Höhere Technische Bundeslehranstalt für Informatik, Perg

Diplomarbeit

Mobile Applikation für die Wartung von Stranggießanlagen

Bauteilverwaltung mit NFC Unterstützung für Android und iOS

eingereicht von:	Lukas Bindreiter <lukas.bindreiter@outlook.com> Michael Hader <michael.hader@outlook.com> David Haunschmied <david.haunschmied@hotmail.com>
eingereicht am:	21. Oktober 2015
Betreuer:	Dipl-Ing. Dr. Michael Buchberger
In Zusammenarbeit mit:	Primetals Technologies Austria GmbH Josef Bogner

Eidesstattliche 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	Lukas Bindreiter
Ort, Datum	Michael Hader
Ort, Datum	David Haunschmied

Danksagung

Wir bedanken uns bei all jenen, die uns bei unserer Diplomarbeit in jeglicher Hinsicht unterstützt haben.

Besonderer Dank gilt unserem Betreuungslehrer Dipl.-Ing. Dr. Michael Buchberger für seine Unterstützung. Sowohl bei technischen Fragen als auch bei der Verfassung dieser Arbeit war er uns aufgrund seiner großen Erfahrung eine große Hilfe.

Weiters möchten wir uns bei Josef Bogner, unserer Kontaktperson der Firma Primetals, bedanken. Er stand uns bei Fragen immer zur Seite und hat auch dafür gesorgt, dass wir eine Stranggießanlage besichtigen durften. Weiterer Dank gilt Dipl.-Ing. Richard Kainerstorfer, der uns diese Arbeit vermittelt hat und ohne den sie daher nie zustande gekommen wäre.

Inhaltsverzeichnis

1 Einführung **11**
- 1.1 Kurzfassung . . . 11
- 1.2 Abstract . . . 12
- 1.3 Auftraggeber . . . 13
- 1.4 Gliederung . . . 14

2 Grundlagen **15**
- 2.1 Stranggießanlage . . . 15
  - 2.1.1 Ablauf des Stranggießverfahrens . . . 15
- 2.2 Vorhandenes EQX System . . . 17
- 2.3 TypeScript . . . 19
  - 2.3.1 Typen . . . 19
  - 2.3.2 Klassen . . . 20
  - 2.3.3 Module . . . 21
  - 2.3.4 Vorteile gegenüber JavaScript . . . 22
- 2.4 AngularJS . . . 24
  - 2.4.1 Single-Page Webapplikationen . . . 24
  - 2.4.2 Model-View-Controller Architektur . . . 24
  - 2.4.3 Direktiven . . . 24
  - 2.4.4 Beispiel . . . 27
  - 2.4.5 Benutzerdefinierte Direktiven . . . 28
  - 2.4.6 Services . . . 29
  - 2.4.7 Konfigurationen . . . 30
  - 2.4.8 Dependency Injection . . . 30
- 2.5 Apache Cordova . . . 31
  - 2.5.1 Übersicht . . . 31
  - 2.5.2 Architektur . . . 31
- 2.6 Near Field Communication (NFC) . . . 35
  - 2.6.1 Funktionsweise von NFC . . . 35
  - 2.6.2 NFC Tags . . . 36
- 2.7 SASS, SCSS . . . 39
  - 2.7.1 Variablen . . . 39
  - 2.7.2 Verschachtelung . . . 40
- 2.8 Bootstrap . . . 41

2.8.1 Responsive Web Design . . . 41
2.9 Build Tools . . . 42
2.9.1 Package Manager . . . 42
2.9.2 Task Runner . . . 42
2.9.3 Gradle . . . 42
2.10 JSON . . . 44
2.10.1 Beispiel . . . 44
2.11 RESTful Webservice . . . 45
2.12 CORS . . . 46
2.13 Spring Framework . . . 47
2.13.1 Dependency Injection . . . 47
2.14 SockJS . . . 49
2.15 STOMP . . . 50
2.15.1 Message Broker . . . 50
2.16 LDAP . . . 51

3 Benutzung **53**
3.1 Übersicht . . . 53
3.2 Hauptmenü . . . 55
3.3 Inventaransicht . . . 56
3.4 Baumansicht der Stranggießanlage . . . 58
3.5 Detailansicht eines Bauteils . . . 60
3.6 Dokumente eines Bauteils . . . 61
3.7 NFC Bauteilerkennung . . . 62

4 Programmstruktur **63**
4.1 Architektur . . . 63
4.2 UML Diagramm der Bauteil Klassenstruktur . . . 65
4.3 Schnittstellen . . . 67
4.3.1 REST Webservice . . . 67

5 Implementierung **73**
5.1 Technologieauswahl . . . 73
5.2 Serverseitig . . . 74
5.2.1 Ablauf der HTTP Request Beantwortung . . . 74
5.2.2 HTTP Routenzuordung . . . 76
5.2.3 JSON Übertragung . . . 78
5.2.4 Service Klassen . . . 83
5.2.5 HTTP Request Filter . . . 85
5.2.6 Repositories . . . 89
5.2.7 Konfigurationsdateien . . . 92
5.2.8 SockJS Server . . . 94
5.2.9 Einheitenumrechnung . . . 95
5.3 Clientseitig . . . 102
5.3.1 Angular Modulstruktur . . . 102

5.3.2 Komponenten . . . 104
5.3.3 Services . . . 105
5.3.4 Konfigurationen . . . 105
5.3.5 Data Transfer Objects . . . 107
5.3.6 SockJS Client . . . 108
5.4 Cordova-Projekt . . . 109
5.4.1 Erzeugen eines Cordova-Projektes . . . 109
5.4.2 Aufbau und Konfiguration . . . 109
5.4.3 Plattformen . . . 111
5.4.4 Verwendete Plugins . . . 111
5.4.5 Kompilierung . . . 112
5.4.6 Ausführen der Applikation . . . 112
5.5 NFC Plugin . . . 113
5.5.1 Ausgangssituation . . . 113
5.5.2 Entwicklung des NFC Plugins . . . 113

6 Beurteilung **119**

Abbildungsverzeichnis **120**

Ausschnittsverzeichnis **122**

Literaturverzeichnis **125**

Kapitel 1

Einführung

1.1 Kurzfassung

EQX Mobile ist eine mobile HTML5 Applikation für Android und iOS. Die Applikation stellt eine mobile Alternative zu einer bereits existierenden Desktop Applikation für die Verwaltung von Bauteilen einer Stranggießanlage dar. Neben der manuellen Auswahl und Anzeige von Bauteilen einer Stranggießanlage ermöglicht die Applikation auch die Identifizierung von Bauteilen mittels Near Field Communication (NFC). Dazu werden alle Bauteile mit NFC-Tags versehen, die von mobilen Geräten gelesen werden können. Zu den Bauteilinformationen gehören unter anderem ein Name, ein Bauteiltyp, zugeordnete Dokumente sowie diverse Einsatzzeitenzähler und Parameter mit zusätzlichen Informationen. All diese Eigenschaften können sowohl angezeigt als auch modifiziert werden. Außerdem ermöglicht die *EQX Mobile* Applikation die Registrierung neuer Bauteile. Des Weiteren können aktuelle Fotos von Bauteilen direkt mit dem mobilen Gerät aufgenommen und einem Bauteil zugeordnet werden. Die Applikation ist außerdem auf einen Mehrbenutzerbetrieb ausgelegt, das bedeutet, dass Änderungen, die von einem Benutzer durchgeführt werden, sofort bei allen anderen Benutzern sichtbar werden.

Die mobile Applikation soll den Arbeitsablauf der Wartung von Bauteilen für Mitarbeiter einer Stranggießanlage erleichtern. Die existierende Desktop Applikation ist nur auf vordefinierten Plätzen wie Leitstand und Wartungswerkstätte installiert und Informationen zu Bauteilen sind demnach nur dort einsehbar. Mit der mobilen Applikation können die Bauteilinformationen direkt vor Ort abgefragt und verändert werden. Außerdem werden auch Benachrichtigungen, wie etwa Warnungen, dass Bauteile bald wieder gewartet werden müssen, ebenfalls in der mobilen Applikation angezeigt. Eine Benutzerverwaltung ermöglicht es für unterschiedliche Benutzer unterschiedliche Zugriffsrechte zu vergeben.

1.2 Abstract

EQX Mobile is a mobile HTML5 application for Android and iOS. The application is a mobile alternative for an already existing desktop application, which is used to manage equipment of a continuous casting plant. Beside the manual selection of equipment parts the app is able to recognize equipment parts via Near Field Communication (NFC) and then display basic data of those. In order to use the NFC recognition, there is an NFC tag attached to every equipment part. Basic equipment data includes a name, an equipment type, documents assigned to the equipment as well as diverse lifetime counters and parameters containing additional information. All those properties can be both read and modified. Furthermore the *EQX Mobile* App supports the registration of recently installed components. It is also possible to take a current picture of an equipment part using the mobile device's camera and easily assign it to that equipment part. Aside from that, the app also provides multi-user support, which means that modifications carried out by one user are automatically updated for all other users.

The mobile application is designated to simplify the maintenance workflow. While the already existing desktop application is only available at fixed locations such as maintenance workshop and operator pulpit the new mobile application provides access to the equipment information anywhere. Another feature is the display of push-notifications such as a reminder about an upcoming maintenance. A user management allows to define different access rights for different users.

1.3 Auftraggeber

Diese Diplomarbeit wurde in Zusammenarbeit mit dem Unternehmen Primetals Technologies [Tec16] entwickelt. Primetals Technologies ist ein Gemeinschaftsunternehmen von Mitsubishi Hitachi Metals Machinery und Siemens und im Bereich Anlagenbau für die Metallindustrie tätig. Unter anderem entwickelt das Unternehmen Level 2 Software (Software für Prozessoptimierung) für Stranggießanlagen. Hierfür ist *EQX Mobile* ein Zusatzangebot für Kunden, welche die Desktop Applikation Equipment Expert von Primetals Technologies verwenden.

1.4 Gliederung

Diese Arbeit ist in sechs Kapitel untergliedert, deren Inhalt hier kurz erläutert wird. Kapitel 1, Einführung, enthält eine Kurzfassung der Arbeit sowohl in deutscher als auch englischer Sprache, sowie eine kurze Beschreibung des Auftraggebers. Kapitel 2, Grundlagen, erläutert die Funktionsweise von Stranggießanlagen und enthält zusätzlich Erklärungen zu Technologien, die in dieser Arbeit verwendet wurden. Im Kapitel Benutzung wird das Endresultat dieser Arbeit vorgestellt und mithilfe von Screenshots die verschiedenen Arbeitsabläufe vorgestellt. Kapitel 4, Programmstruktur, bietet einen allgemeinen Überblick über die Architektur vom Benutzer bis zum EQX Server, sowie eine Übersicht über das Datenmodell. In Kapitel 5, Implementierung, werden sowohl serverseitige als auch clientseitige Implementierungsdetails angeführt und spezielle Problemlösungen behandelt. Abschließend befindet sich in Kapitel 6, Beurteilung, ein Rückblick auf die Erfahrungen, die bei der Erstellung dieser Arbeit gewonnen wurden.

Kapitel 2

Grundlagen

2.1 Stranggießanlage

Stranggießanlagen werden in der Stahlerzeugungsindustrie eingesetzt. Sie dienen zur Aushärtung des noch flüssigen Stahls. Dabei wird der vorher erzeugte flüssige Stahl vergossen und erstarrt. Der vergossene Stahl durchläuft danach noch weitere Stufen der Verarbeitung und Behandlung bis die endgültigen Formen beziehungsweise Eigenschaften des gewünschten Stahls erzielt wurden. [Eis99] [Sta16]

2.1.1 Ablauf des Stranggießverfahrens

In Abbildung 2.1 ist der vereinfachte Aufbau der Bauteile einer Stranggießanlage zu sehen. Zudem ist der Ablauf vom flüssigen bis zum erstarrten Stahl beschrieben. Beim Stranggießen gelangt der flüssige Stahl aus der Gießpfanne über einen Zwischenbehältern (Verteiler) mit regulierbarem Ausguss unter Luftabschluss in die kurze, wassergekühlte Kokille. Die Form der Kokille bestimmt die Form des Stranges. Vor dem Gießbeginn wird der Boden der Kokille mit dem sogenannten Kaltstrang verschlossen. Ist die vorgeschrieben Badspiegelhöhe erreicht, wird die Kokille in Schwingung versetzt, damit der Stahl nicht daran haftet. Der am Rand erstarrte rotglühende Stahl wird zunächst mithilfe des Kaltstrangs und danach durch die Treibrollen aus der Kokille gezogen. Der Strang muss aufgrund seines flüssigen Kerns solange sorgfältig mit Wasser bespritzt und gekühlt sowie allseits von Rollen unterstützt werden, bis er vollständig erstarrt ist. Damit wird verhindert, dass der noch dünne erstarrte Rand aufbricht. Nachdem der Strang vollständig erstarrt ist, kann er durch Schneidbrenner oder Scheren auf gewünschte Längen zerteilt werden. Die intensive Kühlung bewirkt, dass der Strang gleichmäßig erstarrt. [Eis99] [Sta16]

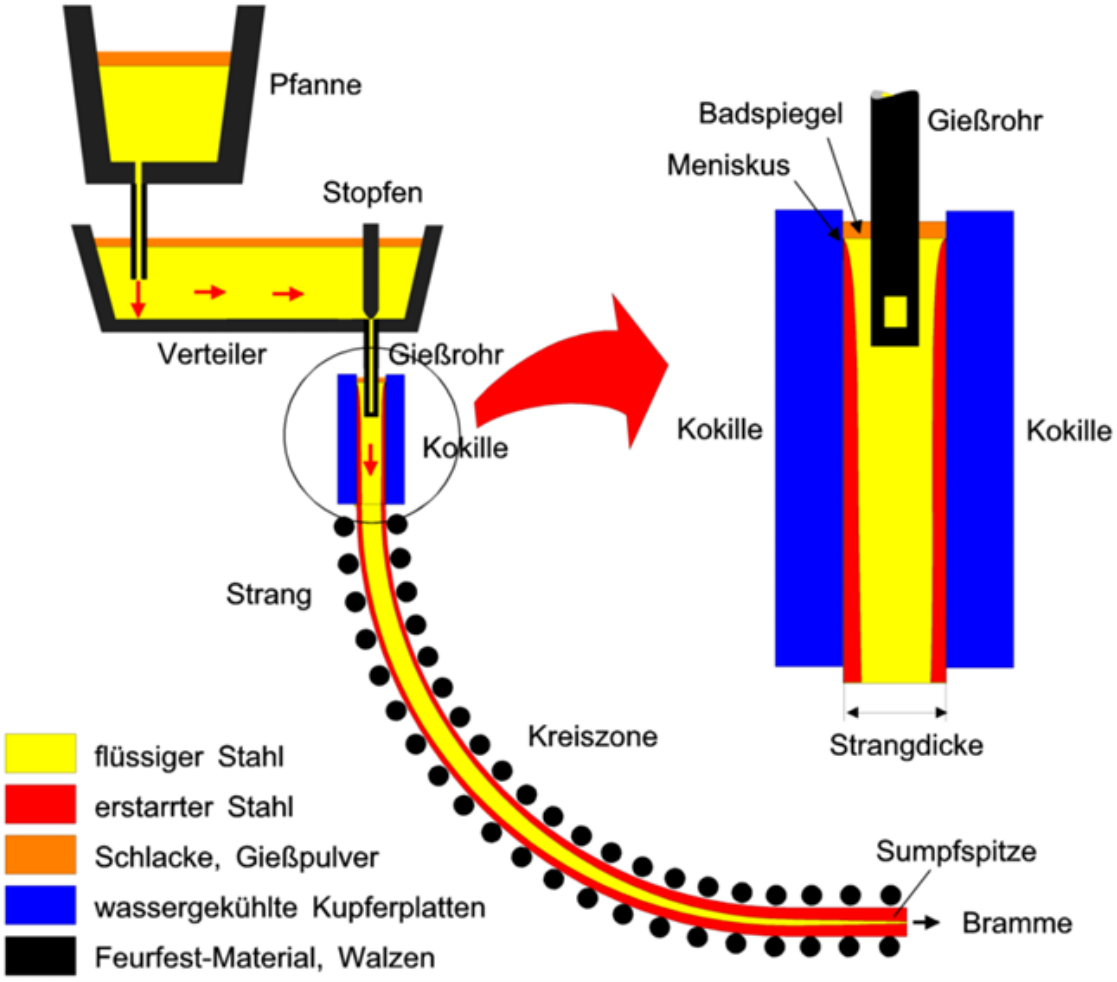


Abbildung 2.1: Stranggießen [Sta16]

2.2 Vorhandenes EQX System

Die mobile Applikation wurde auf der Grundlage einer bereits vorhandenen Desktop Applikationentwickelt. Ein Level 2 System ist nach Client / Server Architektur realisiert und beinhaltet in der Regel mehrere Pakete (Anwendungen). Eines dieser Pakete ist die Bauteilverwaltung EQX (Equipment Expert). Beispiele für weitere Pakete sind Materialverfolgung, Schnittlängenoptimierung oder Kühlmodell. Die Desktop Applikation, auch HMI (Human Machine Interface) genannt, dient als gemeinsame Visualisierung aller installierten Level 2 Pakete. Für die Bauteilverwaltung EQX gibt es eine bestehende EQX Desktop Applikation und einen Anwendungsserver. Dieser Anwendungsserver enthält die Business Logik für die Bauteilverwaltung dient als Service zum Datenaustausch mit dem HMI. Die mobile Applikation verwendet den gleichen Anwendungsserver, wobei sich dazwischen ein Webservice befindet (siehe Kapitel 4.1).

In den Abbildungen 2.2 und 2.3 ist die EQX Funktion im HMI zu sehen. Mit den Buttons *Inventory* und *Installed* in der *View Selection* kann zwischen der Inventaransicht und der Baumansicht der Stranggießanlage gewechselt werden.

Abbildung 2.2 zeigt im linken Abschnitt *Current - Inventory Equipment* das Inventar und im rechten Abschnitt *Current - Inventory Equipment - Details* die Details zum in der Inventaransicht ausgewählten Bauteil. Die Inventaransicht enthält eine Liste aller Bauteiltypen und die vorhandenen Bauteile jedes Bauteiltypen. Die Liste der Bauteile kann für jeden Bauteiltypen aufgeklappt und zugeklappt werden.

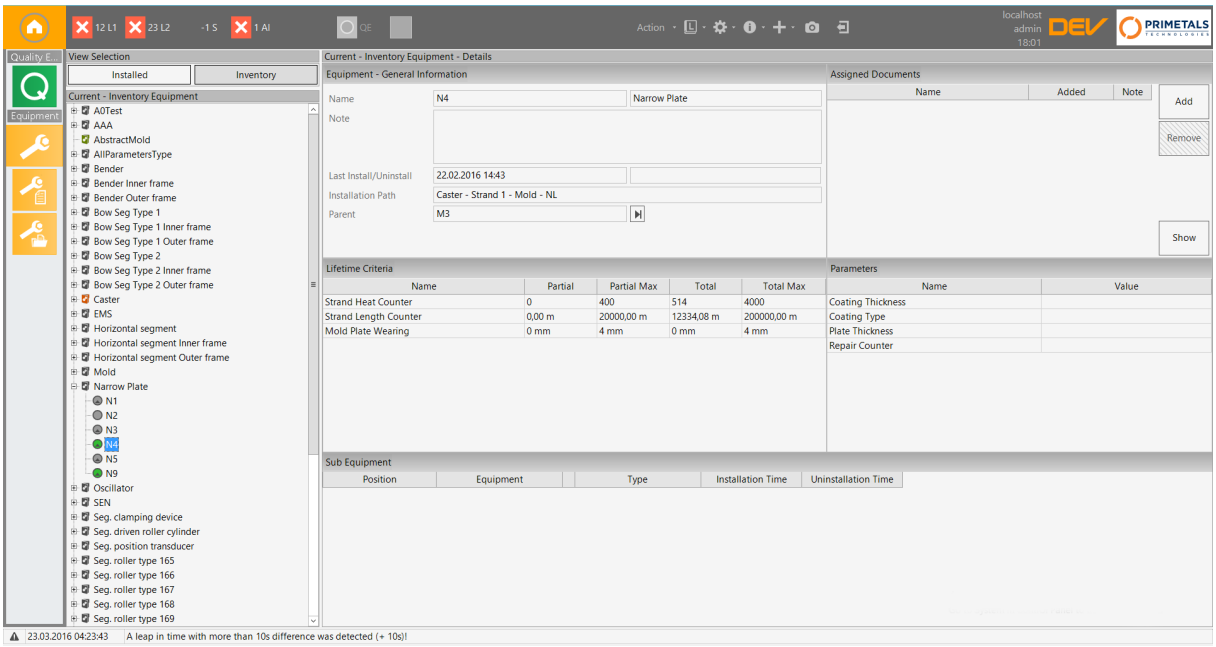


Abbildung 2.2: Inventaransicht und Detailansicht des Bauteils *N4* vom Bauteiltyp *Narrow Plate*

In Abbildung 2.3 ist im linken Abschnitt *Current - Installed Equipment* die Baumansicht

der Strangussanlage zu sehen. Der rechte Abschnitt *Current - Installed Equipment - Details* zeigt die Details zum ausgewählten Bauteil *CC2*. Die Baumansicht besteht aus einer Wurzelposition und deren Unterpositionen. Unterpositionen können wiederum Unterpositionen haben, somit ergibt sich ein rekursiver Aufbau des Baumes ausgehend von der Wurzelposition.

Die EQX Desktop Applikation bietet im Gegensatz zu der mobilen Applikation einige zusätzliche Funktionen, wie zum Beispiel die historische Speicherung von Bauteildaten (welcher Bauteil war wann wo und wie oft in der Anlage installiert und wie waren die Daten zum Installationszeitpunkt) oder Funktionen, die nur von der Desktop Applikation aus möglich sein sollen.

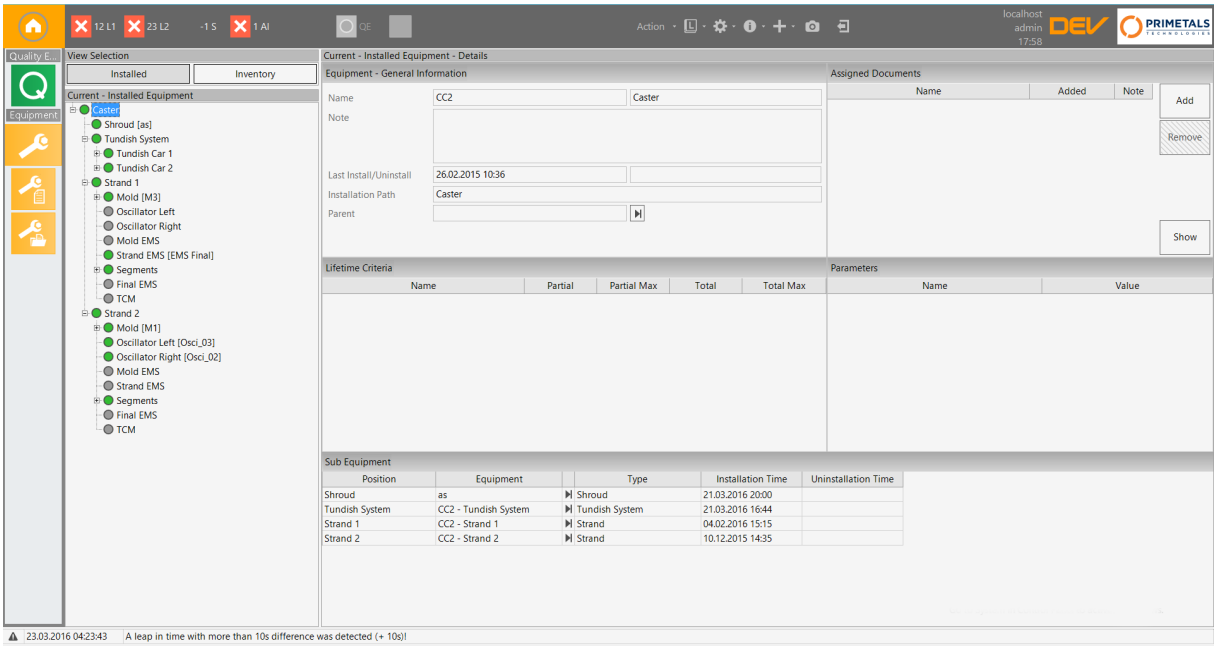


Abbildung 2.3: Baumansicht der installierten Bauteile und Detailansicht des Bauteils *CC2* vom Bauteiltyp *Caster*

2.3 TypeScript

TypeScript [Mic16] ist eine kostenlose Open Source Programmiersprache, die auf JavaScript aufbaut und von Microsoft entwickelt wird. TypeScript bietet im Gegensatz zu JavaScript Typisierung und objektorientierte Programmierung. Der Source Code wird zu JavaScript transpiliert (Übersetzung von Source Code zu Source Code). Der Hauptgrund für den Einsatz von TypeScript ist die vereinfachte Implementierung, da Objektorientierung und Typisierung vorhanden sind und bereits bekannte Syntax und Semantik von JavaScript verwendet wird. Darüber hinaus bietet TypeScript Klassen, Module und Interfaces an.

