher, dass es an dem Interface liegt, dass das Gerät nicht geändert werden kann. Es könnte auch sein, dass die Zentrale anderswo auch noch Gerätedaten speichert und weil diese Fehlen, entscheidet die, dass das Gerät nicht existiert.

Beurteilung

6.1 App

Die Smartphone Applikation ermöglicht es dem Benutzer, mithilfe eines sehr einfachen Interfaces, die wichtigsten Daten der verbundenen Sensoren auszulesen. Geplant war eigentlich, dass es möglich ist, die verbundenen Sensoren beziehungsweise Datenpunkte mithilfe der XML-RPC Verbindung zur CCU2 abzufragen. Allerdings gibt es laut der HomeMatic XML-RPC Dokumentation keine Methode die dies ermöglicht. Über die Methode *listDevices* der Schnittstelle, sollte dies zwar theoretisch möglich sein, jedoch besteht für die Unzahl von Rückgabewerten dieser Methode keine richtige Dokumentation von Seiten HomeMatic.
Die weitere große Herausforderung war der Umgang mit vollkommen neuen Technologien, denn die Verwendung von Ionic in Verbindung mit Angular ist ein völlig anderes Konzept, als jenes der Objektorientierten Programmiersprachen.

6.2 Server

Man kann die Funktionalität des Servers mithilfe der Konfigurationsdatei sehr leicht anpassen und er funktioniert auch, Wenn keine Datenbank oder oder HomeMatic-Zentrale vorhanden ist. Er schreibt die Sensordaten dann immer noch zu der jeweils anderen Schnittstelle.
Zur Zentrale kann der Server die Sensordaten auf drei verschiedene Arten schreiben, wobei das Schreiben in Systemvariablen die bestmögliche Variante ist. Dadurch können nämlich auch Programme abhängig von dem Wert der Systemvariablen gestartet werden. Das Ändern von Systemvariablen ist die Vorgehensweise, die auch von HomeMatic so vorgesehen ist.
Will man die Sensordaten in einem eigenen Gerät speichern, muss man ein bisschen Tiefer ins System eingreifen. Bei meinem Versuch, die Systemdateien so umzuschreiben, dass ein neues Gerät hinzugefügt wird, bin ich leider an dem Problem gescheitert, dass HomeMatic ein physikalisches Gerät kennen muss, mit dem es Kommunizieren kann.
Will man die Sensordaten trotzdem noch in einem Gerät anzeigen lassen, gäbe es noch die Möglichkeit mit CUxD ein Gerät zu erzeugen und dieses in der "homematic.regadom" umzuschreiben. Vielleicht lässt CUxD es dann zu, dass Daten über XMLRPC geändert werden. Höchstwahrscheinlich muss man aber vorher noch in CUxD selbst die Systemdaten ändern, damit es mit dem veränderten Gerät kompatibel ist.
Vermutlich kann man aber nur auf ein Update warten, mit dem man beispielsweise in CUxD ein frei konfigurierbares, virtuelles Gerät hinzufügen kann. Bisher bildet CUxD nur eine Schnittstelle für Geräte von anderen Hausautomationssystemen, damit diese ebenfalls direkt zum HomeMatic-System hinzugefügt werden können. Aber mit dem "System"-Gerät kann es auch schon komplett virtuelle Geräte erzeugen.
Ansonsten gibt es noch die Möglichkeit, dass man auf einen Openhab- oder FHEM-Server umsteigt und dort nach einem Weg sucht, das HomeMatic-System und die Sensoren hinzuzufügen.