2.3.1 Typen

TypeScript bietet die vier Basistypen *String*, *Number*, *Boolean* und *Any*. Weiters gibt es noch *Array*, *Enum* und *Void*. Wie auch JavaScript verwendet TypeScript doppelte- und einfache Anführungszeichen für Strings. Ausschnitt 2.1 zeigt die Syntax für die Verwendung dieser Basistypen.

```
1  var isDone: boolean = false;
2
3  var height: number = 6;
4
5  var name: string = "bob";
6  name = 'smith';
7
8  var notSure: any = 4;
9  notSure = "maybe a string instead";
10 notSure = false;
```

Ausschnitt 2.1: TypeScript Basistypen (vgl. [Mic16])

Wie Ausschnitt 2.2 zeigt, können *Arrays* in TypeScript auf zwei verschiedene Arten deklariert werden.

```
1  var listOne:number[] = [1, 2, 3];
2
3  var listTwo:Array<number> = [1, 2, 3];
```

Ausschnitt 2.2: TypeScript Array (vgl. [Mic16])

Ausschnitt 2.3 zeigt den Einsatz eines *Enums*.

```
1  enum Color {Red, Green, Blue};
2  var c: Color = Color.Green;
```

Ausschnitt 2.3: TypeScript Enum (vgl. [Mic16])

Um zu signalisieren, dass eine Methode keinen Wert zurück gibt, wird der optionale Datentyp *Void* verwendet. Ausschnitt 2.4 zeigt ein Anwendungsbeispiel für diesen Datentypen.

```
1 function warnUser(): void {
2   alert("This is my warning message");
3 }
```

Ausschnitt 2.4: TypeScript Datentyp Void (vgl. [Mic16])

2.3.2 Klassen

Ausschnitt 2.5 zeigt die Deklaration einer Klasse *Greeter* mit der Member Variable *greeting* vom Typ *string* und der Methode *greet()*, welche einen String zurückgibt.

```
1 class Greeter {
2   greeting: string;
3   constructor (message: string) {
4     this.greeting = message;
5   }
6   greet() {
7     return "Hello, " + this.greeting;
8   }
9 }
```

Ausschnitt 2.5: TypeScript Klasse (vgl. [Mic16])

Ausschnitt 2.6 zeigt den erzeugten JavaScript Code der in Ausschnitt 2.5 deklarierten Klasse *Greeter*. Beachten Sie, wie TypeScript die Typen der Membervariablen und Methodenparametern definiert. Diese werden bei der Übersetzung zu JavaScript entfernt, hilft aber den IDEs und Compilern, Fehler wie z.B. das Übergeben eines numerischen Typen beim Konstruktor zu erkennen.

```
1 var Greeter = (function () {
2   function Greeter(message) {
3     this.greeting = message;
4   }
5   Greeter.prototype.greet = function () {
6     return "Hello, " + this.greeting;
7   };
8   return Greeter;
9 })();
```

Ausschnitt 2.6: Erzeugter JavaScript Code [Mic16]

TypeScript bietet eines der wichtigsten Muster der Objektorientierung, die Vererbung, und die Zugriffsmodifikatoren *private*, *protected*, und *public* an. In TypeScript ist jeder

Member (Klassen, Variablen, Methoden) standardmäßig *public*. Ausschnitt 2.7 zeigt die Verwendung von Vererbung und dem Zugriffsmodifikator *private*.

```
1  class Animal {
2    private name:string;
3    constructor(theName: string) { this.name = theName; }
4  }
5
6  class Rhino extends Animal {
7    constructor() { super("Rhino"); }
8  }
9
10 class Employee {
11   private name:string;
12   constructor(theName: string) { this.name = theName; }
13 }
14
15 var animal = new Animal("Goat");
16 var rhino = new Rhino();
17 var employee = new Employee("Bob");
18
19 animal = rhino;
20 animal = employee; //error: Animal and Employee are not
     compatible
```

Ausschnitt 2.7: TypeScript Vererbung, Zugriffsmodifikatoren (vgl. [Mic16])

2.3.3 Module

Module fassen Funktionalitäten zusammen und können sich über mehrere Dateien erstrecken. Um Klassen und Variablen auch außerhalb eines Moduls zugänglich zu machen, wird das Schlüsselwort *export* als Präfix verwendet. Um diese exportierten Member in anderen Modulen verwenden zu können, verwendet man das Schlüsselwort *import*. Ausschnitt 2.8 zeigt ein Modul zur Validierung, bei dem die Klassen durch das *export* Präfix auch außerhalb des Moduls zugänglich sind, im Gegensatz zu den Variablen *lettersRegexp* und *numberRegexp*. Über die Punkt Notation kann dann auf die Klassen zugegriffen werden.

```
1  module Validation {
2    export interface StringValidator {
3     isAcceptable(s: string): boolean;
4    }
5
6    var lettersRegexp = /^[A-Za-z]+$/;
7    var numberRegexp = /^[0-9]+$/;
8
9    export class LettersOnlyValidator implements
        StringValidator {
10     isAcceptable(s: string) {
11       return lettersRegexp.test(s);
12     }
13   }
14
15   export class ZipCodeValidator implements StringValidator
        {
16     isAcceptable(s: string) {
17       return s.length === 5 && numberRegexp.test(s);
18     }
19   }
20 }
21
22 var validator: Validation.StringValidator = new Validation.
     ZipCodeValidator();
23 validator.isAcceptable('98052');
```

Ausschnitt 2.8: TypeScript Modul (vgl. [Mic16])

2.3.4 Vorteile gegenüber JavaScript

Wie der Name schon sagt, ist TypeScript im Gegensatz zu JavaScript typisiert. Fehler können schon vor der Ausführung des Codes behoben werden, da der TypeScript Transpiler bei der Übersetzung zu JavaScript auf diese hinweist. Dabei werden auch Zuweisungen von nicht übereinstimmenden Datentypen (siehe Ausschnitt 2.9) erkannt, was bei JavaScript nicht der Fall ist.

```
1 function getAString():string {
2   return "aString";
3 }
4
5 var aNumber:number;
6 // Erzeugt: error TS2322 Type 'string' is not assignable to
      type 'number'.
7 aNumber = getAString();
```

Ausschnitt 2.9: Nicht übereinstimmende Datentypen

Typsicherheit ist einer der größten Vorteile von TypeScript gegenüber JavaScript. Zusätzlich zu den Basisdatentypen können auch eigene komplexe Datenstrukturen in Form von Klassen implementiert werden.
Für größere JavaScript Projekte kann die Umstellung auf TypeScript eine fehlerfreiere Software zur Folge haben.

2.4 AngularJS

AngularJS [Goo16] (im Rahmen dieser Arbeit als Angular bezeichnet), ist ein von Google entwickeltes Open-Source JavaScript Framework, das die Erstellung von Single-Page Webapplikationen ermöglicht.

2.4.1 Single-Page Webapplikationen

Eine Single-Page Webapplikation (SPA) besteht aus einem einzigen HTML-Dokument, dessen notwendiger Code (HTML, JavaScript, CSS) je nach Benutzerinteraktion dynamisch mittels AJAX nachgeladen wird. Das Ziel einer SPA ist es, dem Benutzer eine flüssigere Benutzererfahrung mit einer Ähnlichkeit zu Desktop-Anwendungen zu bieten.

2.4.2 Model-View-Controller Architektur

Um die Applikationslogik, die View und das Model zu trennen, implementiert Angular das MVC (Model-View-Controller) Design Pattern. Eine Angular Webapplikation besteht aus Modulen, Controllern, HTML View-Templates, Konfigurationen, Filtern, Scopes und Services. Diese modulare Struktur wird durch den Dependency-Container von Angular ermöglicht. Angular bietet die Funktion des sogenannten *Two-way Data Binding* Prinzips an, mit dem Änderungen von Datenquellen mit der View synchronisiert werden können und umgekehrt.

2.4.3 Direktiven

Angular *Direktiven* erlauben es dem Entwickler, eigene, wiederverwendbare HTML-Elemente und deren Verhalten zu definieren. Die folgenden Direktiven sind bereits in Angular integriert und werden in dieser Arbeit verwendet:

- ng-app
  Deklariert das Wurzelelement der Angular Applikation und wird meist in der Nähe des Wurzelelements der Seite platziert (meist das <html> Tag). Ausschnitt 2.10 Zeile 2 legt das in Ausschnitt 2.11 Zeile 20 definierte Modul als Applikationsmodul fest.

- ng-controller
  Die ng-controller Direktive legt fest, welche Controller-Klasse für die festgelegte Ansicht zuständig ist. Weiters kann eine Controller-Klasse auch über die Angular Routing-Konfiguration für eine Ansicht festgelegt werden. Zeile 9 in Ausschnitt 2.10 legt den Controller *ExampleCtrl* als Controller fest, der in Ausschnitt 2.11 in Zeile 3 als Klasse definiert und in Zeile 21 als Controller zum Applikationsmodul hinzugefügt wird.

- ng-bind
  Das ng-bind Attribut ersetzt den Inhalt eines HTML Elements mit dem Wert des gegebenen Ausdrucks. Verändert sich die als Ausdruck angegebene Variable, ändert sich auch der Text des Elements. Diese Direktive ist das Equivalent zur doppelt geschwungenen Klammer-Notation {{Ausdruck}}, die nicht als Element Attribut angewendet wird, sondern innerhalb eines Elements steht. Ausschnitt 2.10 Zeile 10 zeigt die Anwendung dieser Direktive. Zeile 11 zeigt die Anwendung der doppelt geschwungenen Klammer-Notation. Im Browser sehen für den Benutzer jedoch beide Arten gleich aus. Beide "binden" ihren Wert an die im Controller *ExampleCtrl* in Ausschnitt 2.11 Zeile 5 definierte Variable *bindValue*.
  Um die Performanz einer Applikation zu steigern, sollten Elemente mit statischen Attributen durch die sogenannte *One-time binding* Syntax mit zwei Doppelpunkten vor dem Ausdruck gebunden werden. Stellt man den beiden Ausdrücken zwei Doppelpunkte voran (*ng-bind="::ctrl.bindValue"* und {{*::ctrl.bindValue*}}), dann wird der Wert nur einmal gebunden und bei einer Veränderung nicht aktualisiert.

- ng-model
  Diese Attribut-Direktive kann auf Input, Select und Textarea HTML-Elemente angewendet werden. Damit wird der Inhalt des gebundenen Elements mit dem Wert einer Controller-Variable verknüpft. Im Gegensatz zu *ng-bind* wird bei der *ng-model* Direktive bei einer Veränderung des Elementes (z.B. einer Checkbox) aber auch die Variable im Hintergrund mit verändert. Zeile 12 in Ausschnitt 2.10 zeigt die Anwendung von *ng-model*. Wird auf die gebundene Checkbox geklickt, ändert sich im Hintergrund die in Ausschnitt 2.11 Zeile 6 definierte Variable *boolValue* mit.

- ng-class
  Die ng-class Direktive erlaubt es, die CSS-Klassen eines Elements dynamisch auf Basis eines Ausdrucks hinzuzufügen und zu entfernen. Ergibt die Auswertung des übergebenen Ausdrucks den Wert *true*, werden dem Element CSS-Klassen hinzugefügt. Zeile 13 in Ausschnitt 2.10 zeigt die Verwendung von *ng-class*. Ergibt der Ausdruck (*ctrl.boolValue*) den Wert *true*, dann wird dem Element die CSS-Klasse *red* hinzugefügt. Verändert sich der Wert nun auf *false*, so wird die CSS-Klasse wieder entfernt. Bei einem Klick auf die in Ausschnitt 2.10 Zeile 12 definierte Checkbox, verändert sich der Wert der Variable *boolValue* und somit auch die CSS-Klasse des Elements in Zeile 13.

- ng-click
  Mit dieser Attribut-Direktive kann man Funktionen definieren, die bei einem Mausklick auf ein Element ausgeführt werden sollen. Zeile 14 in Ausschnitt 2.10 zeigt die Anwendung dieser Direktive. Bei einem Klick auf das Element wird die in Zeile 14 in Ausschnitt 2.11 definierte Funktion *exampleFunction()* aufgerufen.

- ng-disabled
  Diese Direktive fügt einem HTML-Element das *disabled* Attribut hinzu, wenn der Ausdruck in der Direktive *true* ergibt. Dies ist nützlich, um Buttons oder Textfelder auszugrauen und dem Benutzer zu signalisieren, dass diese deaktiviert sind. Zeile 14 in Ausschnitt 2.10 zeigt einen Button, der ausgegraut wird, wenn der übergebene

Ausdruck *ctrl.boolValue* den Wert *true* ergibt. Durch einen Klick auf die in Zeile 12 definierte Checkbox kann man den Button dynamisch deaktivieren und wieder aktivieren, da die Checkbox auf die in Ausschnitt 2.11 Zeile 6 definierte Variable *boolValue* gebunden ist und sich somit der Wert dieser Variable bei einem Klick verändert.

- ng-href
  Legt man mittels der doppelt geschwungenen Klammer-Notation dynamische URLs im *href* Attribut des HTML Anker-Element (<a>) fest, kann es sein, dass der Benutzer auf den Link klickt, bevor Angular die Chance hatte, den Ausdruck auszuwerten und den Link mit dem tatsächlichen Wert zu ersetzen. Um dieses Problem zu lösen, sollte die *ng-href* Direktive anstatt des *href* Attributs für HTML Anker-Elemente (<a>) verwendet werden. Zeile 15 in Ausschnitt 2.10 zeigt die Verwendung dieser Direktive. Der Ausdruck {{*ctrl.url*}} wird im Hintergrund durch die in Ausschnitt 2.11 in Zeile 7 definierte Variable *hrefUrl* ersetzt.

- ng-if
  Die ng-if Direktive zeigt HTML-Elemente und deren Inhalt nur dann an, wenn der übergebene Ausdruck *true* ergibt. Das Element wird komplett aus dem Document Object Model (DOM) entfernt. Im Gegensatz dazu gibt es die ng-show Direktive, welches nur *display: none;* als CSS-Style hinzufügt und somit das Element nur ausblendet. Zeile 15 in Ausschnitt 2.10 zeigt die Verwendung der *ng-if* Direktive. Ergibt der Ausdruck *ctrl.boolValue* den Wert *false*, so wird das Element aus dem DOM entfernt. Ergibt der Ausdruck dann wieder *true*, wird das Element wieder hinzugefügt. So kann man mit der in Zeile 12 definierten Checkbox das Element ein- und ausblenden.

- ng-repeat
  Die ng-repeat Direktive wiederholt ein HTML-Element mitsamt seinem Inhalt für jedes Element in einer gegebenen Liste. Dies ist äußerst nützlich, wenn man beispielsweise Daten von einem Server erhält und seine Single-Page Webapplikation dynamisch aufbauen möchte. Weiters erspart man sich Implementierungsaufwand, da Veränderungen, die das zu wiederholende Element betreffen, nur einmal durchgeführt werden. Zeile 16 in Ausschnitt 2.10 zeigt die Anwendung dieser Direktive. Das Element wiederholt sich für jeden Eintrag in dem in Zeile 9 in Ausschnitt 2.11 definierten String Array *entries* und ist auf den Wert des aktuellen Elements im Array gebunden.

- ng-src
  Ähnlich wie beim *href* Attribut im HTML Anker-Element <a>, funktioniert die *ng-src* Direktive bei HTML Bilder-Elementen <img>. Da die Bildquelle oft dynamisch erzeugt wird und der Browser dieses oft schon lädt bevor Angular den Ausdruck in der doppelt geschwungenen Klammer-Notation ersetzt hat, kann es sein, dass das Bild nicht gefunden wird. Um dieses Problem zu vermeiden, sollte die ng-src Direktive in HTML Bilder-Elementen verwendet werden. Zeile 17 in Ausschnitt 2.10 zeigt die Anwendung der *ng-src* Direktive. Der Ausdruck {{*ctrl.url*}} wird im Hintergrund durch die in Ausschnitt 2.11 in Zeile 8 definierte Variable *imageUrl*

ersetzt.

2.4.4 Beispiel

```
1  <!DOCTYPE html>
2  <html ng-app="example">
3  <head>
4    <meta charset="UTF-8">
5    <title>Angular Beispiel</title>
6    <link rel="stylesheet" href="css/style.css">
7  </head>
8  <body>
9  <div ng-controller="ExampleCtrl as ctrl">
10   <div ng-bind="ctrl.bindValue"></div>
11   <div>{{ctrl.bindValue}}</div>
12   <input type="checkbox" ng-model="ctrl.boolValue">
13   <div ng-class="{red: ctrl.boolValue}">CSS</div>
14   <button ng-disabled="ctrl.boolValue" ng-click="ctrl.
       exampleFunction()">click</button>
15   <a ng-if="ctrl.boolValue" href="" ng-href="{{ctrl.url}}">
       url</a>
16   <div ng-repeat="entry in ctrl.entries" ng-bind="entry"></
       div>
17   <img src="" ng-src="{{ctrl.imageUrl}}" alt="">
18 </div>
19
20   <script type="text/javascript" src="dependencies/
       dependencies.js"></script>
21   <script type="text/javascript" src="bundle.js"></script>
22 </body>
23 </html>
```

Ausschnitt 2.10: Angular Beispiel index.html

```
1  module example {
2
3    export class ExampleCtrl {
4
5      public bindValue:string = "value";
6      public boolValue:boolean = false;
7      public hrefUrl:string = "www.google.com";
8      public imageUrl:string = "image.png";
9      public entries:string[] = ['entry 1', 'entry 2', 'entry
           3'];
10
11     constructor() {
12     }
13
14     public exampleFunction():void {
15       alert("Hello World!");
16     }
17   }
18
19   ExampleCtrl.$inject = [];
20   angular.module('example', [])
21     .controller('ExampleCtrl', ExampleCtrl);
22 }
```

Ausschnitt 2.11: Angular Beispiel app.ts

2.4.5 Benutzerdefinierte Direktiven

Angular erlaubt es dem Entwickler, eigene Direktiven zu implementieren. Um während HTTP-Requests eine Ladeanimation anzuzeigen, wurde die in Ausschnitt 2.12 gezeigte *Spinner* Direktive implementiert. Diese Direktive wird meist in Zusammenhang mit *ng-if* verwendet, um die Animation mit einer Boolean-Variable ein- bzw. auszublenden. Ausschnitt 2.13 zeigt die Verwendung in HTML. Wenn die Controller-Variable *uploading* den Wert *true* hat, wird der Spinner angezeigt.

```
module eqx.components.spinner {

  export class Spinner {

    public templateUrl = 'components/spinner/spinner.html';
    public scope = {};
    public link:(scope:ng.IScope, element:ng.
       IAugmentedJQuery, attrs:ng.IAttributes) => void;

    constructor() {
      Spinner.prototype.link = (scope:ng.IScope, element:ng
         .IAugmentedJQuery, attrs:ng.IAttributes) => {
      };
    }

    public static factory() {
      var directive = () => {
        return new Spinner();
      };
      directive['$inject'] = [];
      return directive;
    }
  }

  angular.module('eqx.components.spinner', [])
    .directive('spinner', Spinner.factory());
}
```

Ausschnitt 2.12: Spinner Direktive

```
<div ng-if="ctrl.uploading">
  <spinner></spinner>
</div>
```

Ausschnitt 2.13: Verwendung Spinner Direktive

2.4.6 Services

Services dienen in Angular als "Werkzeug", um Code, der oft verwendet wird (z.B. für Http-Requests auf einen Server) über die gesamte Applikation zu verteilen. Services sind *Singletons*, das heißt jede Komponente, die das Service verwendet, hat eine Referenz auf das gleiche Service, welches von der Angular Service Factory erzeugt wird. Weiters verwendet Angular bei Services das Entwurfsmuster *Lazy Loading*, wobei ein Service erst instanziert wird, wenn eine Komponente davon abhängt.

2.4.7 Konfigurationen

Konfigurationen wie Routing oder Internationalisierung werden in Angular als *config* registriert und werden beim Bootstrap-Vorgang der Applikation ausgeführt. Durch sie können Einstellungen, welche innerhalb der gesamten Applikation gelten, definiert werden.

2.4.8 Dependency Injection

Dependency Injection ist ein Software Design Pattern, das beschreibt wie Komponenten an ihre Abhängigkeiten kommen. Das Angular Injector Modul verwaltet die Erstellung neuer Komponenten, das Auflösen ihrer Abhängigkeiten und die Bereitstellung von diesen. Das macht es einfacher, Komponenten zu testen, wieder zu verwenden und zu verwalten. Abhängigkeiten sind *Singletons*, das heißt sie werden nur einmal vom Angular Injector Modul erzeugt und an Komponenten, welche davon abhängen, weiter gegeben. Jede Angular Applikation hat einen *Injector*, der für die Konstruktion und Auffindung von Abhängigkeiten zuständig ist. Komponenten in Angular werden durch einen String identifiziert und können so vom *Injector* identifiziert werden.

2.5 Apache Cordova

2.5.1 Übersicht

Apache Cordova [Fou15b] ist ein Open-Source Framework zur Entwicklung von mobilen Applikationen. Die frühere Bezeichnung dafür war Adobe Phonegap. Das Framework wurde 2011 von der Apache Software Foundation (ASK) unter dem Namen Apache Cordova übernommen und weiterentwickelt [Ler12]. Seitdem ist PhoneGap eine Distribution von Apache Cordova.
Mit dem Cordova Framework können mobile Applikationen mithilfe von bekannten Web-Technologien entwickelt werden. Dazu zählen HTML5, CSS3 und JavaScript. Dies ermöglicht eine plattformunabhängige Entwicklung. Für jede Zielplattform wird ein natives Paket zur Installation erzeugt. Dabei werden auch Standardschnittstellen der einzelnen Plattformen für den Zugriff auf native Komponenten des Betriebssystems verwendet. Unter anderem zählen dazu die Kommunikation mit Sensoren oder die Abfrage des Netzwerkstatus. Für spezielle native Zugriffe sind Plugins (siehe Kapitel 2.5.2) notwendig.

2.5.2 Architektur

Eine Cordova Applikation besteht aus mehreren Komponenten. Die einzelnen Komponenten und deren Beziehungen sind in der Abbildung 2.4 zu sehen. Dazu gehören die Webapplikation, die WebView und Plugins. Die WebView und die Plugins sind für jede Plattform nativ implementiert. Diese kommunizieren über Schnittstellen des Betriebssystems mit nativen Komponenten.

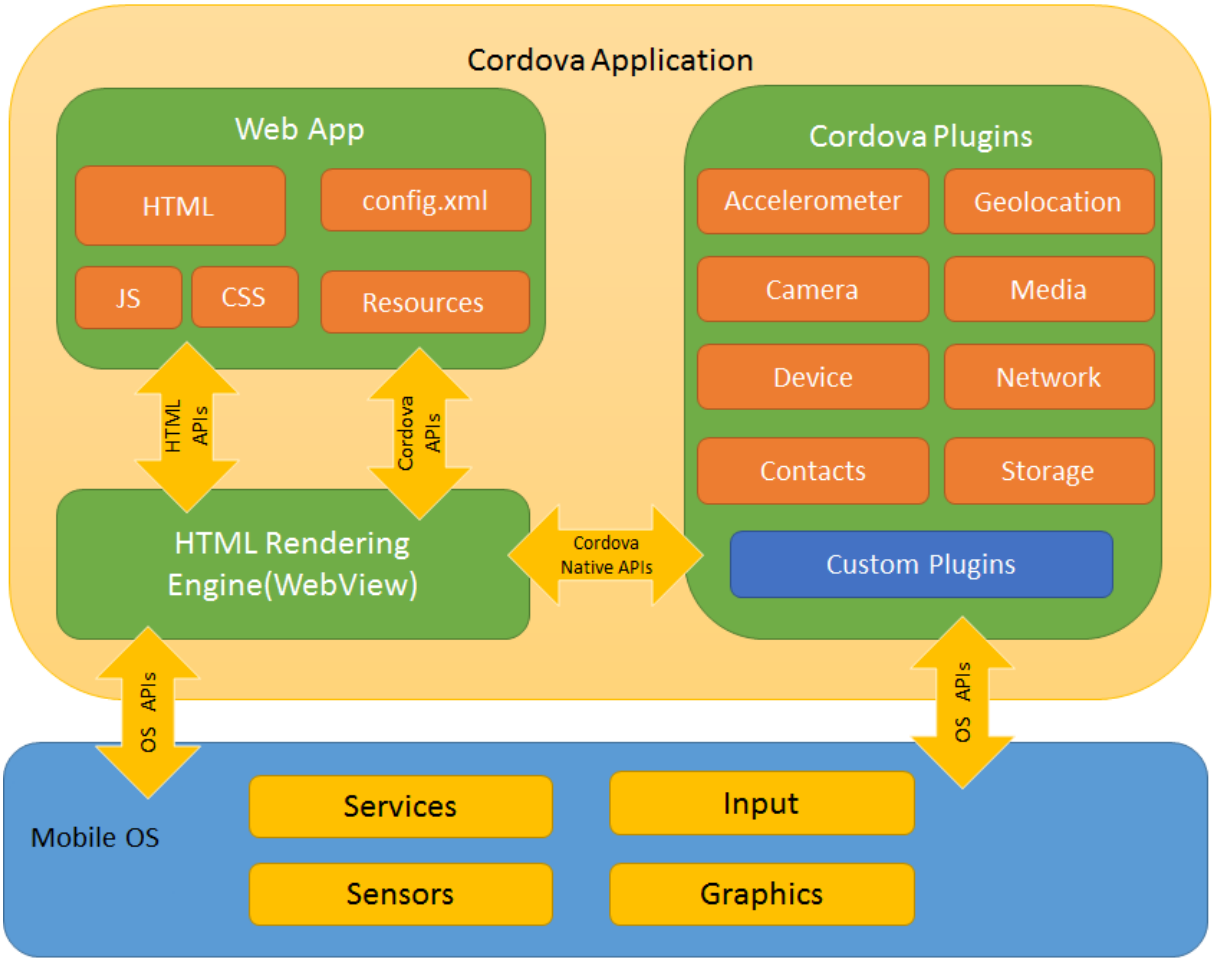


Abbildung 2.4: Cordova Architektur [Fou15a]

Webapplikation und WebView

Der Kern einer Cordova Applikation bildet die native WebView. Je nach Plattform ist diese in einer anderen Programmiersprache implementiert. Die WebView bildet die gesamte Benutzeroberfläche einer Cordova Applikation. Auf einigen Plattformen kann diese auch ein Teil einer größeren Applikation sein, die die Cordova WebView mit nativen Komponenten verbindet. Der Hauptbestandteil der Cordova Applikation ist die Webapplikation. Darin sind der Programmcode und die notwendigen Ressourcen enthalten. Die Webapplikation selber ist wie eine Webseite implementiert. Diese enthält eine HTML Indexdatei, die auf weitere HTML, CSS beziehungsweise JavaScript Dateien verweist. Zudem wird auch auf benötigte Ressourcen wie zum Beispiel Bilder verwiesen. Zur Erzeugung einer Cordova Applikation sind nur HTML5 Webseiten erlaubt. Zur Implementierung dieser Applikation wurden Typescript und SCSS (siehe Kapitel 2.3 und 2.7) verwendet. Diese werden mithilfe des Taskrunners *gulp* (siehe Kapitel gulp) in JavaScript und CSS transpiliert. Beispielsweise funktioniert die Verbindung mit einer MySQL-Datenbank über PHP Dateien in einer Cordova Applikation nicht. Die Webapplikation kann unabhängig von

Cordova wie eine normale Webseite im Browser ausgeführt werden. Die fertige Cordova Applikation startet die Webapplikation in einer WebView, die im nativen Paket enthalten ist.
Die Konfigurationsdatei *config.xml* enthält Informationen über die Cordova Applikation und definiert globale Parameter.

Plugins

Plugins sind ein wichtiger Teil des Cordova Frameworks. Sie bieten eine Schnittstelle für die Kommunikation zwischen Cordova und plattformspezifischen Schnittstellen beziehungsweise Hardware Komponenten. Plugins ermöglichen die Ausführung von nativem Code durch den Aufruf einer JavaScript Funktion. In der Abbildung 2.5 ist die Architektur eines Plugins zu sehen. Es besteht immer aus zwei Teilen. Der erste Teil ist in JavaScript implementiert und befindet sich in der WebView. Der zweite Teil enthält die Funktionalität des Plugins und ist in nativer Sprache implementiert, zum Beispiel Java für Android und Objective-C für iOS. Zwischen diesen befindet sich eine von Cordova zur Verfügung gestellte Verbindungsschnittstelle. [Rip14]

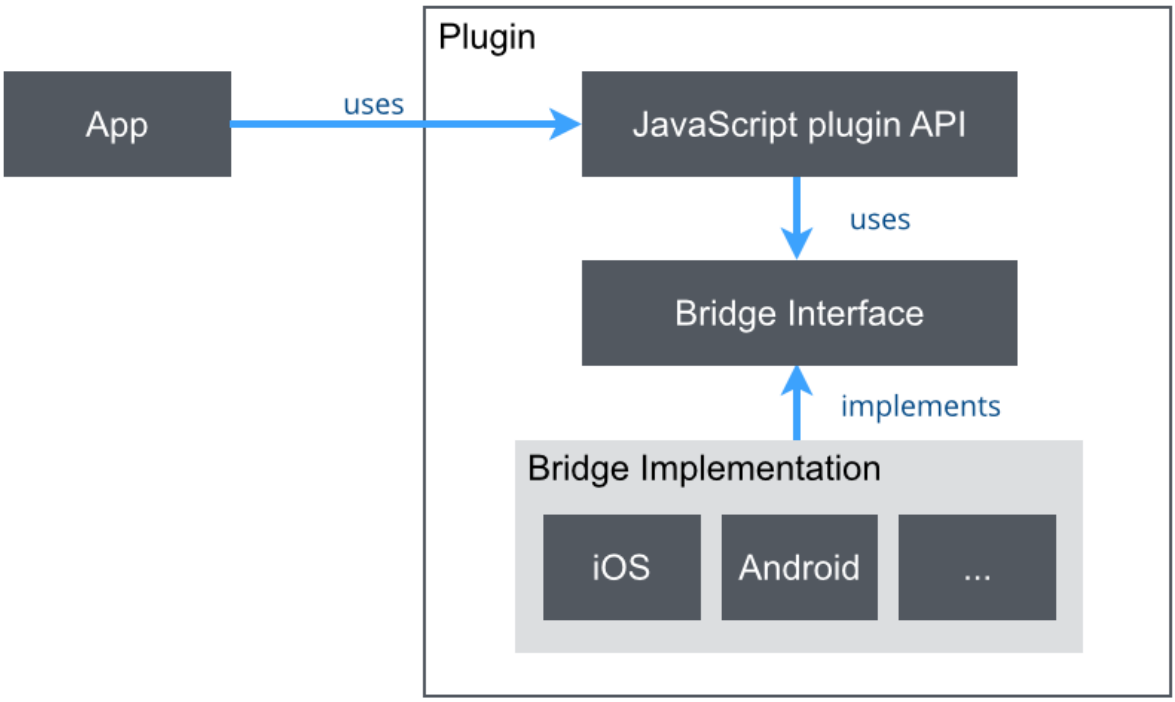


Abbildung 2.5: Cordova Plugin Architektur [Rip14]

Apache Cordova und PhoneGap stellen einige Plugins zur Verfügung. Diese sogenannten *Core Plugins* ermöglichen den Zugriff auf Gerätefunktionen wie zum Beispiel die Kamera oder Netzwerkinformationen. Bei der Erstellung eines Cordova Projektes sind

keine Plugins enthalten. Jedes erwünschte Plugins muss explizit hinzugefügt werden, sogar Core Plugins. Eine Liste der verfügbaren Core Plugins ist auf der offizellen Apache Cordova Webseite dokumentiert. Jedes dieser Plugins ist ein GitHub Projekt und kann so unkompliziert hinzugefügt werden. Zusätzlich zu den Core Plugins gibt es verschiene Plugins von Drittanbietern. Diese können zusätzliche Funktionen bieten, die nicht zwingend auf jeder Plattform verfügbar sind, wie zum Beispiel Near Field Communication (NFC). Erfüllt keine der existierenden Plugins die gewünschten Anforderungen, besteht zudem die Möglichkeit ein eigenes Plugin zu implementieren. Im Rahmen dieser Arbeit wurde ein eigenes NFC-Plugin (siehe Kapitel 5.5) für die Android Plattform entwickelt, da bereits vorhandene NFC-Plugins die keine Lese- und Schreibfunktionen für die verwendeten NFC-Tags (siehe Kapitel: NFC Tags) anbieten.

2.6 Near Field Communication (NFC)

2.6.1 Funktionsweise von NFC

Die Nahfeldkommunikation (engl.: Near Field Communikation, kurz NFC) ist eine drahtlose Kommunikationstechnik. NFC-fähige Komponenten können kontaktlos Daten über kurze Distanzen austauschen. Diese Technik wurde im Jahr 2002 von NXP Semiconductors und Sony entwickelt und basiert auf RFID (Radio Frequency Identification) und elektronische Chipkarten [Har13]. NFC-Systeme basieren auf einer hohen Frequenz von 13.56MHz. Die maximale Reichweite liegt bei 10 cm, ist aber vom Gerät beziehungsweise Tag abhängig. Zur Zeit sind Übertragungsraten von bis zu 424kbit/s möglich. Das Prinzip der NFC Kommunikation gleicht der 13.56 MHz RFID, es gibt einen *Master* und einen *Slave*. Der *Master* ist der Initiator oder *Leser/Schreiber*, der die Verbindung aufbaut. Der *Slave* ist das Zielgerät oder der NFC-Tag [Pro16]. In Abbildung 2.6 ist die Kommunikation über NFC zwischen zwei Geräten im aktiven Kommunikationsmodus. Beide Geräte erzeugen ein eigenes elektromagnetisches Feld zur Kommunikation und haben eine eigene Energiequelle. Die Magnetfelder werden abwechselnd auf- und abgebaut, abhängig davon welches Gerät die Daten sendet.

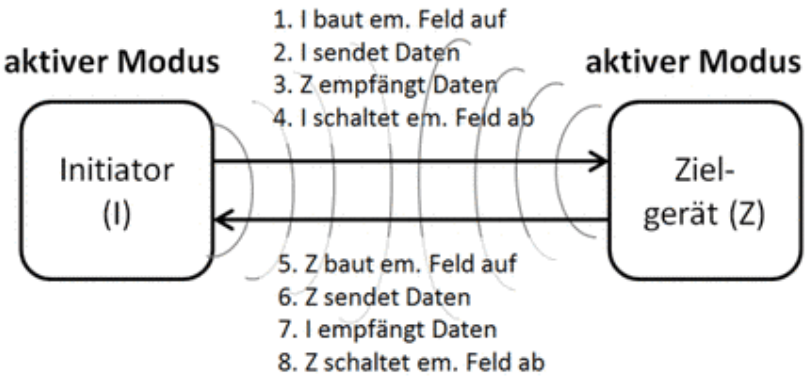


Abbildung 2.6: Aktiver-aktiver Kommunikationsmodus [Har13]

Erzeugt ein NFC-Gerät kein elektromagnetische Feld befindet es sich im passiven Kommunikationsmodus. Es bezieht seine Energie über das magnetische Feld eines anderen Gerätes. Dies ist aufgrund der Induktion möglich. In Abbildung 2.7 ist die Kommunikation zwischen einem Gerät im aktiven und einem Gerät im passiven Kommunikationsmodus zu sehen. Diese Kommunikationstechnologie wird zum Lesen und Beschreiben der auf den Bauteilen befestigten NFC-Tags verwendet.

Abbildung 2.7: Aktiver-passiver Kommunikationsmodus [Har13]

2.6.2 NFC Tags

NFC Tags können in verschiedensten Formen auftreten. Zum Beispiel Aufkleber, Steckkarten, Schlüsselanhänger oder Chips. Ein NFC Tag enthält einen Mikrochip mit kleinen Antennen. Darauf können geringfügige Datenmengen gespeichert werden. Der Speicherplatz ist abhängig vom verwendeten Tag. Die darauf gespeicherten Informationen können von NFC-fähigen Geräten, wie zum Beispiel einem Smartphone, ausgelesen beziehungsweise überschrieben werden. Das Format, in dem die Daten gespeichert sind, ist vom Tag abhängig. In den meisten Fällen sind die Informationen im Format NDEF (NFC Data Exchange Format) gespeichert. [Kim16]

NFC Tag Typ ISO 15693

Die von Primetals eingesetzten Tags verwenden das ISO 15693 Protokoll zum Datentransfer. Diese werden auch *Generic ISO 15693 tags (RFID)* oder *Android NfcV tags* genannt. Das standardisierte Android Format NDEF kann nicht zum Lesen beziehungsweise beschreiben dieser Tags verwendet werden [NFC16]. Zum Datenaustausch wird die NfcV-Technologie von Android verwendet. Mithilfe dieser Technologie kann über das ISO 15693 Protokoll auf den Speicher des Tags zugegriffen werden. In Abbildung 2.8 ist das Format für einen Request auf den Speicher eines ISO 15693 basierten Tags zu sehen. Die Felder *Start of Frame (SOF)*, *CRC* (Prüfsumme) und *End of Frame (EOF)* werden bei einem Request über die NfcV-Technologie von Android ergänzt. Die restlichen Felder müssen manuell gesetzt werden. [STM15]

SOF	Request flags	Command code	Parameters	Data	CRC	EOF

Abbildung 2.8: Standard Request Format [STM15]

In Abbildung 2.9 ist ein konkreter Lesezugriff auf einen einzelnen Block (4 Byte) zu sehen.

Der Kommandocode zum Lesen eines einzelnen Blocks ist *20*. Das UID (Unique Identifier) Feld kann durch Angabe eines bestimmten *Request_flags* frei gelassen werden. Jedes Bit in den *Request_flags* spezifiziert eine bestimmte Aktion, je nach dem ob es gesetzt ist oder nicht. Die Blocknummer definiert den zu lesenden Block. [STM15]

Request SOF	Request_flags	Read Single Block	UID	Block number	CRC16	Request EOF
-	8 bits	20h	64 bits	8 bits	16 bits	-

Abbildung 2.9: Lesen eines einzelnen Blocks: Anfrage [STM15]

In Abbildung 2.10 ist die Antwort auf die Leseanfrage eines einzelnen Blocks zu sehen. Das *Data* Feld enthält die Daten des gelesenen Blocks. Ist dabei in den *Response_flags* das *Error_flag* gesetzt, enthält die Antwort statt dem *Block locking status* und dem *Data* Feld ein *Error code* Feld.

Response SOF	Response_ flags	Block locking status	Data	CRC16	Response EOF
-	8 bits	8 bits	32 bits	16 bits	-

Abbildung 2.10: Lesen eines einzelnen Blocks: Antwort des Tags [STM15]

In Abbildung 2.11 ist der Schreibzugriff auf einen bestimmten Tag zu sehen. Im Gegensatz zum Lesezugriff wird der Kommandocode *21* übergeben und zusätzlich ein *Data* Feld. Dieses enthält den 4 Byte großen Wert, der auf dem Zielblock gespeichert werden soll.

Request SOF	Request_ flags	Write Single Block	UID	Block number	Data	CRC16	Request EOF
-	8 bits	21h	64 bits	8 bits	32 bits	16 bits	-

Abbildung 2.11: Beschreiben eines einzelnen Blocks: Antwort des Tags [STM15]

In Abbildung 2.12 ist die Antwort auf eine Schreibanfrage zu sehen. Ist dabei in den *Response_flags* das *Error_flag* gesetzt, ist das Schreiben fehlgeschlagen und zudem ein 8 Bit großes *Error code* Feld in der Antwort enthalten. [STM15]

Response SOF	Response_flags	CRC16	Response EOF
-	8 bits	16 bits	-

Abbildung 2.12: Beschreiben eines einzelnen Blocks: Antwort des Tags [STM15]

2.7 SASS, SCSS

Syntactically Awesome Stylesheets (SASS) [HC15] ist ein Präprozessor, der SASS oder SCSS (Sassy CSS) Source Code in CSS (Cascading Style Sheets) übersetzt. SASS ermöglicht es dem Entwickler, Features wie Variablen, Verschachtelung oder Vererbung zu verwenden. Neben SASS gibt es noch eine zweite Syntax namens SCSS. Sie ist ein Superset der CSS3 Syntax, das heißt jeder gültige CSS3 Code ist auch gültiger SCSS Code und kann in SCSS verwendet werden. SCSS Dateien enden mit *.scss*. Die zweite, ursprüngliche Hauptsyntax hat im Gegensatz zu SCSS und CSS keine Klammern und Strichpunkte, sondern verwendet Einrückungen um Blöcke zu definieren. Diese Syntax hat die Dateiendung *.sass*. Beide Syntaxen werden zu CSS transpiliert, somit ist es dem Entwickler überlassen, für welche er sich entscheidet. Im Rahmen dieser Arbeit wurde die SCSS Syntax zur Entwicklung verwendet.

2.7.1 Variablen

Variablen werden in SASS und SCSS mit dem *$* Präfix deklariert. In ihnen kann man jegliche Werte speichern, die es in CSS gibt (z.B. Schriftart, Größenangaben, Farbe). Ausschnitt 2.14 zeigt die Verwendung von Variablen in der SCSS Syntax und Ausschnitt 2.15 das gleiche Beispiel mit SASS Syntax zeigt. Beide werden zu normalem CSS, wie in Ausschnitt 2.16 zu sehen ist, transpiliert.

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

Ausschnitt 2.14: SCSS Syntax (vgl. [SAS16])

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

Ausschnitt 2.15: SASS Syntax (vgl. [SAS16])

```
1 body {
2   font: 100% Helvetica, sans-serif;
3   color: #333;
4 }
```

Ausschnitt 2.16: CSS Ergebnis (vgl. [SAS16])

2.7.2 Verschachtelung

HTML hat eine klar verschachtelte Struktur, die bei CSS nicht in dieser Art vorhanden ist. Um die Implementierung zu erleichtern, wird es durch SASS und SCSS ermöglicht, Stylesheets in verschachtelter Struktur zu schreiben, um die selbe Struktur wie die des zu gestaltenden HTML Codes zu haben. Ausschnitt 2.17 zeigt Verschachtelung in SCSS und Ausschnit 2.18 das transpilierte Ergebnis in CSS.

```
1  nav {
2    ul {
3      margin: 0;
4      padding: 0;
5      list-style: none;
6    }
7
8    a {
9      display: block;
10     padding: 6px 12px;
11     text-decoration: none;
12   }
13 }
```

Ausschnitt 2.17: SCSS Verschachtelung (vgl. [SAS16])

```
1  nav ul {
2    margin: 0;
3    padding: 0;
4    list-style: none;
5  }
6
7  nav a {
8    display: block;
9    padding: 6px 12px;
10   text-decoration: none;
11 }
```

Ausschnitt 2.18: CSS Ergebnis (vgl. [SAS16])

2.8 Bootstrap

Bootstrap [Boo16] ist ein kostenloses Open Source Framework für die Entwicklung von Websites und Webapplikationen, das ursprünglich von Twitter entwickelt wurde. Es stellt Komponenten für HTML, CSS und JavaScript bereit, um schneller, einfacher und effizienter entwickeln zu können. Dabei werden unter anderem Komponenten für den Entwurf einer Benutzeroberfläche, wie etwa Formularfelder, Buttons oder Icons in Form von CSS-Klassen bereitgestellt.

2.8.1 Responsive Web Design

Um das Layout von Websites und Webapplikationen auch auf Smartphones und Tablets mit kleineren Bildschirmgrößen anzupassen, wendet man Responsive Web Design [Pet14] an. Bootstrap bietet hierfür ein Grid System mit Zeilen und Spalten an. Hierbei wird die Website oder Webapplikation horizontal in 12 Spalten aufgeteilt. Je nach Bildschirmgröße erstreckt sich eine Komponente über eine verschiedene Anzahl von Spalten, wobei Bootstrap zwischen vier verschiedenen Größen unterscheidet und diese als CSS-Klassen bereit stellt: *xs* für Smartphones, *sm* für Tablets, *md* für Desktops und *lg* für größere Desktops. Ausschnitt 2.19 zeigt die Anwendung des Bootstrap Grid Systems. Spalten können sich nur innerhalb von Zeilen (*rows*) befinden. Die CSS-Klasse *col-xs-12* gibt an, dass sich das *div* Element in Zeile 2 auf einem Smartphone über die gesamte Bildschirmbreite erstreckt. Die CSS-Klasse *col-md-8* gibt an, dass es sich bei Bildschirmen mit der Größe eines Desktops nur über 8 von den verfügbaren 12 Spalten erstreckt. Sind für andere Bildschirmgrößen keine CSS-Klassen angegeben, wird die nächstkleinere Größe verwendet. Da in Ausschnitt 2.19 keine CSS-Klassen für Tablets und größere Desktops (*sm* und *lg*) angegeben sind, gilt für Tablets die CSS-Klasse *col-xs-12* und für größere Desktops die CSS-Klasse *col-md-8*.

```
1 <div class="row">
2   <div class="col-xs-12 col-md-8">Smartphone 12, Tablet 8</
      div>
3 </div>
```

Ausschnitt 2.19: Bootstrap Grid System

2.9 Build Tools

2.9.1 Package Manager

Ein Package Manager automatisiert das Hinzufügen, Aktualisieren, Modifizieren und Entfernen von Software Abhängigkeiten in einem Projekt. Im Rahmen dieser Arbeit werden die Package Manager *npm* [npm16] und *Bower* [Bow16] verwendet. *Npm* verwaltet die Pakete für Gulp (siehe Kapitel 2.9.2) und *Bower* jene Pakete, welche für die Entwicklung notwendig sind wie beispielsweise Angular (siehe Kapitel 2.4) oder Bootstrap (siehe Kapitel 2.8).
Der Unterschied zwischen *npm* und *Bower* liegt darin, dass *npm* einen verschachtelten und *Bower* einen flachen Abhängigkeitsbaum hat. Verschachtelt heißt, dass jede Abhängigkeit wieder ihre eigenen Abhängigkeiten haben kann, welche wieder eigene Abhängigkeiten haben können und so weiter. Bei dieser Methode muss man sich nicht um die verschiedenen Versionen der Abhängigkeiten kümmern, da jede Abhängigkeit ihre eigene Version verwenden. Diese Art der Abhängigkeitsverwaltung eignet sich am besten serverseitig oder für Entwicklungstools wie Gulp, da dort die Größe des Speicherbedarfs der Abhängigkeiten eine kleinere Rolle spielt. Der flache Abhängigkeitsbaum von *Bower* eignet sich sehr gut für Websiten und Webapplikationen, da sich die Abhängigkeiten untereinander auflösen, weniger Speicher brauchen und der Benutzer somit weniger herunterladen muss, wenn er die Applikation aufruft.

2.9.2 Task Runner

Task Runner automatisieren zeitaufwendige, immer wieder vorkommende Aufgaben (Tasks) in einem Projekt wie beispielsweise die automatische Minimierung und Zusammenführung von JavaScript und CSS Dateien, SASS und SCSS Übersetzung (siehe Kapitel 2.7) oder die Optimierung von Bildern für die Webdarstellung. Darüber hinaus können eigene Aufgaben definiert werden. Vordefinierte Tasks können mit dem Package Manager *npm* (siehe Kapitel 2.9.1) hinzugefügt werden. Im Rahmen dieser Arbeit wurde der Task Runner *Gulp* [Fra16] verwendet.

2.9.3 Gradle

Gradle ist ein Tool zur Build-Automatisierung und zum Build-Management. Es dient der Automatsierng von Builds, Tests, Publishing und Deployment. Gradle nutzt dazu eine auf Groovy basierte domainspezifische Sprache (DSL). (vgl. [Ede14])

2.9.3.1 Dependency Management

Java Projekte verwenden oftmals mehrere andere, meistens Open Source Projekte wie *Apache Commons* [Fou16b] oder *Google Guava* [Goo15]. Um alle Abhängigkeiten eines Projektes festzulegen und den Download und die Integrierung dieser Abhängigkeiten zu

automatisieren, bietet Gradle Dependency Management an. Dabei können alle gültigen Gradle oder Maven Projekte als Abhängigkeit zu einem Projekt hinzugefügt werden. Spezifiziert werden diese Abhängigkeiten in der Datei *build.gradle*.

2.10 JSON

JavaScript Object Notation (JSON) ist ein schlankes Datenaustauschformat, das für Menschen einfach zu lesen und zu schreiben und für Maschinen einfach zu parsen und zu generieren ist. Es basiert auf einer Untermenge von JavaScript und kann daher von JavaScript ohne Parser gelesen werden. Bei JSON handelt es sich um ein Textformat, das komplett unabhängig von Programmiersprachen ist. JSON besteht aus Objekten, Arrays, Zeichenketten und Zahlen. (vgl. [JSO16a])

2.10.1 Beispiel

Ausschnitt 2.20 zeigt einen JSON Text für die Darstellung eines "window" Objektes mit den vier Attributen "title", "name", "width" und "height".

```
1 {
2     "window": {
3         "title": "MainWindow",
4         "name": "main_window",
5         "width": 500,
6         "height": 500
7     }
8 }
```

Ausschnitt 2.20: Beispiel JSON eines "window" Objektes (vgl. [JSO16b])

2.11 RESTful Webservice

Bei einem RESTful Webservice handelt es sich um ein Webservice, dass rund um Ressourcen aufgebaut ist. Ressourcen sind Entitäten wie Personen oder Benutzer. Alle diese Ressourcen müssen folgende Anforderungen erfüllen:

Adressierbarkeit Jede Ressource muss über einen eindeutigen Unique Resource Identifier (kurz URI) identifiziert werden können. Ein Kunde mit der Kundennummer 123456 könnte also zum Beispiel über die URI http://ws.mydomain.tld/customers/123456 adressiert werden.

Zustandslosigkeit Die Kommunikation der Teilnehmer untereinander ist zustandslos. Dies bedeutet, dass keine Benutzersitzungen (etwa in Form von Sessions und Cookies) existieren, sondern bei jeder Anfrage alle notwendigen Informationen wieder neu mitgeschickt werden müssen. Durch die Zustandslosigkeit sind REST-Services sehr einfach skalierbar; da keine Sitzungen existieren, ist es im Grunde egal, wenn mehrere Anfragen eines Clients auf verschiedene Server verteilt werden.

Einheitliche Schnittstelle Jede Ressource muss über einen einheitlichen Satz von Standardmethoden zugegriffen werden können. Beispiele für solche Methoden sind die Standard-HTTP-Methoden wie GET, POST, PUT, und mehr.

Entkopplung von Ressourcen und Repräsentation Das bedeutet, dass verschiedene Repräsentationen einer Ressource existieren können. Ein Client kann somit etwa eine Ressource explizit beispielsweise im XML- oder JSON-Format anfordern.

(vgl. [Hel13])

Der HTTP-Standard definiert einen ganzen Satz an Operationen, mit denen auf Ressourcen zugegriffen werden kann. Die Methoden GET und POST werden auch von jedem Browser verwendet, wenn Internetseiten aufgerufen, bzw. Formulare abgeschickt werden. Daneben gibt es aber auch noch einen ganzen Satz weiterer Operationen, die für RESTful Webservices verwendet werden.

GET GET dient dazu, lesend auf Ressourcen zuzugreifen. Per Definition darf eine GET-Anfrage nicht dazu führen, dass Daten auf dem Server verändert werden.

POST Mit einem POST-Request können neue Ressourcen erstellt werden, deren URI noch nicht bekannt ist. Per POST-Request an http://ws.mydomain.tld/customers könnte also beispielsweise ein neuer Kunde mit automatisch vergebener Kundennummer (und URI) erstellt werden.

PUT Ein PUT-Request wird verwendet, um Ressourcen zu bearbeiten, deren URI bereits bekannt ist. Ein PUT-Request an http://ws.mydomain.tld/customers/123456 sollte demnach also einen Kunden mit der Kundennummer 123456 bearbeiten.

DELETE Mit einem DELETE-Request können Ressourcen gelöscht werden.

(vgl. [Hel13])

2.12 CORS

Cross-Origin Resource Sharing (CORS) ist eine Möglichkeit, um im Web Client über Domain-Grenzen hinweg verschiedene Web-Applikationen in einer gemeinsamen Oberfläche zu kombinieren. Grundkonzept bei CORS ist es, dass der Server, welcher die Daten bereitstellt, bestimmte HTTP-Header mitschickt, welche dem Browser signalisieren, dass er eine Ausnahme von der Same-Origin-Policy machen darf. Für das "Handshaking" schickt der Browser eine Anfrage mit der HTTP-Methode "OPTIONS" ohne Payload und informiert sich über die "Allow"-Header. Wenn die Header zu der Anfrage passen, dann wird der Request ohne Weiteres ausgeführt. (vgl. [Bor14])

Die Same-Origin-Policy besagt, dass alle AJAX Anfragen einer Website nur auf einen Webservice mit der selben URL erfolgen dürfen. Mithilfe von CORS ist es möglich, explizit Anfragen anderen Ursprungs zu erlauben. Der wichtigste HTTP Header dabei ist *Access-Control-Allow-Origin*, der die URL der Website definiert, von welcher eine AJAX Anfrage aus durchgeführt werden kann. Um Anfragen von allen Websites aus zu ermöglichen, kann stattdessen das Symbol "*" verwendet werden.

2.13 Spring Framework

Das Spring Framework [PS16b] ist ein Open Source Java Framework, welches einfache Java Objekte, sogenannte Plain-Old-Java-Objects (POJO), als Spring Beans verwaltet. Dabei stellt das Spring Framework einen Inversion of Control (IoC) Container zu Verfügung, der per Dependency Injection (DI) abhängige Spring Beans miteinander verknüpft und konfiguriert. Die wesentlichen Funktionen des Spring Frameworks sind ein auf Plain-Old-Java-Objects basierendes Programmiermodell und Verknüpfungen zwischen Objekten durch Dependency Injection über den Inversion-of-Control Container. (vgl. [Rah12])

Zusätzlich dazu gibt es noch weitere Module die das Spring Framework erweitern. Spring Web MVC unterstützt die Möglichkeit, dynamische und REST-konforme Webanwendungen auf Basis der Servlet API [Ora16] umzusetzen. Spring Boot dient zur Erstellung von Anwendungen, die in einer einzigen .jar Datei ausgeliefert werden und per Kommandozeile gestart werden können, ohne etwa einen Servlet Container zu benötigen. Spring Messaging bietet Unterstützung für Kommunikationen über TCP oder Websockets. (vgl. [Rah12])

2.13.1 Dependency Injection

Der Kern des Spring Frameworks ist ein Dependency Injection Framework. Das bedeutet, dass die Verwaltung der Abhängigkeiten zwischen Objekten von Spring übernommen wird. Dafür werden alle Objekte von Spring als sogenannte *Beans* erstellt und verwaltet. Um dafür zu sorgen, dass ein Objekt einer Klasse als Bean erzeugt wird, musste in älteren Versionen von Spring eine XML Datei erstellt werden, in der alle Beans eingetragen wurden. In neueren Versionen kann dies auch durch Annotationen verwirklicht werden. Dazu existiert die Annotation *@Component*, mit der eine Klasse markiert werden kann von der ein Bean erstellt werden soll. Um Abhängigkeiten zu erhalten existiert die Annotation *@Autowired*. Ausschnitt 2.21 zeigt die Verwendung dieser beiden Annotationen.

```
1  @Component
2  public class First {
3
4      public int calculate(int a, int b) {
5          // ...
6      }
7
8  }
9
10 @Component
11 public class Second {
12
13     @Autowired
14     private First first;
15
16     public void doWork() {
17         int result = first.calculate(6, 8);
18     }
19
20 }
```

Ausschnitt 2.21: Verwendung des Spring Dependency Injection Frameworks

2.14 SockJS

SockJS [VMw16a] ist eine Nachahmung der Websocket API. Es besteht aus mehren Bibliotheken, sowohl für die Implementierung eines SockJS Servers als auch für die Implementierung dazugehöriger Clients. Die JavaScript Client Bibliothek [VMw16b] verwendet als Basis die im Browser integrierte Websocket API und erweitert diese um zusätzliche Funktionalität. Außerdem werden auch ältere Browser, die noch keine Websocket API integriert haben, unterstützt. Ein einfaches Verwendungsbeispiel der SockJS Client Bibliothek in JavaScript ist in Ausschnitt 2.22 zu sehen.

```
1  var sock = new SockJS('http://mydomain.com/my_prefix');
2  sock.onopen = function() {
3      console.log('open');
4  };
5  sock.onmessage = function(e) {
6      console.log('message', e.data);
7  };
8  sock.onclose = function() {
9      console.log('close');
10 };
11
12 sock.send('test');
13 sock.close();
```

Ausschnitt 2.22: SockJS Client Beispiel (vgl. [VMw16b])

2.15 STOMP

Das Simple Text Orientated Messaging Protocol (STOMP) ist ein an HTTP angelehntes Übertragungsprotokoll. Es handelt sich dabei um ein Protokoll, das für TCP Übertragungen angewendet werden kann. Außerdem kann das Protokoll auch für Websockets oder SockJS (siehe Kapitel 2.14) verwendet werden. Neben dem einfachen Senden von Nachrichten zwischen zwei Kommunikationspartnern können Nachrichten auch einfach unterschieden werden. Beim Senden einer Nachricht ist immer eine *Destination* inkludiert. Hierbei handelt es sich um eine Zeichenkette, die angibt wofür diese Nachricht bestimmt ist. Sowohl der Client als auch der Server können Callbackmethoden für verschiedene *Destinations* registrieren und dadurch Nachrichten einfach unterscheiden.