Literaturverzeichnis

[AMP16] APACHE-MAVEN-PROJECT: *What is Maven?* https://maven.apache.org/what-is-maven.html, 2016. – Zuletzt besucht: 23.3.2016

[Cou11] COUCHBASE: *Comparing document-oriented and relational data.* http://docs.couchbase.com/developer/dev-guide-3.0/compare-docs-vs-relational.html, 2011. – Zuletzt besucht: 12.3.2016

[Cou12] COUCHDB: *Technical Overview.* http://wiki.apache.org/couchdb/Technical%20Overview, 2012. – Zuletzt besucht: 12.3.2016

[Cou16] COUCHDB: *CouchDB - A Database for the Web.* http://couchdb.apache.org/, 2016. – Zuletzt besucht: 12.3.2016

[DOL16] DATENBANKEN-ONLINE-LEXIKON: *NoSQL.* http://lwibs01.gm.fh-koeln.de/wikis/wiki_db/index.php?n=Category.NoSQL, 2016. – Zuletzt besucht: 8.3.2016

[FH16] FH-HAGENBERG: *Mesh-Sensornetzwerk.* http://sensornetzwerk.hagenberg.servus.at/, 2016. – Zuletzt besucht: 16.3.2016

[Goo16] GOOGLE: *Angular Documentation.* https://docs.angularjs.org/guide, 2016. – Zuletzt besucht: 24.03.2016

[Hom13] HOMEMATIC: *JSON-RPC API.* https://sites.google.com/site/homematicplayground/api/json-rpc, 2013. – Zuletzt besucht: 22.3.2016

[Hom16] HOMEMATIC: *HomeMatic XML-RPC-Schnittstelle.* http://www.eq-3.de/Downloads/Software/HM-CCU2-Firmware_Updates/Tutorials/HM_XmlRpc_API.pdf, 2016. – Zuletzt besucht: 28.3.2016

[Ion15] IONIC: *Start building with Ionic!* http://ionicframework.com/getting-started/, 2015. – Zuletzt besucht: 21.03.2016

[Ion16] IONIC: *Ionic Documentation.* http://ionicframework.com/docs/, 2016. – Zuletzt besucht: 01.04.2016

[Lan15] LANGHAMMER, Uwe: *CUx-Daemon Dokumentation.* http://www.ehomeportal.de/downloads/cuxd/cuxd-handbuch.pdf, 2015. – Zuletzt besucht: 25.3.2016

[Lun11] LUNDGREN, Henrik: *Ektorp Reference Documentation.* http://ektorp.org/reference_documentation.html, 2011. – Zuletzt besucht: 22.3.2016

[NPA12] NPA: *.NET Persistence API.* http://www.npersistence.org/overview, 2012. – Zuletzt besucht: 19.3.2016

[Sas15] SASS: *SASS Documentation.* http://sass-lang.com/guide, 2015. – Zuletzt besucht: 25.03.2016

[Sch15] SCHILLER, B.: *BidCos - Funkprotokoll fuer die Hausautomation.* smarthomewelt.de/bidcos-funkstandard-eq-3-hausautomation, 2015. – Zuletzt besucht: 23.03.2016

[Uga12] UGAYA40: *MVVM-Konzept*. https://de.wikipedia.org/wiki/Model_View_ViewModel/media/File
2012. – Zuletzt besucht: 03.04.2016

[Wik15] WIKIMATIC: *HomeMatic Wiki*. http://www.wikimatic.de/wiki/Hauptseite, 2015. – Zuletzt besucht: 23.3.2016

[Wil16] WILLIAMS, Chris: *Node Serialport*. https://github.com/voodootikigod/node-serialport, 2016. – Zuletzt besucht: 20.3.2016

[Win03] WINER, Dave: *XML-RPC Specification*. http://xmlrpc.scripting.com/spec.html, 2003. – Zuletzt besucht: 26.03.2016