2.15.1 Message Broker

Ein im Zusammenhang mit STOMP mehrmals auftauchender Begriff sind sogenannte *Message Broker*. Dabei handelt es sich um Vermittler, die Nachrichten für zwei Kommunikationspartner übersetzen, wenn diese unterschiedliche Protokolle verwenden. Die STOMP Server Implementierung enthält ebenfalls einen Message Broker, damit möglichst viele verschiedene Programmiersprachen und Platformen STOMP verwenden können.

2.16 LDAP

Das Lightweight Directory Access Protocol (LDAP) ist ein Protokoll zur Abfrage und Änderung von Verzeichnisdiensten. Verzeichnisdienste dienen zur Verwaltung von Informationen zu Netzwerkressourcen wie Rechnern, Druckern oder Benutzern. Jede Ressource wird durch einen eindeutigen Namen in einer Hierarchie identifiziert. Die digitalen Zertifikate der Benutzer werden in modernen Verzeichnissen als Attribute gespeichert und somit die Informationen von diesen. Der Verzeichnisdienst erlaubt es, Objekte mit bekanntem Namen oder gewissen Eigenschaften aufzufinden und zu lesen. Verzeichnisdienste stellen Informationen über Benutzer und andere Netzwerkressourcen in einem Unternehmen gesammelt in einem Repository zur Verfügung. Somit wird die Verwaltung von System und Netzwerk vereinfacht und Informationen über Drucker und andere Netzwerkressourcen sind zentralisiert verfügbar. Benutzer können sich mit ihrem Zentralisierten Benutzer-Account an jedem autorisierten Computer im Netzwerk anmelden. Im Rahmen dieser Arbeit wurde der Open Source Apache Directory LDAP Server der Apache Software Foundation [Fou16a] zur Authentifizierung von Benutzern eingesetzt.

Kapitel 3

Benutzung

3.1 Übersicht

In Abblildung 3.1 sind die Arbeitsabläufe der Applikation zu sehen. Der Ausgangspunkt beim Start der Applikation ist das Hauptmenü. Der Benutzer kann daraus die drei Hauptfunktionen „Invantaransicht“, „Baumansicht der Stranggießanlage“ und „NFC“ aufrufen. Zudem können in den Einstellungen globale Konfigurationen vorgenommen werden, wie zum Beispiel die Sprache oder Einheitenkultur.

Die Inventaransicht zeigt beim Aufruf eine Liste aller existierenden Bauteiltypen an. Die vorhanden Bauteile eines Bauteiltypen werden durch einen Klick darauf angezeigt. In der Baumansicht der Stranggießanlage ist dessen Aufbau zu sehen. Dabei werden die Positionen und die darauf installierten Bauteile angezeigt. Mithilfe der NFC-Funktion kann der Benutzer die Bauteile schneller finden und muss sie nicht in der Invantaransicht suchen. Neue NFC-Tags können einem vorhanden oder neu hinzugefügten Bauteil zugewiesen werden. Um Bauteile auf einer Anlage zu installieren beziehungsweise zu deinstallieren, können alle drei Hauptfunktionen verwendet werden. Über die Detailansicht eines bestimmten Bauteils ist es möglich dieses zu bearbeiten, löschen oder die Anhänge (Dokumente oder Bilder) zu verwalten. Zudem kann über eine direkte Referenz auf die Detailansicht des Elternbauteils und des Bauteiltypen gewechselt werden.

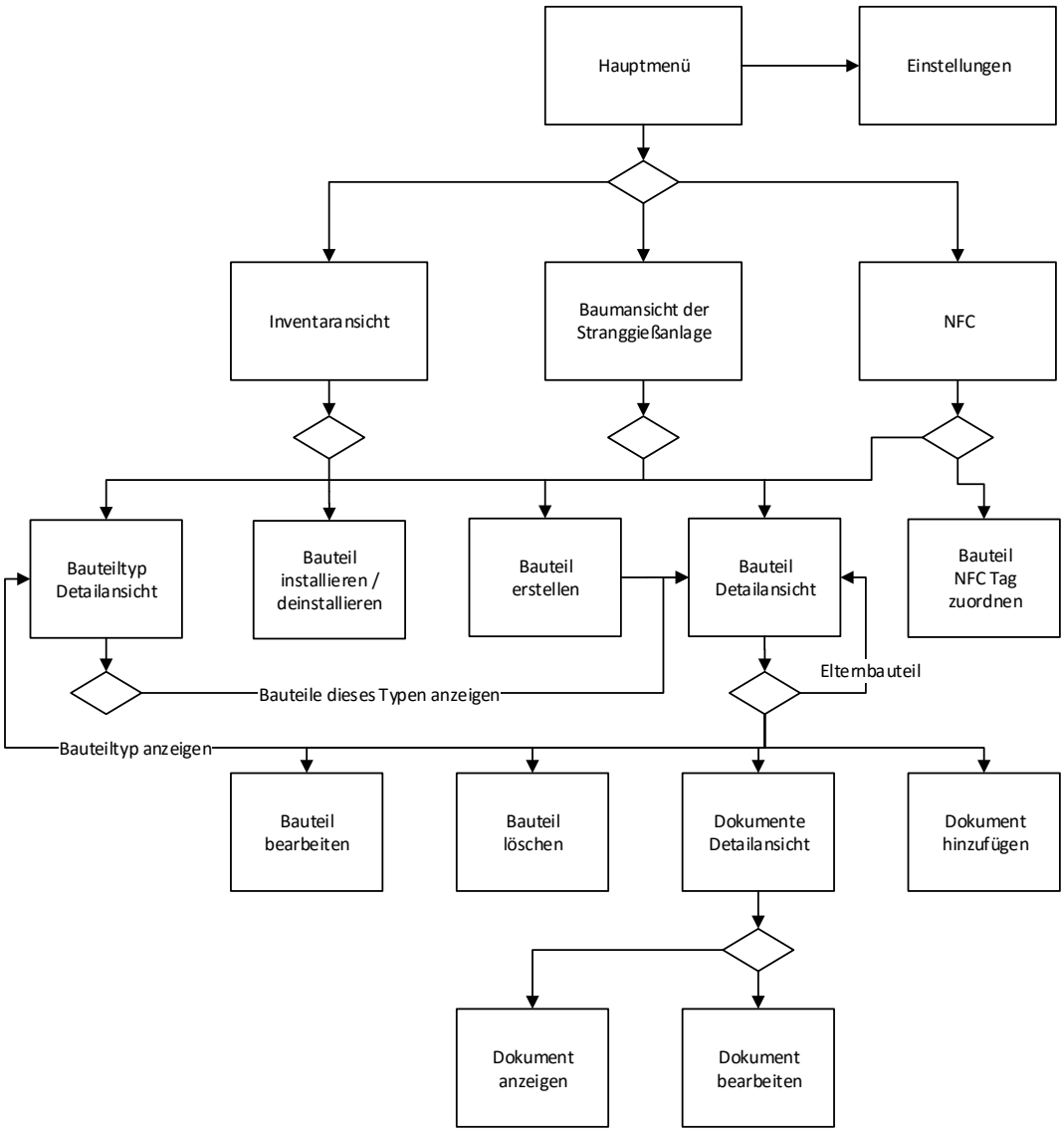


Abbildung 3.1: Skizzierung der Arbeitsabläufe

3.2 Hauptmenü

Beim Start der Applikation wird das Hauptmenü, wie in Abbildung 3.2 zu sehen, angezeigt. Durch einen Klick auf den Button am rechten oberen Bildschirmrand, können Information über die Applikation (*About*) und die globalen Einstellungen aufgerufen werden. Das Auswahlmenü, welches durch Klicken dieses Knopfes angezeigt wird, ist abhängig von der Ansicht, auf der sich der Benutzer befindet. Der grüne Haken in der Navigationsleiste zeigt an, dass die Verbindung zum Server funktioniert, ansonsten ist ein rotes Kreuz zu sehen. Zudem enthält die Navigationsleiste einen *Zurück*-Button, um zur vorherigen Ansicht zu wechseln, und einen *Home*-Button um zu dieser Ansicht zu gelangen.

Über die drei Buttons *Inventory*, *Installed* und *NFC* kann zu den Ansichten Inventar (siehe Kapitel 3.3), Installiert (siehe Kapitel 3.4) und NFC (siehe Kapitel 3.7) gewechselt werden.

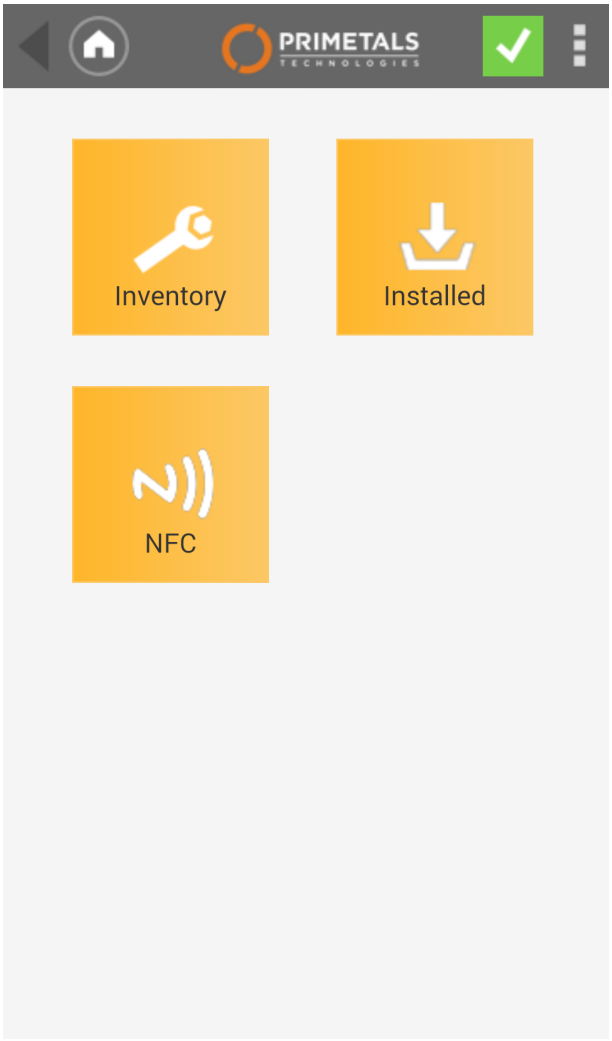


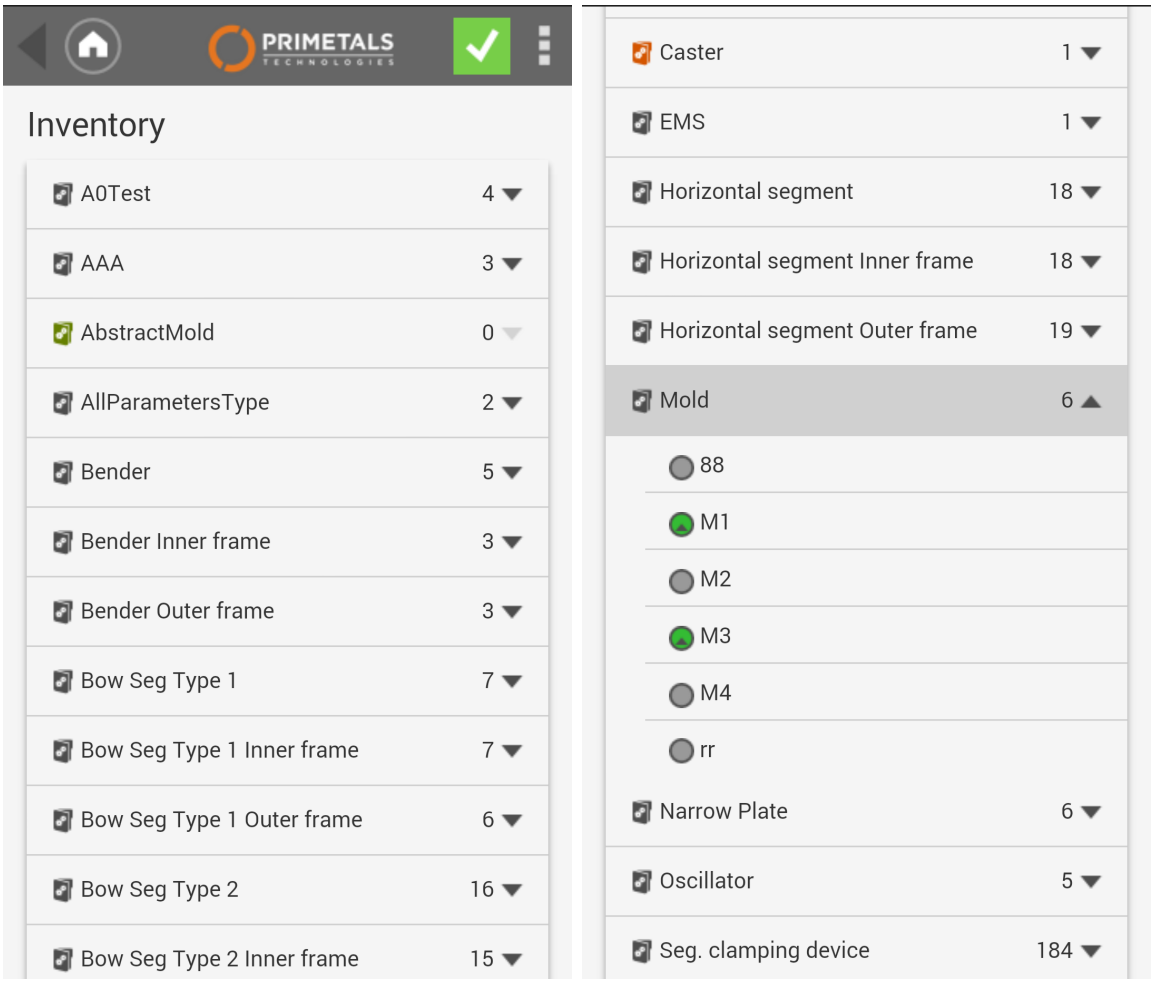
Abbildung 3.2: Hautpmenü

3.3 Inventaransicht

Eine der drei Hauptfunktionen ist die Inventaransicht. Wie in Abbildung 3.3 (a) zu sehen ist, wird eine Liste aller existierenden Bauteiltypen angezeigt. Die Zahl am rechten Rand neben jedem Bauteiltyp gibt an, wie viele Bauteile dieses Bauteiltypen im Unternehmen vorhanden sind. Es werden sowohl die im Lager befindlichen, als auch die installierten Bauteile angezeigt. Anhand des Bildes links neben jedem Bauteil ist zu erkennen, ob es sich um einen *abstrakten* Bauteiltypen (grün) oder um einen *Behälter* (rot) handelt. *Abstrakte* Bauteiltypen sind mit einer Schablone vergleichbar. Von einem *abstrakten* Bauteiltyp kann kein Bauteil angelegt werden, es können lediglich andere Bauteiltypen diesen als Supertyp verwenden. Dadurch wird erreicht, dass Bauteile ähnlicher Bauteiltypen an den gleichen Positionen installiert werden können, und Eigenschaften vom Supertypen übernehmen. Ein *Behälter* ist kein realer Bauteil sondern gruppiert mehrere Bauteile zu einer Einheit.

Durch Klick auf einen Bauteiltypen wird die Liste der vorhanden Bauteile dieses Bauteiltypen aufgeklappt. Die aufgeklappte Liste des Bauteiltypen *Mold* ist in Abbildung 3.3 (b) zu sehen. Anhand des Bildes links neben jedem Bauteil is zu erkennen, ob ein Bauteil auf einer Anlage installiert (grün mit Pfeil) ist. Andernfalls befindet sich das Bauteil im Lager. Dabei wird unterschieden, ob das Bauteil auf einem anderen Bauteil installiert ist (Pfeil), oder einzeln im Lager liegt.

Durch einen Klick auf einen Bauteil wird die Detailansicht angezeigt. Durch einen langen Klick auf einen Bauteil beziehungsweise auf einen Bauteiltypen wird ein Menü angezeigt. Die auswählbaren Aktionen durch einen langen Klick auf einen Bauteil sind in Kapitel 3.5 beschrieben. Das Menü eines Bauteiltypen bietet die Möglichkeit, neue Bauteile anzulegen oder zur Detailansicht dieses Bauteiltypen zu wechseln.



(a) Liste der Bauteiltypen

(b) Aufgeklappte Liste der vorhandenen Bauteile des Bauteiltypen *Mold*

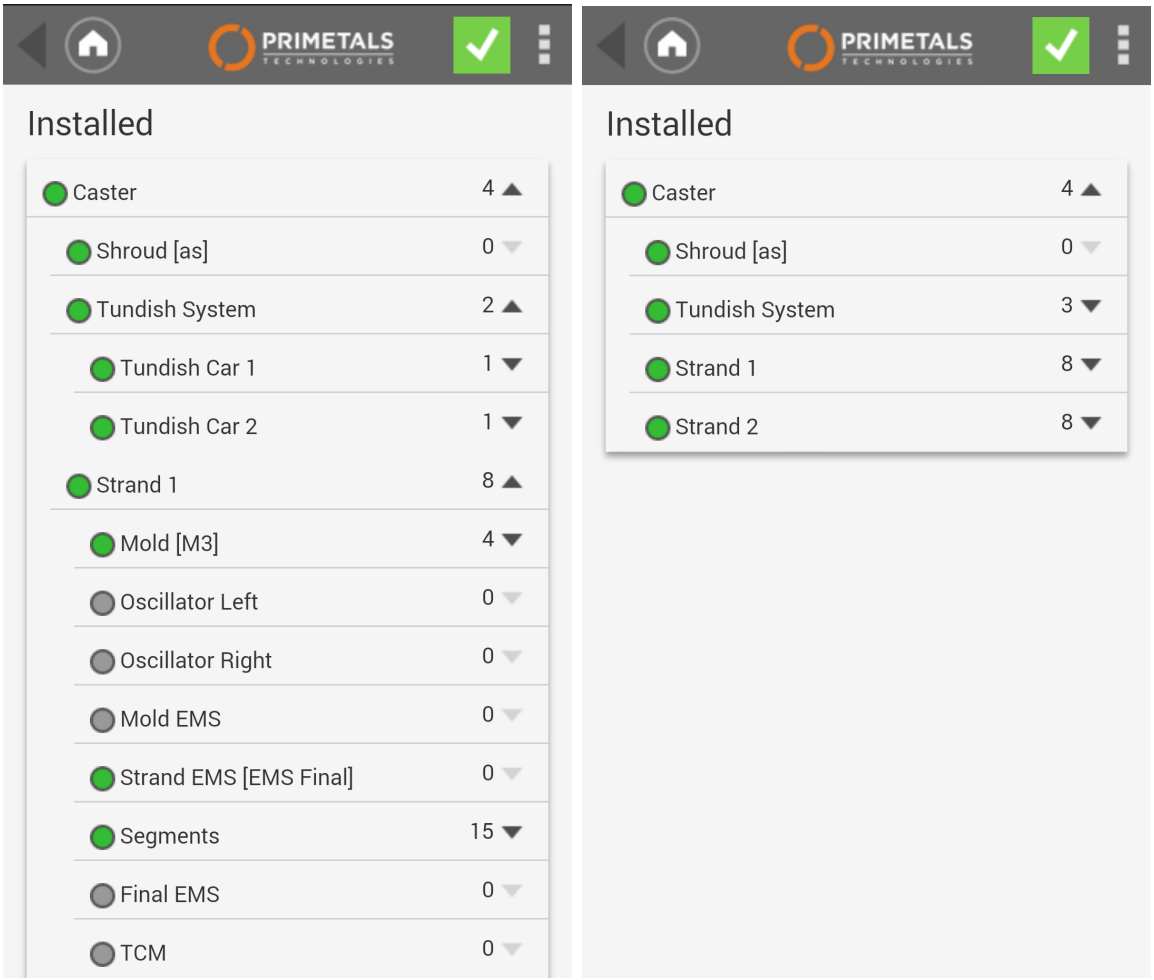
Abbildung 3.3: Inventaransicht

3.4 Baumansicht der Stranggießanlage

Die zweite Hauptfunktion ist die Baumansicht der Stranggießanlage. Dabei wird ausgehend vom Wurzelelement der Aufbau der jeweiligen Anlage angezeigt. In Abbildung 3.4 ist diese Ansicht mit der Wurzelposition *Caster* zu sehen. Es wird unterschieden, ob es sich um installierte Bauteile oder Positionen, auf denen kein Bauteil installiert ist, handelt. An dem Punkt links neben dem Namen der Position ist zu erkennen, ob ein Bauteil installiert ist (grün) oder nicht (grau).

Die Unterpositionen eines bestimmten Bauteils sind vom Bauteiltypen abhängig. Der Name des darauf installierten Bauteils ist in eckiger Klammer neben dem Namen der Position zu sehen. Dieser wird bei einem *Behälter* nicht angezeigt, da diese keine realen Bauteil sind, sondern eine Gruppe von Unterpositionen repräsentieren. Der auf der Position *Caster* installierte *Behälter* vom gleichnamigen Bauteiltypen besteht in diesem Fall immer aus vier Unterpositionen. Diese sind in Abbildung 3.4 (b) zu sehen und umfassen den Bauteil *as* des Bauteiltypen *Shroud* sowie drei weitere *Behälter*.

Auf der Position *Mold* kann ein Bauteil des gleichnamigen Bauteiltypen installiert werden. Dieser Bauteiltyp hat vier weitere Unterpositionen. Am Bauteil *M3* sind drei dieser vier Positionen besetzt. Wird dieser Bauteil installiert, werden automatisch alle darauf installierten Bauteile auf dieser Anlage mit installiert.



(a) Aufgeklappte Baumansicht (b) Unterpositionen von *Caster*

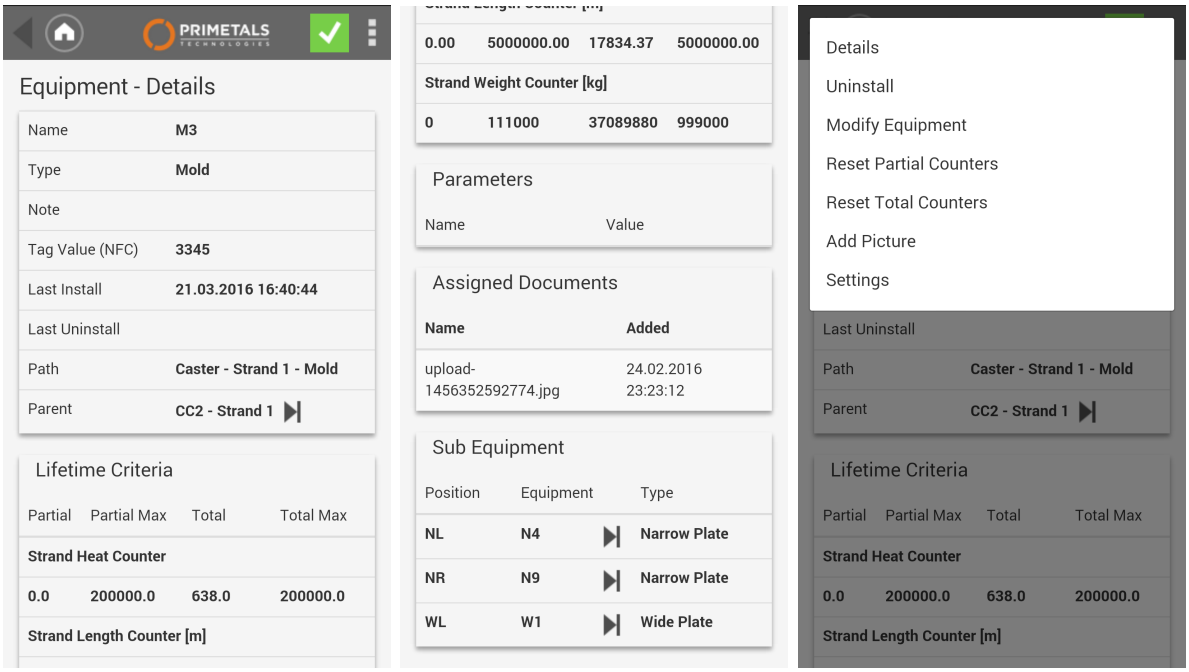
Abbildung 3.4: Baumansicht der Stranggießanlage

3.5 Detailansicht eines Bauteils

In der Detailansicht eines Bauteils sind alle dazu gespeicherten Information zu sehen. In den Abbildungen 3.5 (a) und (b) sind die Details zum Bauteil *M3* vom Bauteiltyp *Mold* zu sehen. Unter anderem wird bei einem installierten Bauteil der *Pfad* zu der Position angezeigt, wo der Bauteil installiert ist. Mit dem darunter befindlichen Pfeil kann zur Ansicht des installierten Bauteils *Strand 1* der nächsten übergeordneten Position gewechselt werden. Zudem sind die Haltbarkeitskriterien beziehungsweise Zähler und optionale Parameter zu sehen. Weiters können auf Bauteilen des Bauteiltypen *Mold* wie in Kapitel 3.4 beschrieben vier Bauteile installiert werden. Abbildung 3.5 (b) zeigt, dass drei dieser vier Positionen besetzt sind. Über die Pfeile kann zu dem jeweiligen Bauteil navigiert werden.

In Abbildung 3.5 (c) ist das Auswahlmenü der verfügbaren Aktion zu sehen. Dieses Auswahlmenü kann durch einen Klick auf den Button in der rechten oberen Ecke in der Detailansicht oder durch einen langen Klick auf den Bauteil in der Invantaransicht geöffnet werden. In diesem Auswahlmenü kann das Bauteil zum Beispiel installiert beziehungsweise deinstalliert werden. Zudem können die Zähler, beispielsweise nachdem der Bauteil gewartet wurde, zurückgesetzt werden, oder auch zur Änderungsansicht gewechselt werden.

Der Link zu den *Settings* befindet sich in jedem Auswahlmenü.



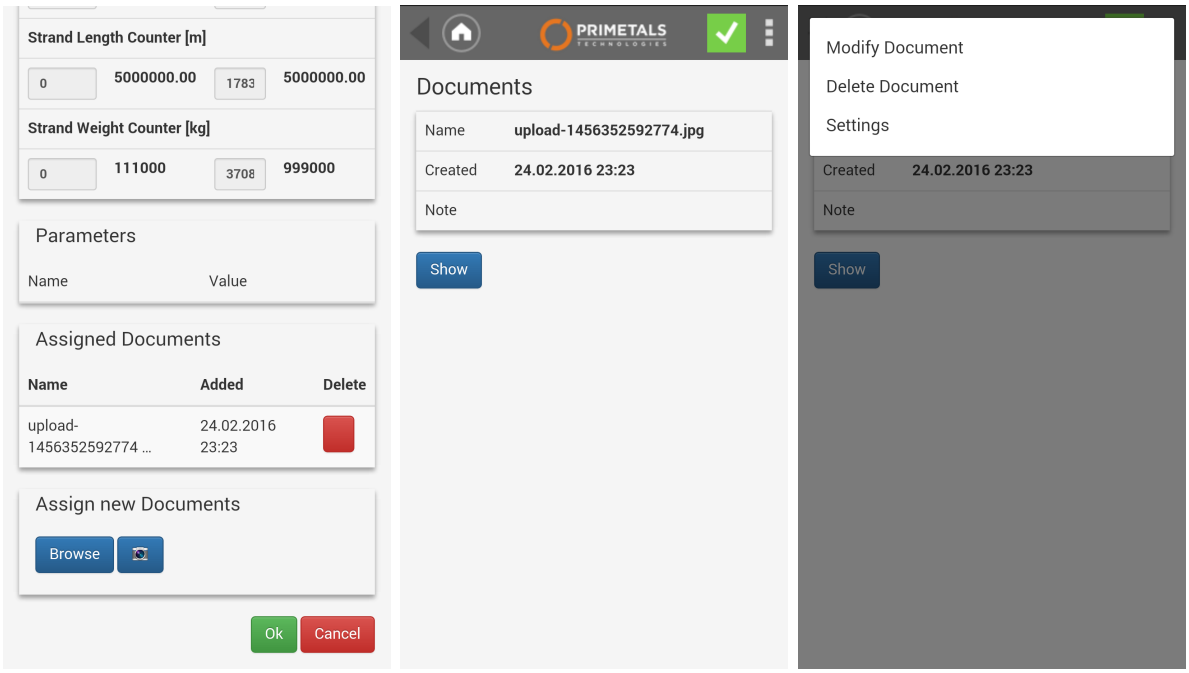
(a) (b) (c) Auswahlmenü

Abbildung 3.5: Detailansicht eines Bauteils

3.6 Dokumente eines Bauteils

Einem Bauteil können beliebig viele Dokumente zugewiesen sein. Dies erleichtert die Wartung dahingehend, da Bilder von Schäden oder Datenblätter hinzugefügt werden können. In Abbildung 3.6 (a) ist die Änderungsansicht des Bauteils *M3 (Mold)* zu sehen. Hier ist es möglich, bereits zugewiesene Dokumente zu löschen oder neue Dokumente, entweder vom Dateisystem oder direkt durch die Aufnahme eines neuen Fotos, hinzuzufügen.

Abbildung 3.6 (b) zeigt die Detailansicht eines Dokuments. Durch einen Klick auf den *Anzeigen* Button wird das Dokument in einem externen Browser angezeigt. Das Auswahlmenü in Abbildung 3.6 (c) kann durch einen Klick auf den Button in der rechten oberen Ecke in Abbildung 3.6 (b) geöffnet werden. In diesem Auswahlmenü kann das Dokument gelöscht oder zur Änderungsansicht gewechselt werden.



(a) Unterer Teil der Änderungsansicht des Bauteils *M3 (Mold)*

(b) Dokumentenansicht

(c) Auswahlmenü

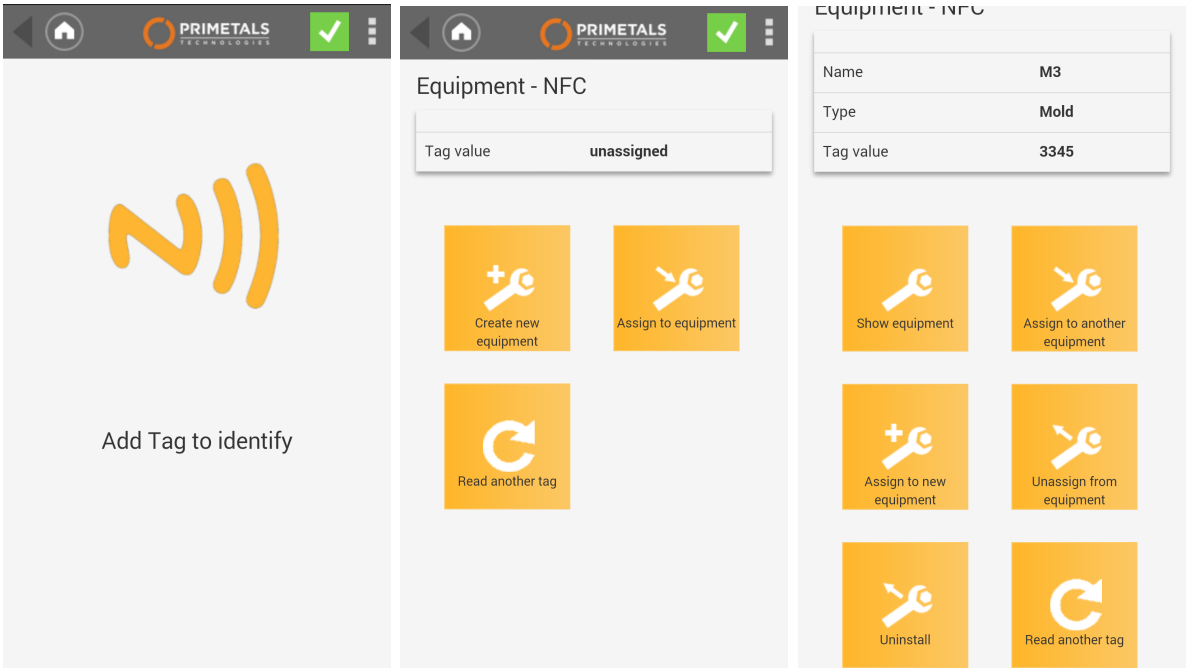
Abbildung 3.6: Dokumentenverwaltung eines Bauteils

3.7 NFC Bauteilerkennung

Die dritte Hauptfunktion der mobilen Applikation ist die Bauteilerkennung mithilfe von Near Field Communication, kurz NFC (siehe Kapitel 2.6). Beim Aufruf der NFC-Funktion im Hauptmenü wird die Ansicht, die in Abbildung 3.7 (a) zu sehen ist, angezeigt. Dazu muss NFC am Gerät aktiviert sein. Jetzt können NFC-Tags (siehe Kapitel 2.6.2) erkannt werden. Dabei wird unterschieden, ob der NFC-Tag bereits einem existierenden Bauteil zugewiesen ist oder nicht.

In Abbildung 3.7 (b) sind die Funktionen für nicht zugewiesene NFC-Tags zu sehen. Dieser Tag kann einem existierenden Bauteil zugewiesen werden. Zudem ist es möglich einen neuen Bauteil eines bestimmten Bauteiltypen anzulegen und den NFC-Tag diesem zuzuweisen.

Wird ein bereits zugewiesener NFC-Tag gefunden, ist die Ansicht in Abbildung 3.7 (c) zu sehen. In diesem Fall handelt es sich um den NFC-Tag des Bauteils *M3* vom Bauteiltypen *Mold*. Jetzt kann direkt zur Detailansicht des Bauteils *M3* gewechselt werden. Zudem is es möglich, den Bauteil direkt zu installieren beziehungsweise zu deinstallieren. Außerdem kann die Zuordnung des NFC-Tag vom aktuellen Bauteil entfernt werden. Der NFC-Tag kann auch direkt einem neuen oder bereits existierenden Bauteil zugewiesen werden. Diese Funktion wird verwendet, wenn das aktuelle Bauteil aus dem Unternehemen ausscheidet.



(a) Suche nach einem NFC-Tag (b) Funktionen für nicht zugewiesene NFC-Tags (c) Funktionen für einen zugewiesenen NFC-Tag

Abbildung 3.7: NFC Bauteilerkennung

Kapitel 4

Programmstruktur

4.1 Architektur

Abbildung 4.1 zeigt die aufgebaute Software- beziehungsweise Hardwarearchitektur. Der zu sehende Level 2 Server speichert die Daten zur Bauteilverwaltung. Der Webservice kommuniziert über eine von Primetals zur Verfügung gestellte Bibliothek mit dem EQX, der sich auf dem Level 2 Server befindet. Die auf den Android beziehungsweise iOS Geräten installierte Applikation bezieht die Daten vom Webservice.

Der bereits existierende Level 2 Server ist ein Windows Server. Darauf läuft ein in Java implementierter Anwendungsserver. Der Anwendungsserver bietet ein Schnittstelle zum Zugriff auf die Datenbank. Über diese Schnittstelle können die Daten abgerufen beziehungsweise manipuliert werden. Die bestehende Desktop Applikation greift direkt auf den Anwendungsserver zu.

Der Webservice läuft entweder auf einem eigenständigen Windows Server oder direkt auf dem Level 2 Server. Dieser ist in Java implementiert. Die von Primetals zur Verfügung gestellte Bibliothek ist ebenfalls in Java geschrieben. Über bestimmte Objekte können Daten vom Anwendungsserver abgerufen werden, oder Methoden aufgerufen werden. Das zur Anfertigung des Webservices verwendete Framework ist *Spring MVC*. Dieser Webservice verwendet *Tomcat* als Webserver. Der Grund für die Anfertigung eines eigenen Webservices ist die Notwendigkeit einer RESTful HTTP Schnittstelle (siehe Kapitel 2.11) zum Datenaustausch mit dem Client. Zur Datenübertragung wird JSON verwendet. Mithilfe des *Jackson JSON Mappers* wandelt *Spring MVC* die Daten automatisch in das JSON Format um.

Die mobile HTML5 Applikation stellt den Client dar. Dieser kommuniziert über die RESTful HTTP Schnittstelle mit dem Webservice. Der Datenaustausch erfolgt über bestimmte URLs. Die zum Webservice gesendeten und davon erhaltenden Daten sind in JSON formatiert.

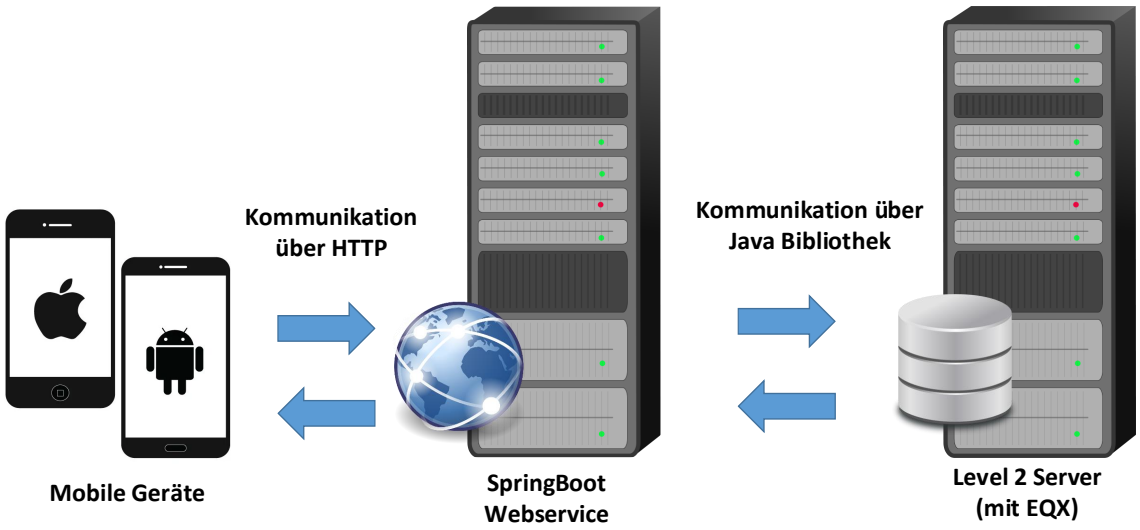


Abbildung 4.1: Architektur

4.2 UML Diagramm der Bauteil Klassenstruktur

Abbildung 4.2 zeigt die Klassenstruktur für die Repräsentation von Bauteilen einer Stranggießanlage. Diese Struktur ist gleichzeitig eine Übersicht über die vorhandenen DTOs (siehe Kapitel 5.2.3), die zur Übertragung der Daten zwischen Server und Client verwendet werden.

Bauteile werden durch die Klassen *Component* und *ComponentDetail* repräsentiert. Die Aufteilung in eine Basisklasse mit den Grundlegenden Eigenschaften und einer detailreichen Klasse wird vorgenommen um Zirkelreferenzen zu unterbinden. Diese Problematik ist in Kapitel 5.2.3 näher beschrieben.

Jeder Bauteil wird durch eine eindeutige Id identifiziert. Zusätzliche Eigenschaften von Bauteilen sind der Name des Bauteils und der Installationsstatus (Installiert auf der Anlage, Installiert auf einem anderem Bauteil, Nicht installiert). Die detailreiche Klasse enthält weitere Eigenschaften, wie eine Referenz auf den übergeordneten Bauteil und den Bauteiltypen. Als übergeordneter Bauteil wird jener Bauteil bezeichnet, auf dem der aktuelle Bauteil installiert ist. Außerdem enthält die detailreiche Bauteilklasse noch eine Liste von Zählern, die die Lebensdauer von Bauteilen bestimmen sowie eine Liste von Parametern mit denen Bauteile näher beschrieben werden können. Zusätzlich enthält jeder Bauteil auch eine Liste von Dokumenten, die diesem Bauteil zugeordnet sind.

Bauteiltypen haben ebenfalls eine eindeutige Id sowie einen Namen. Zusätzlich haben sie noch drei Boolean-Eigenschaften, *Container*, *Abstract* und *Consumable*. Diese definieren die genaue Art des Bauteiltyps. Die detailreiche Bauteiltypklasse enthält außerdem eine Referenz auf eine Liste von Bauteilen dieses Bauteiltyps.

Alle Bauteil Parameter bestehen aus einem Namen und einem Wert. Allerdings kann dieser Wert von unterschiedlichen Datentypen sein. Da der Jackson JSON Mapper (siehe Kapitel 5.2.3) Generische Datentypen nicht korrekt deserialisieren kann, wurde stattdessen für jede Parameterart eine konkrete Subklasse von *ComponentParameter* implementiert. *BooleanComponentParameter*, *TextComponentParameter* und *IntegerComponentParameter* enthalten einen Parameterwert mit dem Datentypen des jeweiligen Klassennamen. *ListComponentParameter* sind Parameter, in denen der Wert einer in einer Liste von vordefinierten Werten ist. *DoubleComponentParameter* dient zur Speicherung von zum Beispiel Länger-, Temperatur- oder Zeitangaben und stellt einen Parameter dar, der Einheitenwerte enthält. Alle *DoubleComponentParameter* können in die drei Einheitenkulturen SI Einheiten, EU Einheiten und US Einheiten umgerechnet werden (siehe Kapitel 5.2.9).

Zähler speichern Betriebsdaten wie etwa die Anzahl der auf diesem Bauteil erzeugten Tonnen Stahl und bestehen aus einem Namen und vier Werten. Mit diesen Werten wird die Lebensdauer von Bauteilen gemessen. *Partial* und *PartialMax* geben den Wert dieses Zählers seit der letzten Wartung an, beziehungsweise den Maximalwert, ab dem der Bauteil erneut gewartet werden muss. Nach der erfolgreichen Wartung wird der *Partial* Wert zurückgesetzt. *Total* und *TotalMax* zählen den Wert seit der ersten Installation des Bauteils in der Anlage mit. Sobald der *Total* Wert den *TotalMax* Wert erreicht hat,

muss das Bauteil ausgetauscht werden. Ähnlich wie beim *DoubleComponentParameter* haben auch Zähler eine Einheit, die in verschiedene Einheitenkulturen umgerechnet werden kann.

Dokumente bestehen aus einer eindeutigen Id, einem Namen, Erstellungsdatum, Kommentar und einem Attribut das den Dateitypen beschreibt.

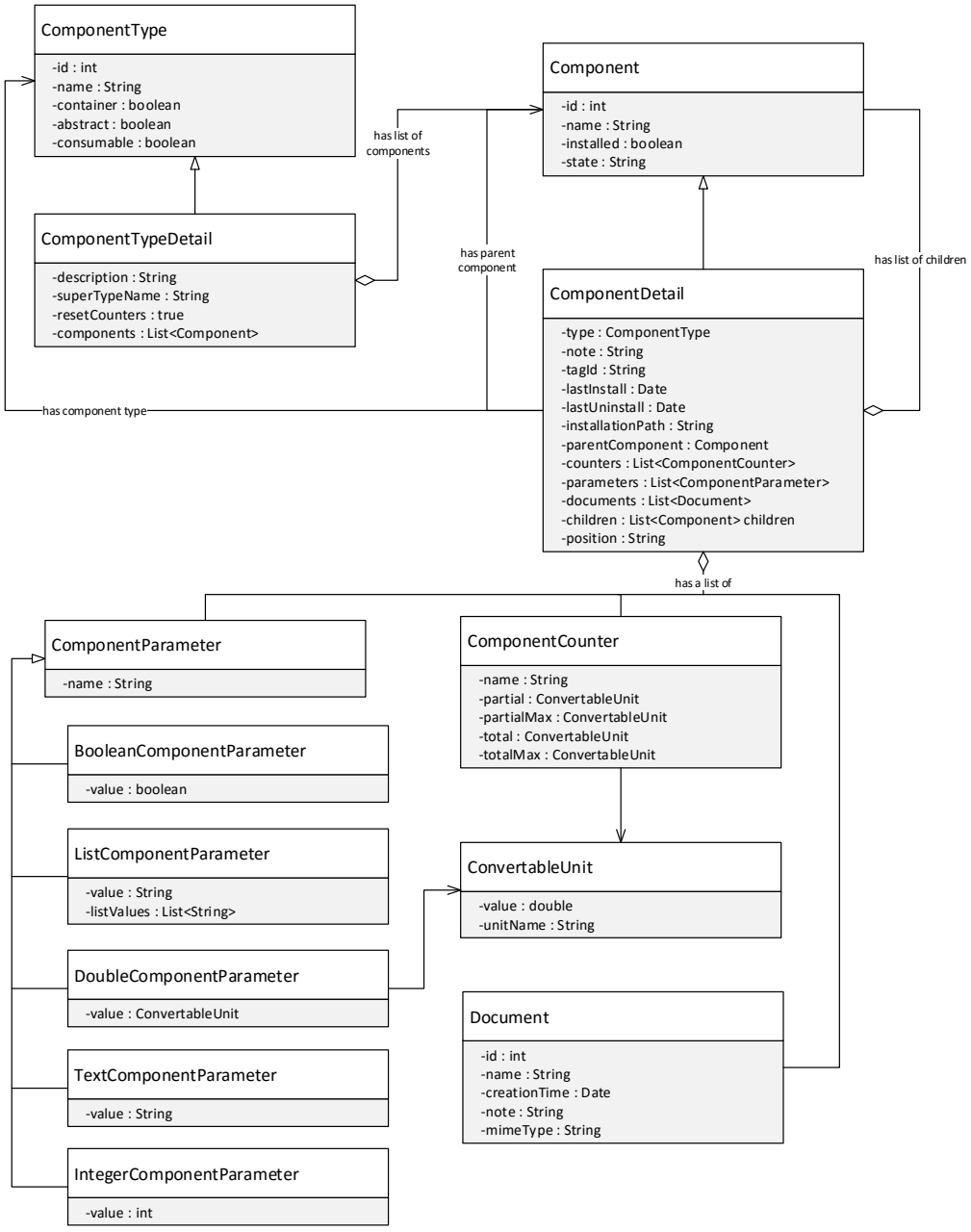


Abbildung 4.2: UML Diagramm der Bauteil Klassenstruktur

4.3 Schnittstellen

4.3.1 REST Webservice

Im Sinne des REST Standards (siehe Kapitel 2.11) bietet das EQX Webservice eine HTTP Schnittstelle für den Zugriff und die Modifikation der Entitäten Bauteile, Bauteiltypen und Dokumente (siehe Kapitel 4.2). In diesem Kapitel wird diese Schnittstelle für jede Ressource näher erläutert und die Funktion eines jeden HTTP Endpunkts sowie die benötigten Parameter und der zu erwartende HTTP Response angeführt.

Jede HTTP Schnittstelle wird in dem Format `HTTP-Verb URI` angeführt. Das HTTP Verb ist entweder GET, POST, PUT oder DELETE und hängt von der durchzuführenden Operation ab. Die URI stellt den Teil der URL dar, der für die Identifikation der Schnittstelle notwendig ist. Davor befindet sich bei einem HTTP Request noch die URL des Webservers, auf dem das Webservice gestartet ist. Eine komplette URL für die URI */api/components/inventory* ist zum Beispiel *http://eqx.htl-perg.ac.at/api/components/inventory*.

4.3.1.1 Bauteile

Inventar

```
GET /api/components/inventory
```

Dieser HTTP Endpunkt ist für die Zusammenstellung des Inventars zuständig. Der Rückgabewert ist eine Liste von Bauteiltypen, wobei jeder Bauteiltyp eine Liste von Bauteilen dieses Typs enthält.

Baumansicht der Stranggießanlage

```
GET /api/components/installed
```

Für die Baumansicht der Stranggießanlage ist der */installed* Endpunkt zuständig. Es sind keine Parameter notwendig und der Rückgabewert ist ein Baumknoten, wobei jeder Baumknoten wiederum eine Liste von untergeordneten Baumknoten enthält. Dadurch wird die rekursive Baumansicht der Stranggießanlage repräsentiert. Jeder Baumknoten enthält zusätzlich noch eine Eigenschaft für den Namen der Position, und falls an dieser Position ein Bauteil installiert ist die Id dieses Bauteils.

Bauteil Detailansicht

```
GET /api/components/{id}
```

Für den Erhalt der Details eines bestimmten Bauteils existiert diese Schnittstelle. Erforderlich ist ein Parameter, nämlich die Id des gewünschten Bauteils. Die URI für den

Bauteil mit der Id *100* lautet */api/components/100*. Falls ein Bauteil mit dieser Id vorhanden ist, wird die Detailansicht dieses Bauteils zurückgegeben. Falls kein Bauteil mit der gewünschten Id vorhanden ist, ist der HTTP Status Code *404 Not Found* und die Antwort enthält eine Nachricht dass der gewünschte Bauteil nicht gefunden wurde.

Neuer Bauteil

```
POST /api/components
```

Mit einem POST Request kann ein neuer Bauteil eingefügt werden. Die erforderlichen Angaben sind dafür ein Name für den Bauteil und die Id des Bauteiltypen und werden im JSON Format als Request Body erwartet. Falls der Bauteil erzeugt werden konnte, beträgt der HTTP Status Code *201 Created* und als Antwort wird die Detailansicht des neuen Bauteils zurückgegeben. Diese Antwort enthält ebenfalls die vom Webservice generierte Id für den Bauteil, mithilfe welcher dieser Bauteil von nun an referenziert werden kann. Falls die Generierung des neuen Bauteils nicht funktioniert hat, weil etwa die angegebene Bauteiltyp Id nicht vorhanden war, wird je nach Fehlerart ein HTTP Status Code von *400-499* gesetzt.

Bauteil modifizieren

```
PUT /api/components/{id}
```

Mit einem PUT Request kann ein vorhandener Bauteil modifiziert werden. Der erforderliche Parameter Id dient zur Identifizierung des Bauteils, der geändert werden soll. Im Request Body werden alle gewünschten Modifizierungen im JSON Format erwartet. Wenn die Modifizierung erfolgreich war, wird die Detailansicht des modifizierten Bauteils zurückgegeben. Falls der Bauteil mit der angegebenen Id nicht gefunden wurde, wird stattdessen der HTTP Status Code *404 Not Found* gesetzt.

Bauteil löschen

```
DELETE /api/components/{id}
```

Mit dieser Schnittstelle kann ein Bauteil gelöscht werden. Falls der Bauteil mit der angegebenen Id gefunden und gelöscht wurde, wird als HTTP Status Code *204 No Content* gesetzt, da die Operation für das Löschen eines Bauteils keinen Rückgabewert enthält. Falls der Bauteil mit der angegebenen Id nicht gefunden wurde, wird stattdessen der HTTP Status Code *404 Not Found* gesetzt.

Zähler zurücksetzen

```
PUT /api/components/{id}/resetPartial
PUT /api/components/{id}/resetTotal
```

Diese beiden Schnittstellen dienen zum Zurücksetzen der partiellen Werte beziehungsweise der totalen Werte aller Zähler eines Bauteils. Der erforderliche Parameter Id dient zur Identifikation des gewünschten Bauteils. Bei erfolgreicher Durchführung der Operation wird die Detailansicht des angegebenen Bauteils zurückgegeben. Falls der Bauteil mit der angegebenen Id nicht gefunden wurde, wird stattdessen der HTTP Status Code *404 Not Found* gesetzt.

Mögliche Installationspositionen eines Bauteils

`GET /api/components/inventory/{componentId}/positions`

Diese Schnittstelle gibt alle Positionen in der Baumansicht der Stranggießanlage an, an welche der Bauteil mit der Id *componentId* installiert werden kann. Der Rückgabewert dieser Schnittstelle ist eine Liste von Installationspositionen. Falls keine Installationsposition existieren, auf die der gewünschte Bauteil installiert werden kann wird eine leere Liste zurückgegeben. Falls der Bauteil mit der angegebenen Id nicht gefunden wurde, wird stattdessen der HTTP Status Code *404 Not Found* gesetzt.

Mögliche Bauteile einer Installationsposition

`GET /api/components/installed/{componentTypeId}/positions`

Diese Schnittstelle gibt alle Bauteile an, die an einer bestimmten Installationsposition installiert werden können. Auf jeder Installationsposition können nur Bauteile eines bestimmten Bauteiltypen installiert werden, daher gibt diese Schnittstelle alle Bauteile dieses Bauteiltypen zurück, die noch nicht installiert sind. Falls keine solchen Bauteile existieren, wird eine leere Liste zurückgegeben. Falls der Bauteiltyp mit der angegebenen Id nicht gefunden wurde, wird stattdessen der HTTP Status Code *404 Not Found* gesetzt.

Installation und Deinstallation von Bauteilen

`PUT /api/components/{componentId}/install/{parentComponentId}`
`PUT /api/components/{componentId}/uninstall`

Mit diesen beiden Schnittstellen kann ein Bauteil installiert beziehungsweise deinstalliert werden. Für beide Operationen ist die Id des gewünschten Bauteils erforderlich, bei der Installation muss zusätzlich noch die Id des übergeordneten Bauteils angegeben werden. Falls der Bauteil mit der angegebenen Id nicht gefunden wurde, wird stattdessen der HTTP Status Code *404 Not Found* gesetzt.

Dokumente von Bauteilen

`GET /api/components/{id}/documents`

Eine Liste mit allen Dokumenten, die einem bestimmten Bauteil zugeordnet sind kann mit dieser Operation erhalten werden. Der erforderliche Parameter *id* gibt die Id des gewünschten Bauteils an. Falls diesem Bauteil keine Dokumente zugeordnet sind, wird eine leere Liste zurückgegeben, ansonsten enthält die Liste Metainformationen wie den Namen und die Id zu allen zugeordneten Dokumenten. Falls der Bauteil mit der angegebenen Id nicht gefunden wurde, wird stattdessen der HTTP Status Code *404 Not Found* gesetzt.

Dokument hochladen und einem Bauteil zuordnen

```
POST /api/components/{componentId}/documents
```

Diese Schnittstelle dient zum Hochladen eines neuen Dokuments und der Zuordnung eines Bauteils zu diesem Dokument. Der erforderliche Parameter *componentId* gibt die Id des Bauteils an, dem das Dokument zugeordnet werden soll. Das Dokument selbst wird im Request Body erwartet. Zurückgegeben werden die Metainformationen des Dokuments, in denen auch die vom Server generierte Id für dieses Dokument enthalten ist. Falls der Bauteil mit der angegebenen Id nicht gefunden wurde, wird stattdessen der HTTP Status Code *404 Not Found* gesetzt.

Dokumentzuordnung zu einem Bauteil entfernen

```
DELETE /api/{componentId}/documents/{documentId}
```

Eine Zuordnung zwischen einem Bauteil und einem Dokument kann mit dieser Schnittstelle entfernt werden. Sowohl der Bauteil und das Dokument bleiben bestehen, nur die Zuordnung zwischen den beiden wird gelöscht. Die erforderlichen Parameter *componentId* und *documentId* dienen zur Identifikation des Bauteils und des Dokuments. Falls die Operation erfolgreich durchgeführt wurde, wird als HTTP Status Code der Wert *204 No Content* gesetzt. Falls entweder der Bauteil oder das Dokument mit der gewünschten Id nicht gefunden wurde, wird stattdessen der HTTP Status Code *404 Not Found* gesetzt.