Abbildungsverzeichnis

2.1 Model-View-ViewModel Entwurfsmodell [Uga12] 14
2.2 Vergleich einer relationalen und Dokumentenbasierten Datenbank [Cou11] 20
2.3 Futon ist die Webbasierte Schnittstelle zu CouchDB 25
2.4 Auswahl des Wizards zur erstellung eines Maven-Projekts 27
2.5 Suche nach einer Maven Bibliothek 28
2.6 Liste der in Eclipse standartmäßig verfügbaren Buildprozesse 28
2.7 Der Aufbau der Java Persistence API und wie diese die Verbindung zur Datenbank regelt. [NPA12] 29
2.8 Der Aufbau der Masterarbeit MeshSSN [FH16] 35
2.9 In dem C#-Programm werden die Sensoren in einer Baumstruktur angezeigt. . . 36
2.10 Ansicht der installierten Add-Ons von Homematic 38
2.11 Das Grundsätzliche Klassenmodell von HomeMatic 41
2.12 Das interne und externe Modell von HomeMatic-Geräten 44

3.1 Die Konfigurationsdatei des Servers. 50
3.2 Überblick der App-Oberflächen 52

5.1 Das Klassendiagramm für den Server, der die Sensordaten überträgt. 72
5.2 Die erforderlichen Einstellungen zum Erzeugen eines “(28) System”-Geräts. . . . 84
5.3 Anzeige des hinzugefügten Geräts in der HomeMatic-Benutzeroberfläche 92

Quellcodeverzeichnis

2.1 Verschachtelungen in SCSS 13
2.2 Definition von Variablen in SCSS 14
2.3 Definition vonm Mixins in SCSS 14
2.4 Erstellen neuer Direktiven 15
2.5 Kommandozeilenbefehle zur Installation von Ionic[Ion15] 17
2.6 Ein curl-Aufruf mit dem man in CouchDB eine Datenbank erstellt 24
2.7 Ein curl-Aufruf mit dem man in einer Datenbank von CouchDB ein Dokument erstellt 24
2.8 Ein einfaches NodeJS-Programm in dem ein http-Server erstellt wird. 33
2.9 Beispiel für einen Request mit XML-RPC 42
2.10 Beispiel für ein Antwort mit XML-RPC 43
2.11 Ein kleines Beispiel für die Syntax von TCL-Skript 47
5.1 Ausschnitt App-Routing 63
5.2 Persistierung der App-Einstellungen mittels Local Storage 63
5.3 Verbindung zur CCU2 Zentrale mithilfe der jsxmlrpc-Bibliothek 64
5.4 Dynamisches Generieren der Detailansicht 66
5.5 Kommandozeilenbefehl für die Installation des Serialport-Moduls 67
5.6 Kommandozeilenbefehl für die Installation des Serialport-Moduls 67
5.7 Fehlermeldung, wenn man einen unbesetzten COM-Port übergeben hat. 68
5.8 Befehl zum Neustart des Windows-Systems mit zusätzlichen Optionen 69
5.9 Konfiguration zum Generieren einer Ausfürhbaren Jar-Datei 70
5.10 Konfiguration zum Generieren einer Jar-Datei mit integrierten Bibliotheken . . . 70
5.11 Tag, welches einstellt, wann das Assembly-Plugin ausgeführt werden soll. 71
5.12 Bei diesem Code wird je nach Konfiguration eine Bestimmte Instanz der Klasse HomematicWriter erstellt. 73
5.13 Das Singleton-Design-Pattern in der ProjectConfiguration-Klasse 74
5.14 Die standardmäßige Konfiguration der Konfigurationsdatei des Servers. 75
5.15 Die von Ektorp generierte Callback-Methode einer View. 76
5.16 Der Konstruktor eines Repositories in Ektorp. 77
5.17 Ein XMLRPC-Methodenaufruf mit einer Bibliothek von Java. 80
5.18 Ein Aufruf von XML-API, der zwei Systemvariablen ändert. 80
5.19 Der in “statechange.cgi” verwendete String, welcher in ein HMScript umgewandelt wird. 82
5.20 Das HMScript, welches einen Kommandozeilenbefehl ausführt und dessen Rückgabewert ausgibt. 85
5.21 Diese Methode Schreibt Zeilen in eine Datei, bis eine Zeile einem bestimmten String entspricht. 91