4.3.1.2 Bauteiltypen

Bauteiltyp Detailansicht

```
GET /api/componentTypes/{id}
```

Für den Erhalt der Details eines bestimmten Bauteiltypen existiert diese Schnittstelle. Erforderlich ist ein Parameter, nämlich die Id des gewünschten Bauteiltypen. Die URI für den Bauteiltyp mit der Id *100* lautet */api/componentTypes/100*. Falls ein Bauteiltyp mit dieser Id vorhanden ist, wird die Detailansicht dieses Bauteiltypen zurückgegeben. Diese Detailansicht enthält den Namen des Bauteiltypen sowie eine Liste aller Bauteile desselben. Falls kein Bauteiltyp mit der gewünschten Id vorhanden ist, ist der HTTP

Status Code *404 Not Found* und die Antwort enthält eine Nachricht dass der gewünschte Bauteiltyp nicht gefunden wurde.

Zähler zurücksetzen

```
PUT /api/componentTypes/{id}/resetPartial
PUT /api/componentTypes/{id}/resetTotal
```

Diese beiden Schnittstellen dienen zum Zurücksetzen der partiellen Werte beziehungsweise der totalen Werte aller Zähler aller Bauteile eines Bauteiltypen. Der erforderliche Parameter Id dient zur Identifikation des gewünschten Bauteiltypen. Bei erfolgreicher Durchführung der Operation wird die Detailansicht des angegebenen Bauteiltypen zurückgegeben. Falls der Bauteiltyp mit der angegebenen Id nicht gefunden wurde, wird stattdessen der HTTP Status Code *404 Not Found* gesetzt.

4.3.1.3 Dokumente

Dokument Metadaten

```
GET /api/documents/{id}
```

Für den Erhalt der Metadaten eines bestimmten Dokuments existiert diese Schnittstelle. Erforderlich ist ein Parameter, nämlich die Id des gewünschten Dokuments. Die URI für das Dokument mit der Id *100* lautet */api/documents/100*. Falls ein Dokument mit dieser Id vorhanden ist, werden die Metadaten dieses Dokuments zurückgegeben. Diese bestehen aus dem Namen, dem Erstellungsdatum, einer Notiz und dem Mimetype des Dokuments. Falls kein Dokument mit der gewünschten Id vorhanden ist, ist der HTTP Status Code *404 Not Found* und die Antwort enthält eine Nachricht dass das gewünschte Dokument nicht gefunden wurde.

Dokument herunterladen

```
GET /api/documents/download/{id}
```

Mit dieser Schnittstelle kann das Dokument mit der angegebenen Id heruntergeladen werden. Der HTTP Response enthält den Inhalt des Dokuments. Falls der Browser den Mimetype des Dokuments versteht, wird das Dokument direkt im Browser angezeigt. Dies ist zum Beispiel bei Bildern oder PDF-Dateien der Fall. Alle anderen Dateiformate werden vom Browser in die Downloadliste hinzugefügt. Falls das Dokument mit der gewünschten Id nicht gefunden wurde, wird stattdessen der HTTP Status Code *404 Not Found* gesetzt.

Metadaten eines Dokuments modifizieren

`PUT /api/documents/{id}`

Mit einem PUT Request können die Metadaten eines vorhandenen Dokuments modifiziert werden. Der erforderliche Parameter Id dient zur Identifizierung des Dokuments, das geändert werden soll. Im Request Body werden alle gewünschten Modifizierungen im JSON Format erwartet. Wenn die Modifizierung erfolgreich war, werden die Metadaten des Dokuments zurückgegeben. Falls das Dokument mit der angegebenen Id nicht gefunden wurde, wird stattdessen der HTTP Status Code *404 Not Found* gesetzt.

Dokument löschen

`DELETE /api/documents/{id}`

Mit dieser Schnittstelle kann ein Dokument gelöscht werden. Falls das Dokument mit der angegebenen Id gefunden und gelöscht wurde, wird als HTTP Status Code *204 No Content* gesetzt, da die Operation für das Löschen eines Dokuments keinen Rückgabewert enthält. Falls das Dokument mit der angegebenen Id nicht gefunden wurde, wird stattdessen der HTTP Status Code *404 Not Found* gesetzt.

Kapitel 5

Implementierung

5.1 Technologieauswahl

Zur Auswahl der verwendeten Technologien wurden verschiedene Prototypen angefertigt. Diese waren notwendig um die für die Implementierung und den Auftraggeber passendste Technologie zu finden. Dabei handelt es sich um die Technologie, die für die Entwicklung der mobilen Applikation eingesetzt wurde. Zur Entscheidungsfindung der clientseitigen Technologie wurden zwei Prototypen implementiert. Schlussendlich fiel die Entscheidung darauf, dass die Applikation mithilfe von Cordova und HTML implementiert wird.

Die serverseitige Implementierung erfolgte aufgrund der zur Verfügung gestellten Bibliothek in Java. Daher kam die Verwendung einer anderen Technologie nicht in Frage, lediglich die Auswahl des Frameworks. Zur Implementierung des Webservices wurde nach kurzen Kompatibilitätstests das Framework *SpringBoot* verwendet.

Zur Implementierung eines Prototypen der Applikation wurde *Xamarin* [Inc15] verwendet. Xamarin ermöglicht die Erzeugung von verschiedenen nativen Applikationen, beispielsweise Android und iOS, mit einer gemeinsamen *C#* Codebasis. Es bietet an, plattformunabhängigen Quellcode und native Benutzeroberflächen in einer Entwicklungsumgebung und Programmiersprache zu implementieren. Diese Technologie wurde aufgrund der Erfahrungen in der *C#*-Programmierung, der Performance und der unterschiedlichen Benutzeroberflächen für beide Plattformen (*Look and Feel*) bevorzugt. Der Nachteil sind die hohen Lizenzkosten, die derzeit 999$ pro Jahr und Entwickler betragen. Diese Lizenzkosten waren dem Auftraggeber zu hoch, wodurch die Verwendung dieser Technologie außer Frage stand.

Ein weiterer Prototyp der Applikation wurde mit der Entwicklungsumgebung *RAD Studio* des Unternehmens *Embarcadero* [ET15] implementiert. *RAD Studio* ermöglicht es, mit einer Delphi oder C++ Code Basis performante, native Applikationen für Windows, Mac OS, iOS und Android zu erzeugen. Dabei wird das *Look and Feel* für die jeweilige Plattform beibehalten. Aufgrund mangelnder Erfahrung in C++ und Delphi und der hohen Lizenzkosten dieser Technologie wurde diese Technologie jedoch abgelehnt.

5.2 Serverseitig

5.2.1 Ablauf der HTTP Request Beantwortung

Alle eingehenden HTTP Requests durchlaufen einen gewissen Ablauf, welcher in Abbildung 5.1 skizziert ist. Zu Beginn wird jeder Request von dem *Request Processor* entgegengenommen. Als erstes durchläuft jeder Request eine Reihe von Filtern, die sogenannte Filterkette oder auch Filterchain. Ein Filter überprüft einen Request und modifiziert ihn gegebenenfalls. Außerdem kann jeder Filter die Filterkette stoppen und dadurch die Behandlung des Requests abbrechen. Eine genauere Beschreibung der Filterkette sowie aller implementierten Filter befindet sich im Kapitel 5.2.5. Am Ende der Filterkette wird der Request wieder dem *Request Processor* übergeben. Dieser bestimmt nun anhand der HTTP Routenzuordnung (Kapitel 5.2.2), welche Programmmethode für das Bearbeiten dieses Requests zuständig ist. Für das durchführen der durch den Request angefragten Operation verwendet diese Methode verschiedene Services, welche in Kapitel 5.2.4 näher erläutert werden. Durch die Auslagerung aller Operationen in Serviceklassen wird die Wiederverwendbarkeit gewährleistet. Die Services laden Daten von verschiedensten Repositories (siehe Kapitel 5.2.6). Anschließend werden die erhaltenen Daten in sogenannte DTOs umgewandelt. Dies geschieht mithilfe von Mapper Klassen (Kapitel 5.2.3). Die Aufteilung in Controller, Services und Repositories kommt vom Prinzip des Domain-Driven Design, welches Eric J. Evans in seinem gleichnamigen Buch [Eva04] beschreibt.
Im Unterschied zu Repositories im Domain-Driven Design erhalten die implementierten Repositories in dieser Arbeit die Daten nicht aus einer Datenbank, sondern von einem von Primetals entwickeltem Java Server. Die Kommunikation mit diesem Server erfolgt mithilfe einer von Primetals bereitgestellten Bibliothek.

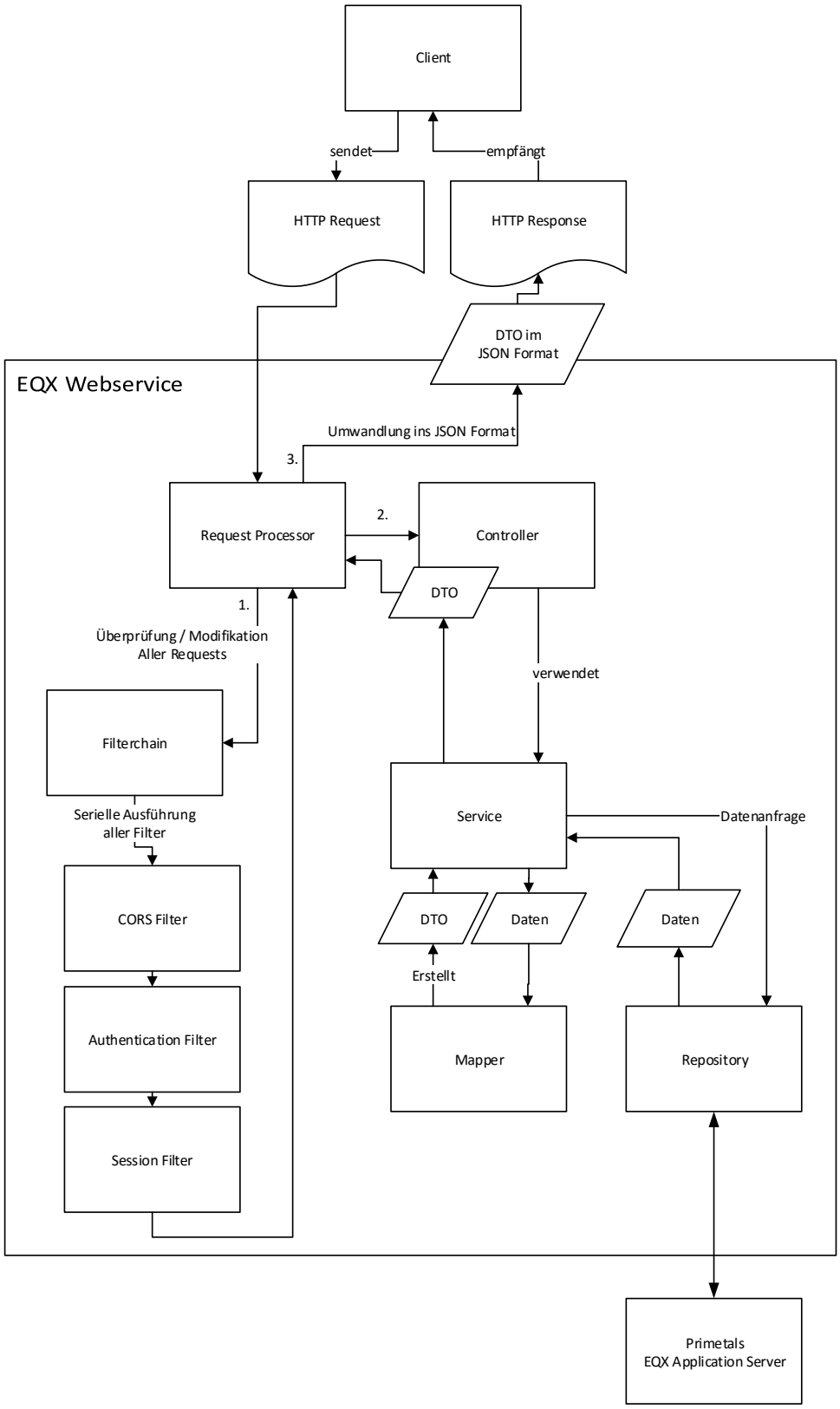


Abbildung 5.1: Ablauf der HTTP Request Behandlung

5.2.2 HTTP Routenzuordung

Die einzelnen Funktionen des Webservers können durch HTTP Requests auf verschiedene URLs des Servers aufgerufen werden. Dafür ist eine Zuordnung einer URL auf eine bestimmte Programmmethode notwendig. Spring MVC bietet für diesen Zuordnungsprozess diverse Annotationen an. Mit der Annotation *@RestController* kann signalisiert werden, dass eine Klasse ein HTTP Controller ist, also Methoden enthält, welche über einen HTTP Request aufgerufen werden können. Rest steht in diesem Zusammenhang für einen RESTful Webservice (siehe Kapitel 2.11).

Mit der *@RequestMapping* Annotation erfolgt die Zuordnung einer bestimmten URL zu einer Methode der Controller-Klassen. Das Annotationsattribut *value* stellt einen String dar, welcher die URL dieser Methode darstellt. Mit dem Attribut *method* kann zusätzlich das HTTP Verb spezifiziert werden, mit welchem die Methode angesprochen werden kann.

Ausschnitt 5.1 stellt eine Methode dar, welche der URL */hello* als HTTP GET Request zugeordnet ist und den String *"hello"* zurückgibt. Der Rückgabewert dieser Methode stellt gleichzeitig die Antwort auf den HTTP Request dar.

```
1 @RequestMapping(value = "/hello", method = RequestMethod.
     GET)
2     public String hello() {
3     return "hello";
4 }
```

Ausschnitt 5.1: Beispiel Routemapping

Für einige Funktionen ist es auch notwendig Parameter an die Methode zu übergeben. Dafür gibt es zwei Möglichkeiten. Man kann die Parameter als HTTP Request Parameter übergeben oder sie als Teil der URL übergeben und dann aus der URL extrahieren. Solche Parameter werden Pfad Variablen genannt. Ausschnitt 5.2 zeigt die Implementierung dieser beiden Varianten zur Parameterübergabe.

```
1  // HTTP Request Parameter
2  // /hello?name=max
3  @RequestMapping(value = "/hello", method = RequestMethod.
       GET)
4      public String hello(@RequestParam("name") String name)
           {
5      return "hello " + name;
6  }
7
8  // HTTP Path Variable
9  // /hello/max
10 @RequestMapping(value = "/hello/{name}", method =
11     RequestMethod.GET)
12     public String hello(@PathVariable("name") String name)
           {
13     return "hello " + name;
14 }
```

Ausschnitt 5.2: Routemapping mit Parameter

Im Sinne des REST Standards (siehe Kapitel 2.11) wurden in dieser Arbeit hauptsächlich Pfad Variablen zur Variablenübergabe verwendet.

Für einige Funktionen, wie zum Beispiel die Erstellung neuer Bauteile, ist es auch erforderlich komplette Objekte zu übergeben. Diese werden im HTTP Request Body übergeben. Für diesen Zweck gibt es die *@RequestBody* Annotation, die auf einen Controllermethoden-Parameter angewandt werden kann und signalisiert, dass der Inhalt dieses Parameters aus dem HTTP Request Body gelesen werden soll. Ausschnitt 5.3 zeigt die Implementierung einer solchen Methode. Als HTTP Verb wurde für diese Funktion POST verwendet. Der Unterschied zwischen GET und POST besteht darin, dass bei POST Parameter und Daten nicht in der URL sondern im Request Body mitgeschickt werden. Eine genauere Beschreibung aller HTTP Verben kann in Kapitel 2.11 nachgelesen werden.

```
1  @RequestMapping(value = "", method = RequestMethod.POST)
2  public ComponentDetail createComponent(@RequestBody
       ComponentDetail component) {
3      return componentService.createComponent(component);
4  }
```

Ausschnitt 5.3: Request Body Parameter

Einige Funktionen, wie das Löschen von Bauteilen haben keinen Rückgabewert. In diesem Fall bietet sich an, als HTTP Status Code den Status 204 NO CONTENT zu verwenden. Dafür existiert die Annotation *@ResponseStatus*, mit welcher der gewünschte Status angegeben werden kann. Ausschnitt 5.4 zeigt die Verwendung dieser Annotation.

```
1 @RequestMapping(value = "/hello", method = RequestMethod.
     GET)
2 @ResponseStatus(value = HttpStatus.NO_CONTENT)
3 public void hello() {
4 }
```

Ausschnitt 5.4: HTTP Status Code

Die Annotation *@ResponseStatus* bietet noch eine weitere Variante zum festlegen des HTTP Status Codes, welche vor allem in der Fehlerbehandlung sinnvoll ist. Die Annotation *@ResponseStatus* kann nämlich auch auf die Klassendefinition von Exceptions angewandt werden. Falls eine Methode nun mit solch einer Exception abbricht, wird automatisch der entsprechende HTTP Status Code gesetzt. Ausschnitt 5.5 zeigt die Implementierung einer Exception, die einen bestimmten HTTP Status Code setzt.

```
1 @ResponseStatus(value = HttpStatus.NOT_FOUND)
2 public class EntityNotFoundException extends
     RuntimeException{
3 }
```

Ausschnitt 5.5: HTTP Status Code für Exceptions

5.2.2.1 Implementierte HTTP Controller

Die implementierten HTTP Controller befinden sich im Package *vai.eqx2.webservice.controllers*. Der *ComponentController* ist für Funktionen zuständig, welche die Verwaltung der Bauteile durchführen. *ComponentTypeController* behandelt Funktionen, welche Bauteiltypen betreffen und der *DocumentController* zum Hochladen und zum Herunterladen von Dateien verwendet.

Prinzipiell wurden für Bauteile, Bauteiltypen und Dokumente Controllermethoden zum Erzeugen, Bearbeiten, Löschen und Auslesen implementiert. Zusätzlich gibt es für spezielle Funktionen, wie das Suchen eines Bauteils mit dem auf einem NFC-Tag gespeicherten Wert (siehe Kapitel 2.6) eigene Controllermethoden. Diese Methoden implementieren die Funktionalität nicht selbst, sondern rufen entsprechende Servicemethoden auf (Kapitel 5.2.4). Dies hat den Vorteil, dass bestimmte Funktionalitäten, welche mehrmals benötigt werden, einfacher wiederverwendet werden können. Eine Übersicht dieser Architektur ist in Kapitel 5.2.1 zu finden.

5.2.3 JSON Übertragung

Die Übertragung von Daten zwischen dem Server und dem Client erfolgt mithilfe des HTTP Protokolls. Da HTTP ein textbasiertes Protokoll ist, müssen alle zu Übertragenden Objekte zuvor serialisiert werden. Grundsätzlich haben sich dafür zwei Formate etabliert, XML (Extensible Markup Language) und JSON (JavaScript Object Notation). Da der

im Rahmen dieser Arbeit implementierte Client JavaScript verwendet, bietet sich JSON an, da JavaScript dieses ohne etwaige Zusatzbibliotheken interpretieren kann.

Für die Umwandlung von Objekten in das JSON Format (siehe Kapitel 2.10) hat Spring MVC bereits den Jackson JSON Mapper inkludiert. Dieser ist in der Lage, beliebige Objekte in das JSON Format umzuwandeln. Umgekehrt kann der Jackson JSON Mapper auch aus einem JSON String wieder Objekte generieren. Ausschnitt 5.6 zeigt die Klassendefinition eines Bauteils und Ausschnitt 5.7 den daraus resultierenden JSON String des Bauteils "Tundish CC2". Für die Umwandlung muss keine spezielle Methode des Jackson JSON Mappers aufgerufen werden, da die Rückgabewerte aller HTTP Controllermethoden automatisch in das JSON Format übertragen werden.

```
1 public class Component {
2
3     private int id;
4     private String name;
5     private boolean installed;
6     private String state;
7
8     // ... Getters and Setters
9 }
```

Ausschnitt 5.6: Bauteil Klassendefinition

```
1 {
2     "id":3329,
3     "name":"Tundish CC2",
4     "installed":true,
5     "state":"INSTALLED_ON_PLANT"
6 }
```

Ausschnitt 5.7: Bauteil JSON Repräsentation

Für die Analyse beliebiger Objekte verwendet der Jackson JSON Mapper Runtime Type Information (RTTI). RTTI beschreibt die Möglichkeit, zur Laufzeit Informationen über den Datentyp, also alle Attribute und Methoden eines Objekts zu ermitteln. In Java ist RTTI als Reflection bekannt. Alle *public* Methoden eines Objektes, die mit "get" beginnen werden als Attribut in das JSON Format übertragen. Bei der Deserialisierung wird für jedes Attribut eine Setter Methode gesucht und mit dem im JSON angegebenen Wert aufgerufen.

5.2.3.1 Data Transfer Objects

Data Transfer Objects (DTOs) sind Objekte, die nur zur Übertragung von Daten verwendet werden. Die Klassendefinition solcher Objekte enthalten keinerlei Funktionalität, sondern nur Attribute sowie deren Getter- und Setter Methoden. Eine Übersicht über alle

implementierten DTOs und eine Beschreibung zu jedem DTO ist in Kapitel 4.2 nachzulesen.

Für ein und dieselbe Klasse können mehrere DTO Klassen existieren, die unterschiedliche Attribute dieser Klasse enthalten. Dieses Prinzip wurde auch in dieser Arbeit mehrmals verwendet. So existieren für Bauteile und Bauteiltypen je eine DTO Klasse, die nur die grundlegenden Eigenschaften wie *Id* und *Name* enthalten, sowie je eine detailreiche Klasse mit allen Eigenschaften. Ausschnitt 5.8 zeigt die detailreiche Bauteilklasse, die von der in Ausschnitt 5.6 zu sehenden Basisklasse erbt und weitere Attribute ergänzt.

```
1  public class ComponentDetail extends Component {
2
3      private ComponentType type;
4      private String note;
5      private String tagId;
6      private Date lastInstall;
7      private Date lastUninstall;
8      private String installationPath;
9      private Component parentComponent;
10     private List<ComponentCounter> counters;
11     private List<ComponentParameter> parameters;
12     private List<Document> documents;
13     private List<Component> children;
14     private String position;
15
16     // ... Getters and Setters
17
18 }
```

Ausschnitt 5.8: Bauteil Detailreiche Klasse

Ein häufig auftretendes Problem bei der Serialisierung von Objekten sind Zirkelreferenzen. Das bedeutet, dass zwei Objekte sich gegenseitig referenzieren und dadurch beim Serialisieren eine Endlosschleife auftritt. Dies wäre zum Beispiel bei Bauteilen mit dem Elternelement und den Kindelementen der Fall. Jeder Bauteil kann eine Referenz auf ein Elternelement sowie Referenzen auf mehrere Kindelemente haben. Diese Kindelemente hätten dann aber ebenfalls wieder eine Referenz auf ihr Elternelement und dadurch entsteht eine Endlosschleife. Behoben wird dieses Problem durch die Aufteilung auf eine Klasse mit den wesentlichen Attributen und eine Klasse mit den detailreichen Attributen. Wie man im Ausschnitt 5.8 in Zeile 9 und 13 erkennen kann, werden hier das Elternelement und die Kindelemente als *Component*, also als Objekte der Basisklasse gespeichert und die Zirkelreferenz dadurch unterbunden. Wichtig dabei ist aber, dass die Objekte die in diesen Attributen gespeichert werden auch Objekte der Basisklasse sind und nicht der detailreichen Klasse, da die Objekte mit RTTI analysiert werden.

5.2.3.2 Mapper Klassen

Für die Transformation von Objekten in ihre DTOs gibt es bereits Java Bibliotheken wie MapStruct [Mor16]. Allerdings bieten solche Bibliotheken nur primitive Umschreibungen von Attributen eines Objektes auf ein Attribut des DTO an. Für Bauteile ist es aber notwendig, die *Id* des Bauteiltyps auszulesen und den Namen dieses Typs dann vom EQX-Server zu erfragen. Da solche Operationen nicht mithilfe von vorhandenen Bibliotheken möglich sind, wurden eigene Klassen implementiert, die diese Umwandlung von Objekten in ihre DTOs übernehmen. Ausschnitt 5.9 zeigt die Klasse für die Umwandlung von Bauteiltypen in das DTO. Die Operationen in den Zeilen 11, 12 und 17-20 sind eine Übertragung von Attributwerten in das DTO. Allerdings wird in den Zeilen 13-16 eine Liste von Bauteilen dieses Bauteiltyps erzeugt und ebenfalls in das DTO gespeichert. Dafür werden zuerst vom Server die Bauteile des aktuellen Bauteiltyps erfragt und dann mithilfe der Bauteil-Mapperklasse in Bauteil-DTOs übertragen.

```
1  public class ComponentTypeMapper extends
2      EntityDTOMapper<IComponentType, ComponentType> {
3
4      @Override
5      public ComponentType createDTO(IComponentType entity) {
6          return mapToDTO(entity, new ComponentType());
7      }
8
9      public ComponentType mapToDTO(IComponentType entity,
10             ComponentType dto) {
11         dto.setId(entity.getTypeId());
12         dto.setName(entity.getName());
13         List<IComponent> componentsOfThisType =
14             componentRepository.getComponentsOfType(entity)
15         List<Component> componentsDtos = componentMapper
16             .createDTOs(componentsOfThisType);
17         dto.setComponents(componentsDtos);
18         dto.setContainer(entity.isContainer());
19         dto.setAbstract(entity.isAbstract());
20         dto.setConsumable(entity.isConsumable());
21         return dto;
22     }
23 }
```

Ausschnitt 5.9: Mapper Klasse für Bauteiltypen

5.2.3.3 Polymorphismus bei der Deserialisierung

Manche DTOs haben als Attribute abstrakte Klassen, von denen unterschiedlichste Implementierungen verwendet werden können. Beim Serialisieren stellt dies kein Problem dar, da ja bekannt ist um welche konkrete Klasse es sich bei diesen Objekten handelt. Bei

der Deserialisierung ist es für den JSON Mapper nicht möglich, die zu verwendende Klasse automatisch zu erkennen. Um dem Deserialisierer mitzuteilen, welche Implementierung verwendet werden soll, muss der Name dieser Klasse im JSON String inkludiert werden. Dies kann mit der Annotation *@JsonTypeInfo* erreicht werden. Zusätzlich dazu gibt es noch die Annotation *@JsonSubTypes*, mit der eine Liste von zugelassenen Implementierungen der abstrakten Klasse definiert werden kann. Ausschnitt 5.10 zeigt die Anwendung dieser beiden Annotation, sowie die beiden konkreten Parameter Klassen *BooleanComponentParameter* und *ListComponentParameter*.

```
1  @JsonTypeInfo(use = JsonTypeInfo.Id.CLASS,
2      include = JsonTypeInfo.As.PROPERTY,
3      property = "className")
4  @JsonSubTypes({
5      @Type(value = BooleanComponentParameter.class),
6      @Type(value = TextComponentParameter.class),
7      @Type(value = ListComponentParameter.class),
8      @Type(value = IntegerComponentParameter.class),
9      @Type(value = DoubleComponentParameter.class)
10 })
11 public abstract class ComponentParameter {
12     private String name;
13 }
14
15 public class BooleanComponentParameter extends
       ComponentParameter {
16     private boolean value;
17
18     // ... Getters and Setters
19
20 }
21
22 public class ListComponentParameter extends
       ComponentParameter {
23     private Set<String> listValues;
24     private String value;
25
26     // ... Getters and Setters
27
28 }
```

Ausschnitt 5.10: Abstraktes DTO

Die dadurch erzeugten JSON Strings für einen *BooleanComponentParameter* und einen *ListComponentParameter* sind in Ausschnitt 5.11 zu sehen.

```
1  {
2      "className":"vai.eqx2.webservice.dtos.parameters.
         BooleanComponentParameter",
3       "name":"Rebricking done",
4       "value":true
5  }
6  {
7      "className":"vai.eqx2.webservice.dtos.parameters.
         ListComponentParameter",
8      "name":"Mold Type",
9      "value":"Straight",
10     "listValues":["unknown","Straight", "Diaface"]
11 }
```

Ausschnitt 5.11: JSON Strings mit *className* Attribut

5.2.3.4 Benutzerdefinierte Mapper

Falls eine direkte Umwandlung eines Objektes in JSON nicht ausreicht, können auch benutzerdefinierte Serialisierer und Deserialisierer implementiert werden. Dafür gibt es zwei abstrakte Klassen, *JsonSerializer* und *JsonDeserializer*, in denen eine Methode zum Erzeugen bzw. Parsen des JSON Strings überschrieben werden muss. In dieser Arbeit wurde ein solcher benutzerdefinierter Serialisierer und Deserialisierer für die automatische Einheitenumschaltung implementiert. Eine Anleitung zur Implementierung benutzerdefinierter Serialisierer/Deserialisierer sowie weitere Details zur Einheitenumschaltung sind in Kapitel 5.2.9 nachzulesen.

5.2.4 Service Klassen

Sämtliche Funktionalität wurde in Service Klassen implementiert. Diese können durch Dependency Injection (siehe Kapitel 2.13.1) von jeder anderen Klasse aus verwendet werden. Dadurch wird die Wiederverwendung bestehenden Codes vereinfacht. In dieser Arbeit wurden verschiedenste Services für gewisse Funktionalitäten implementiert, die in diesem Kapitel näher beschrieben werden.

5.2.4.1 Authentication Service

Das Authentication Service stellt Funktionalität zu Verfügung, um zu überprüfen, ob eine Kombination von Benutzername und Passwort gültig ist. Dafür wird eine Verbindung zu einem LDAP Server (siehe Kapitel 2.16) aufgebaut, der die Benutzer für die Applikation speichert. Als Authentifizierungsmethode wird HTTP Basic Authentication verwendet. Bei dieser Methode werden Benutzername und Passwort als HTTP Header bei jedem HTTP Request mitgeschickt. Mithilfe des *BasicAuthFilter* (siehe Kapitel 5.2.5.4)

werden diese Werte ausgelesen und dann vom *AuthenticationService* überprüft. Falls die Kombination von Benutzername und Passwort gültig ist, wird der HTTP Request an die Controllermethode weitergeleitet (Kapitel 5.2.2), ansonsten wird der HTTP Request abgebrochen und dem Client eine Fehlermeldung, dass die Anmeldedaten ungültig sind, zurückgegeben.

5.2.4.2 Component Service

Das Component Service enthält Methoden zum Verwalten von Bauteilen. Neben den CRUD Operationen (Create, Read, Update und Delete) enthält es auch noch zusätzliche Operationen wie das Suchen eines Bauteils anhand des auf einem NFC-Tag gespeicherten Wertes (siehe Kapitel 2.6). Außerdem enthält das Service auch Operationen um die Dokumente eines einzelnen Bauteils zu erfragen oder einem Bauteil ein neues Dokument zuzuordnen.

5.2.4.3 ComponentType Service

Das ComponentType Service enthält Methoden zum Verwalten von Bauteiltypen. Da das Erzeugen, Modifizieren und Löschen von Bauteiltypen nur von der Destkopversion aus möglich sein soll und daher keine Anforderung an die mobile Applikation ist können Bauteiltypen nur angezeigt werden. Dafür enthält das ComponentType Service eine Methode, welche einen Bauteiltypen anhand der Id zurückgibt. Außerdem stellt das Service eine weitere Methode zur Verfügung, mit der alle Zähler (siehe Kapitel 4.2) aller Bauteile dieses Bauteiltypen zurückgesetzt werden können.

5.2.4.4 Document Service

Das Document Service stellt Funktionalität zur Verwaltung von Dokumenten zur Verfügung. Es gibt Methoden um die Metadaten von Dokumenten zu verwalten. Außerdem sind auch Operationen zum Hochladen eines neuen Dokumentes sowie zum Löschen von existierenden Dokumenten vorhanden.

5.2.4.5 Installed Service

Das Zusammensetzen der Baumansicht aller installierten Bauteile an der Stranggießanlage (siehe Kapitel 3.4) erfolgt mithilfe des Installed Service. Dieses enthält Methoden, um rekursiv die Bauteile einer Stranggießanlage zu durchlaufen und ein DTO Modell dieses Baums zu erzeugen. Sobald dieses DTO Modell einmal erzeugt wurde, wird es zwischengespeichert und nicht mehr neu berechnet. Damit Änderungen an der Stranggießanlage trotzdem in das Modell übernommen werden, benachrichtigt das Notification Service (siehe Kapitel 5.2.4.7) das Installed Service, sobald sich die Baumstrutkur verändert hat und das Installed Service berechnet das DTO Modell neu.

5.2.4.6 Inventory Service

Das Inventar ist eine Liste aller Bauteiltypen, von denen jeder eine Liste von Bauteilen gespeichert hat. Die Zusammensetzung dieser Liste wird vom Inventory Service durchgeführt. Ähnlich wie das Installed Service speichert auch das Inventory Service die fertig berechnete Liste und berechnet diese nur neu, wenn vom Notification Service die entsprechende Benachrichtigung dafür kommt.

5.2.4.7 Notification Service

Die von Primetals zur Verfügung gestellte Bibliothek zur Kommunikation mit dem EQX Server enthält auch eine Schnittstelle, um Callbacks für bestimmte Ereignisse zu registrieren. Diese Schnittstelle wurde dazu genutzt, bei Änderungen, die das Installed Service oder das Inventory Service betreffen diese zu benachrichtigen, ihre Caches neu zu laden. Außerdem wird gleichzeitig eine Nachricht an alle momentan verbunden Clients gesendet, dass sich das Inventar oder die Baumansicht der Stranggießanlage verändert hat und diese neu geladen werden müssen. Zur Übermittlung dieser Nachricht wird das Streaming Text Oriented Messaging Protocol (STOMP) verwendet (siehe Kapitel 2.15).

5.2.5 HTTP Request Filter

Jeder vom Webservice empfangene HTTP Request durchläuft vor der Bearbeitung durch eine Controllermethode (siehe Kapitel 5.2.2) eine Reihe von Filtern, die den HTTP Request modifizieren oder abbrechen können. Die implementierten Filter für das EQX Webservice sowie deren Aufgabe und Implementierungsdetails werden in diesem Kapitel behandelt.

5.2.5.1 Filterkette

Die Ausführung der Filter findet seriell, also alle Filter hintereinander, statt. Dazu gibt es die sogenannte *FilterChain*, oder zu deutsch Filterkette. Diese enthält Referenzen auf alle Filter und führt die Filter hintereinander aus. Da ein Filter die weitere Bearbeitung eines HTTP Requests abbrechen kann und damit auch die Ausführung der restlichen Filter unterbindet, ist die Reihenfolge, in welcher Filter durchgeführt werden von Bedeutung. Um die Reihenfolge von Benutzerdefinierten Filtern festzulegen, bietet Spring die Annotation *@Order* an, mit welcher diese Reihenfolge festgelegt werden kann. Die Reihenfolge in welcher die Filter in dieser Arbeit angeführt sind stellt gleichzeitig die Reihenfolge dar, in der die Filter im Webservice ausgeführt werden.

5.2.5.2 Implementierung benutzerdefinierter Filter

Um einen benutzerdefinierten Filter zu erzeugen, muss eine Klasse implementiert werden, die das *Filter* Interface implementiert, welches in der Java Servlet API definiert ist [Ora16]. Eigene Filter müssen die Methoden *init*, *doFilter* und *destroy* überschreiben. In der *init*

Methode können Operationen zur Initialisierung von benötigten Ressourcen und Objekten durchgeführt werden. Das gegenteil davon stellt die *destroy* Methode dar, mit welcher Ressourcen wieder freigegeben werden können. Die eigentliche Filterlogik findet in der Methode *doFilter* statt. Dieser Methode werden drei Parameter übergeben, ein Objekt zur Repräsentation des HTTP Requests, eines zur Repräsentation des HTTP Response und eine Referenz auf die Filterkette.

5.2.5.3 CORS Filter

Cross-Origin Resource Sharing (CORS) ist ein Sicherheitsmechanismus, mit dem Webservices festlegen können, von welchen Websites aus sie mittels AJAX Requests aufgerufen werden können. Eine genaue Beschreibung zu CORS ist in Kapitel 2.12 zu finden. Der CORS Filter ist dafür zuständig, dass im HTTP Response die richtigen HTTP Header enthalten sind, um jeden HTTP Request, der von der App aus gesendet wird, zuzulassen. Am wichtigsten ist der HTTP Header *Access-Control-Allow-Origin*, der die URL, also den Origin angibt, von dem aus Requests akzeptiert werden. Der Inhalt dieses Header Parameters ist entweder eine URL, ein Asterisk (*) oder *null*. Falls es sich um eine URL handelt, werden nur AJAX Requests, die von einer Website mit der angegebenen URL stammen zugelassen. Im Falle eines Asterisk werden AJAX Requests von allen Websites aus zugelassen. Dies ist erforderlich, wenn man ein öffentliches Webservices anbietet, welches von beliebigen Websites aus aufgerufen werden kann. Das EQX Webservice wird von einer Cordova App (siehe Kapitel 2.5) aus verwendet. Cordova öffnet die *index.html* Datei der App direkt als Datei im Browser. Dadurch hat die APP auch keinen gültigen Origin und damit AJAX Requests zugelassen werden muss der *Access-Control-Allow-Origin* Header den Wert *null* enthalten. Während der Implementierung wurde die App oftmals im Browser getestet, dafür wurde ein lokaler Webserver gestartet, der die App auf dem Port *8099* auf dem lokalen Rechner zugängig macht. Der Origin während der Testphase lautet also *http://localhost:8099*. Die zulässigen Origins sind also *null* für AJAX Requests aus der Cordova App und *http://localhost:8099* während der Entwicklung der App. Leider erlaubt CORS es nicht, mehrere gültige Origins, etwa durch eine Beistrich getrennte Liste anzugeben. Daher überprüft der CORS Filter, von welchem Origin aus der HTTP Request kommt. Falls dieser einer der beiden erlaubten Origins ist, wird der entsprechende Wert im HTTP Response gesetzt. Diese Überprüfung ist in Ausschnitt 5.12 in Zeile 15 zu sehen. Mit dem Befehl in Zeile 19 wird der Filterkette mitgeteilt, das der nächste Filter durchgeführt werden soll. Wenn diese Methode nicht aufgerufen wird, wird damit die Bearbeitung des HTTP Requests unterbrochen. Mit der *@Order* Annotation in Zeile 2 wird festgelegt, dass der CORS Filter immer vor allen anderen Filtern ausgeführt werden muss. Dies ist notwendig, da andere Filter die Bearbeitung des HTTP Requests unterbrechen können und wenn dies vor der Ausführung des *CORSFilters* geschieht, wird der HTTP Request vom Browser blockiert.

```
1  @Component
2  @Order(value=1)
3  public class CORSFilter implements Filter {
4
5      private static final List<String> allowedOrigins =
          Arrays.asList("null", "http://localhost:8099");
6
7      public void doFilter(ServletRequest req,
          ServletResponse res, FilterChain chain) throws
          IOException, ServletException {
8          HttpServletResponse response = (HttpServletResponse
              ) res;
9          HttpServletRequest request = (HttpServletRequest)
              req;
10
11         String origin = request.getHeader("Origin");
12         if(origin == null) {
13             origin = "null";
14         }
15         if(allowedOrigins.contains(origin)) {
16             response.setHeader("Access-Control-Allow-Origin
                  ", origin);
17         }
18         // ... set other CORS related Headers
19         chain.doFilter(request, res);
20     }
21
22     public void init(FilterConfig filterConfig) {}
23     public void destroy() {}
24 }
```

Ausschnitt 5.12: CORS Filter

5.2.5.4 Authentication Filter

Das Webservice ist mittels HTTP Basic Authentication gesichert. Das bedeutet, dass bei jedem HTTP Request die Benutzerdaten, also Benutzername und Passwort als HTTP Header mitgesendet werden. Die Überprüfung dieser Benutzerdaten übernimmt der *AuthenticationFilter*. Dazu wird der *Authorization* HTTP Header ausgelesen und die darin enthaltenen Benutzerdaten mithilfe des *AuthenticationService* (siehe Kapitel 5.2.4.1) überprüft. Falls der Benutzername unbekannt ist oder das Passwort nicht korrekt, wird die Filterkette unterbrochen und als HTTP Status der Wert *401 Unauthorized* gesetzt. Dadurch werden alle HTTP Requests nur durchgeführt, wenn gültige Benutzerdaten mitgesendet werden. Da diese nicht verschlüsselt, sondern nur Base64 codiert übertragen

werden, ist diese Art der Authentifizierung unsicher. Daher sollte für das Webservice das Hypertext Transfer Protocol Secure (HTTPS) anstatt des herkömmlichen HTTP verwendet werden, da dadurch die komplette Übertragung verschlüsselt wird. Ausschnitt 5.13 zeigt einen Teil der *doFilter* Methode der *BasicAuthFilter* Klasse. Mit dem Befehl in Zeile 1 wird der HTTP Header *Authorization* ausgelesen und in Zeile 2 wird überprüft, ob die angegebenen Benutzerdaten korrekt sind.

```
1 String authorization = request.getHeader("Authorization");
2 if(!authenticationService.verifyUser(authorization)) {
3     response.setStatus(401);
4     response.setHeader("WWW-Authenticate", String.format("
        Basic realm=\"%s\"", request.getHeader("Host")));
5 } else {
6     chain.doFilter(request, res);
7 }
```

Ausschnitt 5.13: Auslesen des Authorization HTTP Headers und überprüfen der Benutzerdaten

5.2.5.5 Session Filter

Für die Eineitenumrechnung (siehe Kapitel 5.2.9) ist es notwendig, dass der Benutzer dem Webservice mitteilt, in welcher Einheitenkultur Zahlenwerte angegeben werden sollen. Um die zu verwendende Einheitenkultur dem Server mitzuteilen, sendet die App einen HTTP Header mit dem Namen *Unit-Conversion*. Der Wert dieses Headers wird von dem *SessionFilter* vor der Ausführung der zugeordneten Request Methode ausgelesen und in einer HTTP Session gespeichert. Dadurch kann dieser Wert aus der Session herausgelesen werden und alle Zahlenwerte in die richtige Einheitenkultur umgerechnet werden. Ausschnitt 5.14 zeigt die Klasse *SessionFilter*, die den HTTP-Header ausliest und den Wert in der aktuellen HTTP Session speichert.

```
1  @Component
2  @Order(value=3)
3  public class SessionFilter implements Filter {
4
5      @Autowired
6      private HttpSession session;
7
8      @Override
9      public void init(FilterConfig filterConfig) throws
           ServletException {
10     }
11
12     @Override
13     public void doFilter(ServletRequest request,
           ServletResponse response, FilterChain chain) throws
           IOException, ServletException {
14         if(request instanceof HttpServletRequest) {
15             String units = ((HttpServletRequest)request).
                   getHeader("unit-conversion");
16         session.setAttribute("unit-conversion", units);
17         }
18         chain.doFilter(request, response);
19     }
20
21     @Override
22     public void destroy() {
23     }
24 }
```

Ausschnitt 5.14: *SessionFilter* für das Auslesen der verwendeten Einheitenkultur

5.2.6 Repositories

Alle Entitäten, also Bauteile, Bauteiltypen und Dokumente sind am EQX Server gespeichert. Für die Kommunikation mit diesem Server wird eine von Primetals zur Verfügung gestellte Java Bibliothek verwendet. Mithilfe der in dieser Bibliothek vorhandenen Methoden können alle Entitäten gelesen, gespeichert und gelöscht werden. Allerdings werden all diese Operationen nicht direkt verwendet, stattdessen existiert für jede Entität ein Repository, in dem Methoden für alle Operationen mit der jeweiligen Entität existieren. Dies hat den Vorteil, dass bestimmte Operationen, die mehrere aufeinanderfolgende Methodenaufrufe der Primetals Bibliothek erfordern zusammengefasst und einfacher wiederverwendet werden können. Außerdem kann bei einigen Operationen eine Exception auftreten, die in den Repositories behandelt werden. Das Prinzip, Datenzugriffe nicht direkt durchzuführen sondern diese in Repository Klassen durchzuführen stammt vom

Prinzip des Domain-Driven Design [Eva04]. In den meisten Fällen greifen Repositories auf eine Datenbank zu, allerdings kann die Datenschicht auch wie in diesem Fall eine andere Quelle sein. In diesem Kapitel werden die implementierten Repositores und die darin enthaltenen Methoden erläutert. Außerdem wird auch näher auf die Primetals Bibliothek eingegangen.

5.2.6.1 Eqx Provider

Um die Primetals Java Bibliothek zu verwenden, muss zuerst eine Verbindung zu dem EQX Server hergestellt werden. Dazu existiert die Klasse *Registry*, mit deren Methode *get* ein Objekt des *Manager* Interface erhalten werden kann. Dass *IEqxManager* Interface beschreibt die Schnittstelle für alle Operationen die Bauteile und Bauteiltypen betreffen. Für die Verwaltung von Dokumenten wird eine Implementierung des *IDocumentManager* Interfaces benötigt, welches ebenfalls mit *Registry.get* erhalten werden kann. Ausschnitt 5.15 zeigt den Verbindungsaufbau zum EQX Server und den Erhalt eines *IEqxManagers* und eines *IDocumentManagers*. Zeile 1 zeigt baut eine Verbindung zu einem EQX Server auf. Die Parameter *host*, *port*, *name* und *mustRun* geben die Verbindungsdaten zum EQX Server an. Diese werden in Kapitel 5.2.7 näher beschrieben.

```
1 Registry.connectTo(host, port, name, mustRun);
2 IEqxManager eqxManager = (IEqxManager)Registry.get("
     EquipmentManager2");
3 IDocumentManager documentManager = (IDocumentManager)
     Registry.get("EqxDocumentManager");
```

Ausschnitt 5.15: Verbindungsaufbau zum EQX Server

IEqxManager und *IDocumentManager* werden von den Repositories verwendet. Die Objekte dieser Interfaces werden Repository übergreifend verwendet. Daher wurde die Klasse *EqxProvider* entwickelt. Diese stellt eine Verbindung zu dem EQX Server her und stellt die Methoden *getEqx* und *getDocumentManager* zur Verfügung, um die jeweiligen Manager zu erhalten.

5.2.6.2 ComponentRepository

Das ComponentRepository enthält alle Bauteilspezifischen Operationen. Es enthält eine Methode, um einen Bauteil anhand einer Id auszulesen sowie wie Methoden zum Speichern und Löschen. Außerdem gibt es eine Methode um alle Bauteile eines Bauteiltypen zu erhalten, die für die Zusammenstellung des Inventars verwendet wird. Für die Baumansicht der Stranggießanlage gibt es eine Methode, die das Wurzelelement des Baumes zurückgibt. Methoden wie das Auslesen eines Bauteils anhand einer Id existieren in zweifacher Form, da zu dieser Id ein Bauteil existieren kann oder nicht. Um *Nullpointer Exceptions* zu vermeiden, wurde anstatt eines *null* Rückgabewertes für nicht existierende Bauteile stattdessen die in Java integrierte Klasse *Optional* verwendet. Diese stellt einen optionalen Wert dar, der entweder vorhanden ist oder nicht. Ausschnitt 5.16 zeigt die Verwendung

des *Optional* Typen als Rückgabewert für die Methode, die einen Bauteil anhand dessen Id sucht. Der in Zeile 2 zu sehende Methodenaufruf *provider.getEqx* liefert eine Referenz auf einen *IEqxManager*. *Provider* ist dabei der in Kapitel 5.2.6.1 beschriebene *EqxProvider*.

```
1 public Optional<IComponent> getById(int id) {
2     return Optional.ofNullable(provider.getEqx().
          findComponent(id));
3 }
```

Ausschnitt 5.16: Verwendung von Optional als Rückgabewerten in Repositories

Ausschnitt 5.17 zeigt die Verwendung der in Ausschnitt 5.16 definierten Methode.

```
1 Optional<IComponent> optional = componentRepository.getById
     (id);
2 if(optional.isPresent()) {
3     IComponent component = optional.get();
4 }
```

Ausschnitt 5.17: Beispielaufruf der Methode in Ausschnitt 5.16

Falls vom Webservice ein Bauteil mit einer Id angefordert wird, die nicht existiert, soll mit dem HTTP Status Code *404 Not Found* geantwortet werden. Dafür wird die in Ausschnitt 5.5 angeführte *EntityNotFoundException* ausgelöst. Um die Überprüfung ob eine Id einem Bauteil zugeordnet ist nicht an mehren Codestellen durchführen zu müssen, existiert eine zweite Methode namens *getByIdOrThrowException*, die einen Bauteil anhand einer Id zurückliefert. Diese löst eine *EntityNotFoundException* aus, falls die Id nicht existiert und unterbricht dadurch die Bearbeitung des HTTP Requests. Als HTTP Status Code wird dadurch gleichzeitig *404 Not Found* gesetzt.

5.2.6.3 ComponentTypeRepository

Alle Operationen, die Bauteiltypen betreffen werden vom *ComponentTypeRepository* durchgeführt. Ähnlich wie beim *ComponentRepository* gibt es zwei Methoden, die einen Bauteiltypen anhand einer Id suchen. Eine dieser Methoden verwendet *Optional* für den Fall dass kein Bauteiltyp gefunden wurde, die andere Methode löst eine *EntityNotFoundException* aus.

Das *ComponentTypeRepository* bietet zusätzlich noch eine Methode, um alle Bauteiltypen zu erhalten. Diese ist in Ausschnitt 5.18 zu sehen und wird bei der Zusammenstellung des Inventars verwendet.

```
1 public Collection<IComponentType> getAll() {
2     return provider.getEqx().getComponentTypes();
3 }
```

Ausschnitt 5.18: Methode um alle Bauteiltypen zu erhalten

5.2.6.4 DocumentRepository

Für die Verwaltung von Dokumenten ist das *DocumentRepository* zuständig. Darin befinden sich Methoden für das Suchen, Speichern und Löschen von Dokumenten. Außerdem existieren Methoden um Dokumente zu einem Bauteil zuzuordnen beziehungsweise diese Zuordnung wieder zu löschen. Diese Methoden sind in Ausschnitt 5.19 zu sehen.

```
1  public void addDocumentComponentReference(int componentId,
       int documentId) {
2      addDocumentReference(new EqxRefDocument(componentId,
           EqxRefDocument.COMP, documentId));
3  }
4
5  public void addDocumentReference(EqxRefDocument reference)
       {
6      try {
7          provider.getDocumentManager().putReference(
               reference);
8      } catch (DocumentManagerException e) {
9          log.error("Could not add document reference", e);
10     }
11 }
12
13 public void deleteDocumentReference(EqxRefDocument
       reference) {
14     try {
15         provider.getDocumentManager().deleteReference(
               reference);
16     } catch (DocumentManagerException e) {
17         log.error("Could not delete document reference", e)
               ;
18     }
19 }
```

Ausschnitt 5.19: Zuordnung zwischen Dokumenten und Bauteilen

5.2.7 Konfigurationsdateien

Einige Einstellungen, wie die IP-Adresse des Rechners auf dem der EQX Application Server läuft können durch Konfigurationsdateien angepasst werden. Konfigurationsdateien sind Dateien, in denen Attribute und Attributwerte definiert werden. Spring verwendet dafür die *YAML* Auszeichnungssprache [Eva09] um Wertepaare zu definieren. Ausschnitt 5.20 zeigt die Datei *eqx.yml*, in welcher Einstellungen in der *YAML* Auszeichnungssprache definiert sind. Die Zeilen 3 - 6 enthalten Einstellungen, die für den Verbindungsaufbau zu einem EQX Server notwendig sind. Der Schlüssel um auf die Werte dieser Einstellungen zuzugreifen ist eine Zusammensetzung aller übergeordneten Schlüssel. So enthält

der Schlüssel *eqx.server.port* den Wert *14970*. Die IP-Adresse des Rechners auf dem der EQX Server läuft ist je nach Umgebung unterschiedlich. Während der Test und Implementierungsphase auf dem lokalen Rechner handelt es sich um eine lokale IP-Adresse. Gleichzeitig wird das Service auch auf einem von der HTL-Perg zur Verfügung gestelltem Schulserver ausgeführt, welcher eine unterschiedliche IP-Adresse besitzt. Um dieses Problem zu lösen gibt es Einstellungsprofile. Über einen Parameter beim Start der Applikation kann konfiguriert werden, welches Profil geladen wird. Wenn als Profilname der Wert *development* angegeben wird, beträgt die in *eqx.server.host* spezifizierte IP-Adresse *172.91.92.123*, ansonsten *10.114.57.48*.

```
1  eqx:
2    server:
3      host: 10.114.57.48
4      port: 14970
5      name: EqxManagerRemoteAccessDemo
6      mustRun: true
7
8  ---
9  spring:
10   profiles: development
11 eqx.server.host: 172.91.92.123
```

Ausschnitt 5.20: Ausschnitt aus der YAML Konfigurationsdatei *eqx.yml*

Um diese Einstellungen im Java Programm zu verwenden, bietet Spring die Annotation *@ConfigurationProperties*. Ausschnitt 5.21 zeigt die Klasse *EqxProperties*. Beim Start der Applikation wird von dieser Klasse ein Objekt erzeugt und die Werte dieses Objektes mit den ausgelesenen Werten aus der Konfigurationsdatei befüllt. Mittels Dependency Injection (siehe Kapitel 2.13.1) kann dann eine Referenz auf das Objekt der Klasse *EqxProperties* erhalten werden, wie in Ausschnitt 5.22 zu sehen ist.

```
1  @Configuration
2  @EnableConfigurationProperties
3  @ConfigurationProperties(
4      locations = "classpath:eqx.yml", prefix = "eqx.server")
5  public class EqxProperties {
6      private String host;
7      private int port;
8      private String name;
9      private boolean mustRun;
10
11     // Getters and Setters ...
12
13 }
```

Ausschnitt 5.21: *EqxProperties* Klasse zum Laden der Serverkonfiguration

```
1  public class EqxProvider {
2      @Autowired
3      private EqxProperties properties;
4      public void initializeEqx() {
5          String host = properties.getHost();
6          int port = properties.getPort();
7          String name = properties.getName();
8          boolean mustRun = properties.isMustRun();
9      }
10 }
```

Ausschnitt 5.22: Verwendung der Konfigurationsdatei im Code

5.2.8 SockJS Server

In dieser Arbeit wurde SockJS (siehe Kapitel 2.14) verwendet, um jedem Benutzer der App über Änderungen der anderen Benutzer zu informieren. Das Spring Modul *spring-messaging* [PS16a] enthält Klassen und Methoden für die Implementierung eines SockJS Servers. Ausschnitt 5.23 zeigt die notwendige Konfiguration, um einen SockJS Endpunkt zu implementieren. Mit dem Befehl in Zeile 7 wird die URI "/api/stomp" für den Endpunkt festgelegt. Dieser gibt die URL an, mit der sich Clients mit dem Server verbinden können. Wichtig dafür ist, dass auch CORS (siehe Kapitel 2.12) mit der Methode *setAllowedOrigins* erlaubt wird. Der Befehl in Zeile 12 registriert einen Message Broker (siehe Kapitel 2.15.1), bei dem sich Clients anmelden können um daraufhin alle vom Server gesendeten Nachrichten zu erhalten.

```
1  @Configuration
2  @EnableWebSocketMessageBroker
3  public class WebSocketConfiguration extends
       AbstractWebSocketMessageBrokerConfigurer {
4      @Override
5      public void registerStompEndpoints(
           StompEndpointRegistry registry) {
6          registry.addEndpoint("/api/stomp").
               setAllowedOrigins("*").withSockJS();
7      }
8      @Override
9      public void configureMessageBroker(
           MessageBrokerRegistry registry) {
10         registry.enableSimpleBroker("/notifications");
11         registry.setApplicationDestinationPrefixes("/api");
12     }
13 }
```

Ausschnitt 5.23: SockJS Server Konfiguration

Um eine Nachricht an alle verbundenen Clients zu schicken existiert die Klasse *SimpleMessagingTemplate*. Eine Referenz auf ein Objekt dieser Klasse kann durch Dependency Injection (siehe Kapitel 2.13.1) erhalten werden. Ausschnitt 5.24 zeigt die Klasse *NotificationService*, die Methoden für das Senden von Nachrichten an die Clients enthält. Die Befehle in den Zeilen 8 und 12 schicken eine Nachricht an alle Clients, die die jeweilige Zieladresse abonniert haben. Der notwendige Code am Client für den Erhalt dieser Nachrichten ist in Kapitel 5.3.6 zu sehen.

```
1  @Service
2  public class NotificationService {
3
4      @Autowired
5      private SimpMessagingTemplate template;
6
7      public void sendInventoryChanged(String message) {
8          template.convertAndSend("/notifications/inventory",
               message);
9      }
10
11     public void sendInstalledChanged(String message) {
12         template.convertAndSend("/notifications/installed",
               message);
13     }
14
15 }
```

Ausschnitt 5.24: Senden einer Nachricht vom Server zu allen verbundenen Clients

Die Methoden des *NotificationService* werden aufgerufen, wenn durch die EQX Bibliothek eine Callbackmethode ausgelöst wird, dass sich die Daten am Server geändert haben. Dadurch werden auch Änderungen sofort in der App angezeigt, wenn diese von der Desktop Applikation heraus durchgeführt wurden.

5.2.9 Einheitenumrechnung

Alle im Programm vorkommenden Zahlenwerte müssen in den SI-Einheiten, EU-Einheiten oder US-Einheiten verfügbar sein. Die Auswahl der verwendeten Einheitenkultur liegt beim Benutzer der Applikation, der diese in den Appeinstelungen an seine Bedürfnisse anpassen kann. Prinzipiell kann die Einheitenumrechnung sowohl in der App als auch am Server durchgeführt werden. Da der in Kapitel 5.2.3 beschriebene Jackson JSON Mapper hierfür mit benutzerdefinierten Serialisierern eine gute Möglichkeit bietet, wurde in dieser Arbeit der Serverseitige Ansatz gewählt.

5.2.9.1 Bibliothek für die Einheitenumrechnung

Die Funktionalität für das Umrechnen diverser Einheiten und die formatierte Ausgabe von Werten einer bestimmten Einheit, wurde eine von Primetals zur Verfügung gestellte Bibliothek verwendet. Die prinzipielle Verwendung dieser Bibliothek ist in Ausschnitt 5.25 in einem Unit-Test veranschaulicht.

```
1  @Test
2  public void testConvertFromSItoEU() {
3      AvailableUnitCultures culture = AvailableUnitCultures.
           EU_Culture;
4      UnitConverter converter = new UnitConverter(new File("
           Units.xml"), culture);
5
6      ConversionResult result = converter.convertFromSI(
           Double.valueOf(30), "temperature", culture);
7
8      assertEquals(-243.15, result.getVal(), doubleDelta);
9      assertEquals("-243", result.getFormattedValue());
10     assertEquals("°C", result.getUnitDescription());
11 }
```

Ausschnitt 5.25: Unit-Test für Einheitenumrechnungsbibliothek

Diese Bibliothek verwendet eine XML Datei, in welcher die Umrechnungsfaktoren definiert sind. Ausschnitt 5.26 enthält den für die Temperaturumrechnung benötigten Teil dieser XML Datei. Für die Umrechnung zuständig sind die Attribute *ToSISlope* und *ToSIAdd*. *ToSISlope* gibt den Faktor an, mit dem ein SI Wert multipliziert werden muss, um den Wert in die gewünschte Einheit umzurechnen. *ToSIAdd* enthält den Wert, der dann noch zusätzlich zum SI Wert hinzuaddiert werden muss. Alle in der Applikation verwendeten Einheiten sind in der XML Datei definiert.

```
<Units>
    <UnitCulture>SI Units</UnitCulture>
    <EditGroup>base</EditGroup>
    <UnitKey>temperature</UnitKey>
    <UnitDimension>M</UnitDimension>
    <Unit>K</Unit>
    <Format>#0</Format>
    <ToSISlope>1</ToSISlope>
    <ToSIAdd>0</ToSIAdd>
</Units>
<Units>
    <UnitCulture>EU Units</UnitCulture>
    <EditGroup>base</EditGroup>
    <UnitKey>temperature</UnitKey>
    <UnitDimension>M</UnitDimension>
    <Unit>°C</Unit>
    <Format>#0</Format>
    <ToSISlope>1</ToSISlope>
    <ToSIAdd>273.15</ToSIAdd>
</Units>
<Units>
    <UnitCulture>US Units</UnitCulture>
    <EditGroup>base</EditGroup>
    <UnitKey>temperature</UnitKey>
    <UnitDimension>M</UnitDimension>
    <Unit>°F</Unit>
    <Format>#0</Format>
    <ToSISlope>0.555555555555556</ToSISlope>
    <ToSIAdd>255.37222222222221</ToSIAdd>
</Units>
```

Ausschnitt 5.26: Ausschnitt aus der Units.xml Datei

5.2.9.2 Bestimmung der Einheitenkultur

Für das Festlegen der zu verwendenden Einheitenkultur sendet die App einen "Unit-Conversion" HTTP Header. Dieser wird vom *SessionFilter* ausgelesen und in einer HTTP Session gespeichert. Weitere Details zu diesem Vorgang können in Kapitel 5.2.5.5 nachgelesen werden.

5.2.9.3 UnitConverter

Mit der von Primetals zur Verfügung gestellten Bibliothek zur Einheitenumrechnung muss für jede Einheitenkultur ein Objekt der Klasse *UnitConverter* angelegt werden. Dafür wur-

de die Klasse *UnitConversion* entwickelt, die für die drei Einheitenkulturen SI, EU und US *UnitConverter* erstellt und in einer HashMap mit den Schlüsselwerten *SI*, *EU* und US speichert. *UnitConversion* bietet zusätzlich eine Methode, um den *UnitConverter* zu bekommen, dessen Schlüsselwert in der HTTP Session als "Unit-Conversion" gespeichert ist. Diese Methode ist in Ausschnitt 5.27 zu sehen. In Zeile 9 wird der aus der HTTP Session ausgelesene Schlüsselwert in der HashMap gesucht. Falls dieser Schlüsselwert allerdings nicht definiert ist, da der Sender des HTTP Requests keinen Unit-Conversion Header angegeben hat, wird stattdessen ein Standardwert verwendet, nämlich die SI Einheitenkultur.

```
1  @Component
2  public class UnitConversion {
3
4      @Autowired
5      private HttpSession session;
6
7      public UnitConverter getConverterOfCurrentSession() {
8          String culture = (String) session.getAttribute("
               unit-conversion");
9          return converters.get(culture == null ? DEFAULT :
               culture);
10     }
11 }
```

Ausschnitt 5.27: Methode die den *UnitConverter* der aktuellen HTTP Session zurückgibt

5.2.9.4 ConvertableUnit DTO

Die tatsächliche Einheitenumrechnung geschieht erst beim umwandeln der DTOs (Kapitel 5.2.3) in das JSON Format. Für Werte, die in die verschiedensten Einheiten umgerechnet werden sollen gibt es ein spezielles DTO mit dem Namen *ConvertableUnit*, dessen Klassendefinition in Ausschnitt 5.28 zu sehen ist.

```
1  public class ConvertableUnit {
2
3      private double value;
4      private String unitName;
5
6      // Getters and Setters ...
7
8  }
```

Ausschnitt 5.28: ConvertableUnit DTO

Für diese Klasse werden mit dem in Ausschnitt 5.29 zu sehenden Code benutzerdefinierte Serialisierer und Deserialisierer registriert, welche die Einheitenumrechnung durchführen.

Referenzen auf den *ConvertableUnitSerializer* und den *ConvertableUnitDeserializer* werden durch Dependency Injection (siehe Kapitel 2.13.1) erlangt. In den Zeilen 13 und 14 werden diese beiden Objekte dann für die Klasse *ConvertableUnit* registriert. Dadurch werden für das Umwandeln von Objekten der Klasse *ConvertableUnit* die benutzerdefinierten Methoden *serialize* und *deserialize* aufgerufen. Die Implementierung des *ConvertableUnitSerializer* ist in Ausschnitt 5.30 zu sehen.

```
1  public class JacksonMapperConfiguration {
2      @Autowired
3      private ConvertableUnitSerializer serializer;
4      @Autowired
5      private ConvertableUnitDeserializer deserializer;
6      public ObjectMapper jacksonObjectMapper(
           Jackson2ObjectMapperBuilder builder) {
7          ObjectMapper objectMapper = builder.build();
8          SimpleModule module = new SimpleModule();
9          module.addSerializer(ConvertableUnit.class,
               serializer);
10         module.addDeserializer(ConvertableUnit.class,
               deserializer);
11         objectMapper.registerModule(module);
12         return objectMapper;
13     }
14 }
```

Ausschnitt 5.29: Registrierung der benutzerdefinierten Serialisierer und Deserialisierer

```
1  public class ConvertableUnitSerializer extends
       JsonSerializer<ConvertableUnit> {
2      @Autowired
3      private UnitConversion converter;
4      @Override
5      public void serialize(ConvertableUnit unit,
           JsonGenerator g) {
6          g.writeStartObject();
7          UnitConverter converter = converter.
               getConverterOfCurrentSession();
8          ConversionResult result = converter.convertFromSI(
               unit.getValue(), unit.getUnitName());
9          g.writeNumberField("value", result.getValue());
10         g.writeStringField("formattedValue", result.
               getFormattedValue() + " " + result.
               getUnitDescription());
11         g.writeStringField("formattedNumberValue", result.
               getFormattedValue());
12         g.writeStringField("unitDescription", result.
               getUnitDescription());
13         g.writeStringField("unitName", unit.getUnitName());
14         g.writeEndObject();
15         }  }
```

Ausschnitt 5.30: Benutzerdefinierter Serialisierer für die Einheitenumrechnung

Das Ergebnis der Serialisierung eines *ConvertableUnit* Objektes mit dem Wert *1000* und der Einheit *length* ist in Ausschnitt 5.31 sowohl für die EU Einheitenkultur als auch für die US Einheitenkultur zu sehen.

```
{
    "value": 1000,
    "formattedValue": "1000.00 m",
    "formattedNumberValue": "1000.00",
    "unitDescription": "m",
    "unitName": "length_M"
}

{
    "value": 39370.078740157485,
    "formattedValue": "39370 in",
    "formattedNumberValue": "39370",
    "unitDescription": "in",
    "unitName": "length_M"
}
```

Ausschnitt 5.31: Generiertes JSON für EU und US Einheiten

5.3 Clientseitig

5.3.1 Angular Modulstruktur

Die im Rahmen dieser Arbeit erstellte Applikation ist nach dem von Angular empfohlenen Prinzip einer modularen Struktur aufgebaut. Dabei wird die Anwendung in einzelne Module unterteilt, wobei für jeden Anwendungsbereich ein eigenes Modul zuständig ist. Der Aufbau der Modulstruktur des Clients ist in Abbildung 5.2 zu sehen. Module können wiederum in einzelne Untermodule unterteilt sein. Dieser Aufbau erlaubt das Hinzufügen neuer Anwendungsbereiche mittels Modulen und erleichtert die Entwicklung sowie das spätere Hinzufügen neuer Funktionen. Die Struktur ist aus Basismodulen und deren Untermodulen aufgebaut. Bei den Basismodulen handelt es sich um ein Modul, in welchem die Komponenten definiert sind (siehe Kapitel 5.3.2), ein Modul für die Services (siehe Kapitel 5.3.3), ein Modul für Konfigurationen (siehe Kapitel 5.3.4) und ein Modul, in welchem die Data Transfer Objects (siehe Kapitel 5.3.5) definiert sind.

Das Servicemodul stellt Funktionalitäten zur Kommunikation mit dem Webserver und der Persistierung von Benutzereinstellungen für Komponenten bereit. Eine Komponente besteht grundsätzlich aus einem oder mehreren HTML-Templates, einer CSS-Styledatei und einer Typescript Datei, in welcher die Logik für die jeweilige Komponente in einem oder mehreren Controllern implementiert ist.

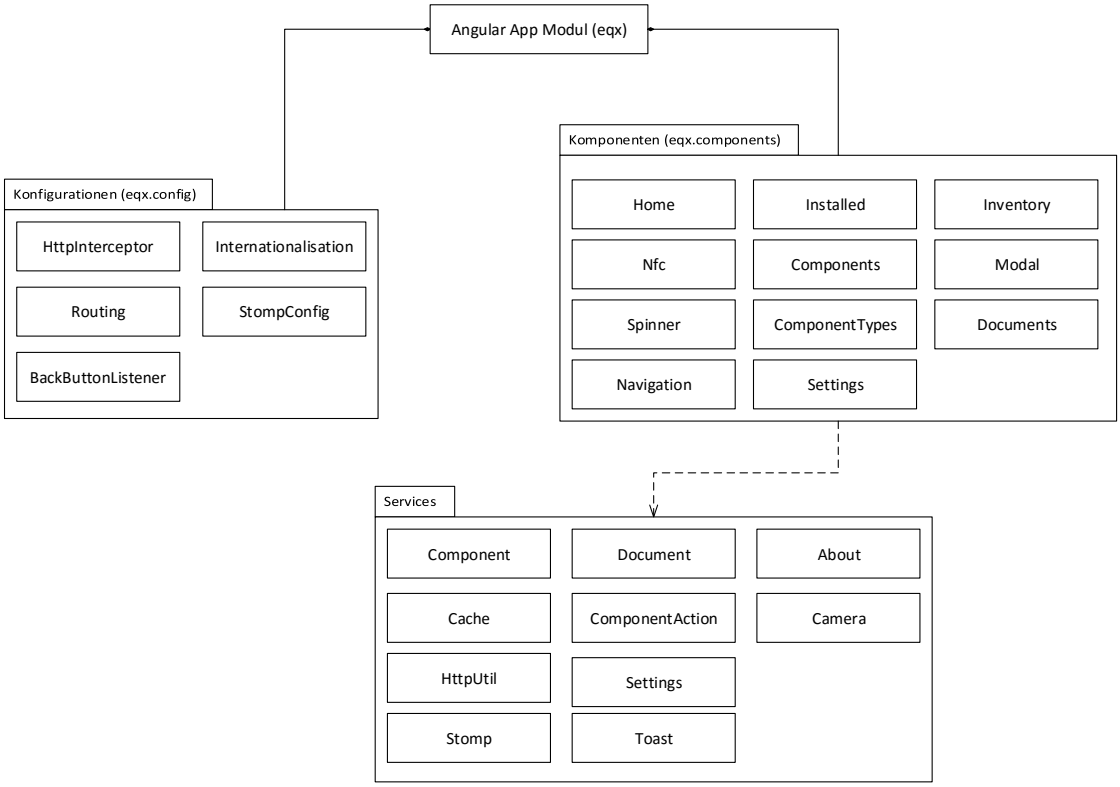


Abbildung 5.2: Client Modulstruktur

Jede Komponente ist einer bestimmten URL zugeordnet. Zuordnungen zwischen Komponenten und URLs werden auch Routen genannt. Diese Routen sind im Konfigurationsmodul definiert und enthalten jeweils den Pfad zum HTML-Template der Komponente, den Namen des Angular Controllers und die URL. Für diese Zuordnungen bietet Angular das Modul *ui.router* zur Konfiguration von Routen an. Dieses Modul stellt den Provider *$stateProvider* für Routenzuordnungen zur Verfügung. Ausschnitt 5.32 zeigt die Zuordnung der URL */documents* zum Controller *DocumentsCtrl* und dem HTML-Template *components/documents/documents.html* im JSON Format. Ausschnitt 2.11 zeigt ein Beispiel für so einen Controller. Ausschnitt 5.33 zeigt das HTML-Template für die *documents* Route.

```
1 stateProvider
2  .state('documents', {
3    url: '/documents/:documentId',
4    templateUrl: 'components/documents/documents.html',
5    controller: 'DocumentsCtrl as ctrl'
6  });
```

Ausschnitt 5.32: Erstellung einer Route

```
<div class="view-content">
  <div class="documents">
    <h1 translate="Documents"></h1>

    <div class="documents-information z2">
      <table class="table">
        <tbody>
        <tr>
          <td translate="Name"></td>
          <td class="bold" ng-bind="ctrl.document.name"></
             td>
        </tr>
        <tr>
          <td translate="Created"></td>
          <td class="bold" ng-bind="ctrl.document.
             creationTime | date:'dd.MM.yyyy HH:mm'"></td>
        </tr>
        <tr>
          <td translate="Note"></td>
          <td class="bold" ng-bind="ctrl.document.note"></
             td>
        </tr>
        </tbody>
      </table>
    </div>

    <a class="btn btn-primary" translate="Show" ng-href="{{
       ctrl.getDocumentLink()}}" target="_blank"></a>
  </div>
</div>
```

Ausschnitt 5.33: HTML Template documents.html

5.3.2 Komponenten

Die *About* Komponente dient zur Anzeige der Versionsnummern des Webservices und der mobilen Applikation. Die Komponente *Components* ist für die Anzeige und Modifizierung von einzelnen Bauteilen zuständig. Die *Documents* Komponente ermöglicht die Anzeige von einzelnen Dokumenten und Informationen. Die *Home* Komponente stellt das Hauptmenü dar und ermöglicht es dem Benutzer zur Inventar-, Nfc-, und Installationsansicht zu navigieren. Sie erscheint beim Start der Applikation. Die *Installed* Komponente zeigt den Installationsbaum an und ermöglicht es dem Benutzer, Bauteile zu installieren und deinstalliern. Die *Inventory* Komponente stellt alle vorhandenen Bauteile nach

ihren Bauteiltypen gruppiert dar. Die Komponente *Modal* besteht aus sieben Untermodulen: *Camera, Component, Confirm, Info, Install, Menu* und *Nfc*. Diese Modale stellen Templates für Kameraufnahmen, Bestätigungsabfragen, Installations- und Kontextmenüs als Services bereit. *Navigation* verwaltet die Navigationsleiste und deren Ereignisse und Menüeinträge. Die *Nfc* Komponente bietet Funktionen im Umgang mit Near Field Communication (NFC) an. Hierbei wird unter anderem die Erstellung und Modifizierung von Bauteilen mittels NFC unterstützt. Die Benutzereinstellungen verwaltet die Komponente *Settings*. Diese bietet eine Oberfläche sowohl zur Eingabe der URL des Webservices, Benutzername und Passwort zur Authentifizierung, als auch Sprache und Einheitenkultur an. Die URL des Servers muss angegeben werden, da die Applikation in mehreren Unternehmen verwendet wird und jedes dieser Unternehmen seinen eigenen Webservice mit Produktionsdaten betreibt. Da die Applikation für alle Unternehmen die selbe ist, müssen die Verbindungsdaten zu diesem Webserivce konfigurierbar sein.

5.3.3 Services

Das *CameraService* ist für den Server Upload von Bildern, die mit der Kamera aufgenommen wurden, zuständig. *ComponentService* bietet Funktionen für die Operation mit Bauteilen. Das Service *DocumentService* wird zum Upload, Download und Löschen von Dokumenten, die einem Bauteil zugeordnet sind, verwendet. *SettingsService* sorgt für die Persistierung von Einstellungen, welche in der Einstellungsansicht vom Benutzer angepasst werden. *ToastService* stellt Funktionen zur Veranschaulichung von Servernachrichten wie das erfolgreiche Speichern einer Komponente oder das fehlerhafte Laden des Inventars dar. Das *StompService* übernimmt die Websocket Kommunikation und wird in Kapitel 5.3.6 näher erläutert

5.3.4 Konfigurationen

Die *HttpInterceptor* Konfiguration fügt jedem HTTP Request ein *Unit-Conversion* Header Feld mit der vom Benutzer eingestellten Einheit hinzu, um dem Server zu signalisieren, in welcher Einheitenkultur die Bauteilwerte geliefert werden sollen. Weiters werden die Anmeldedaten des Benutzers als Basic Authentication Header hinzugefügt. Ausschnitt 5.34 zeigt wie die in der *HttpInterceptor* Klasse festgelegten Header *Unit-Conversion* und *Authorization* der HttpProvider-Konfiguration hinzugefügt werden. Die Werte liefert das *SettingsService*, welches die vom Benutzer eingestellten Einheiten und Anmeldedaten speichert.

```
1  module eqx.config {
2
3    import IHttpProvider = angular.IHttpProvider;
4    import IHttpRequestConfigHeaders = angular.
       IHttpRequestConfigHeaders;
5    import IRequestConfig = angular.IRequestConfig;
6    import SettingsService = eqx.services.settings.
       SettingsService;
7    export class HttpInterceptor {
8
9      constructor(private settingsService:SettingsService) {
10     }
11
12     public request = (config:any) => {
13       config.headers['Unit-Conversion'] = this.
           settingsService.units;
14       config.headers['Authorization'] = 'Basic ' + btoa(
           this.settingsService.userName + ':' + this.
           settingsService.password);
15       return config;
16     }
17   }
18   HttpInterceptor.$inject = ['SettingsService'];
19
20   export class InterceptorSetup {
21
22     constructor(private httpProvider:IHttpProvider) {
23       var interceptor = (settingsService:SettingsService)
           => {
24         return new HttpInterceptor(settingsService);
25       };
26       interceptor['$inject'] = ['SettingsService'];
27       httpProvider.interceptors.push(interceptor);
28     }
29   }
30   InterceptorSetup.$inject = ['$httpProvider'];
31   angular.module('eqx.config.httpInterceptor', ['eqx.
       services.settings'])
32     .config(InterceptorSetup);
33 }
```

Ausschnitt 5.34: HttpInterceptor

Für die Sprachumstellung wird die *i18n* (Internationalisierung) Konfiguration verwendet. Zur Zuordnung von Komponenten zu HTML-Templates und URLs ist die *Routing-*

Konfiguration zuständig. Ausschnitt 5.32 zeigt eine solche Zuordnung.

5.3.5 Data Transfer Objects

Im *Model* Modul befinden sich die für die Server Kommunikation erforderlichen Data Transfer Objects (DTOs). Diese sind Platzhalter für die vom Server im JSON Format (siehe Kapitel 5.2.3) erhaltenen Daten und garantieren Typsicherheit in Typescript, Autovervollständigung bei der Implementierung und eine Übersicht über die Attribute des Objekts. Sie stellen gleichzeitig die Models für das Model View Controller (MVC) Pattern dar. Ausschnitt 5.35 zeigt die Klassendefinition eines Bauteils. Das serverseitige Equivalent dieses DTO wird in Ausschnitt 5.6 dargestellt. Eine Übersicht über alle implementierten DTOs und eine Beschreibung zu jedem DTO ist in Kapitel 4.2 nachzulesen.

```
1 export class Component {
2
3     public id:number;
4     public name:string;
5     public installed:boolean;
6     public state:string;
7 }
```

Ausschnitt 5.35: Bauteil Klassendefinition in TypeScript

Wie auch serverseitig gibt es Basis DTO Klassen mit nur grundlegenden Attributen und detailreiche DTO Klassen mit allen Attributen.
Ausschnitt 5.36 zeigt die Subklasse *ComponentDetail*, welche alle Attribute eines Bauteils abdeckt. Das serverseitige Equivalent dieses DTO wird in Ausschnitt 5.8 dargestellt.

```
1  export class ComponentDetail extends Component {
2
3      public type:ComponentType;
4      public note:string;
5      public tagId:string;
6      public lastInstall:Date;
7      public lastUninstall:Date;
8      public installationPath:string;
9      public position:string;
10     public parentComponent:Component;
11     public counters:Counter[];
12     public parameters:Parameter[];
13     public documents:Document[];
14     public children:Component[];
15 }
```

Ausschnitt 5.36: Bauteil Klassendefinition in Typescript

5.3.6 SockJS Client

In dieser Arbeit wurde SockJS (siehe Kapitel 2.14) verwendet, um jedem Benutzer der App über Änderungen der anderen Benutzer zu informieren. Die Konfiguration und Implementierung eines SockJS Server ist in Kapitel 5.2.8 beschrieben. Das Kapitel *SockJS Client* zeigt die notwendigen Schritte um eine Verbindung zu einem SockJS Server herzustellen und um über Serverseitige Nachrichten informiert zu werden.

Für die Implementierung eines SockJS Client in JavaScript existiert bereits eine Bibliothek [VMw16b]. Als Protokoll für Nachrichten zwischen SockJS Client und Server wird das Simple Text Oriented Messaging Protocol (STOMP [STO12]) verwendet. Auch für STOMP existiert eine JavaScript Bibliothek [Mes12], die in dieser Arbeit verwendet wurde. Ausschnitt 5.37 zeigt die Verwendung dieser beiden Bibliotheken. In den Zeilen 3 - 5 ist der Verbindungsaufbau zu dem SockJS Server zu sehen. Die *webserviceURL* kommt dabei vom *SettingsService* (siehe Kapitel 5.3.3) und entspricht der vom Benutzer eingegeben URL. In den Zeilen 6 und 7 werden die beiden Callbackmethoden *inventoryCallback* und *installedCallback* registriert, die aufgerufen werden wenn vom Server aus eine Nachricht mit der jeweiligen Adresse gesendet wird. Diese Methoden lösen dann weitere Schritte aus, damit die Inventaransicht beziehungsweise die Baumansicht der Stranggießanlage aktualisiert wird.

```
1  export class StompService {
2      public openWebsocketConnection() {
3          var socket = new SockJS(webserviceUrl + "/api/stomp
               ");
4          var stompClient = Stomp.over(socket);
5          stompClient.connect({}, (frame:StompFrame) => {
6              stompClient.subscribe("/notifications/inventory
                   ",    (data) => this.inventoryCallback(data))
                   ;
7              stompClient.subscribe("/notifications/installed
                   ",    (data) => this.installedCallback(data))
                   ;
8          });
9      }
10
11     public inventoryCallback(data:any) {
12         // handle inventory changed event
13     }
14     public inventoryCallback(data:any) {
15         // handle installed tree changed event
16     }
17 }
```

Ausschnitt 5.37: Verwendung der SockJS [VMw16b] und STOMP [Mes12] Bibliotheken

5.4 Cordova-Projekt

5.4.1 Erzeugen eines Cordova-Projektes

Um aus der implementierten Webapplikation eine native Android- bzw. iOS-Applikation zu erzeugen, wird Apache Cordova (siehe Kapitel 2.5) verwendet. Zuerst muss Cordova als Command-Line Interface installiert werden, damit die Applikation über die Kommandozeile verwaltet werden kann. Für die Installation von Cordova wird der in Ausschnitt 5.38 in Zeile 1 zu sehende Befehl verwendet. Mit dem Befehl in Ausschnitt 5.38 in Zeile 2 wird das Projekt erzeugt. Dabei wird ein Verzeichnis mit der Grundstruktur der Cordova Applikation angefertigt. Die Parameter stehen in übergebener Reihenfolge für den Namen des angelegten Projektverzeichnisses, den Domänennamen und den Namen der Applikation.

```
1 > npm install -g cordova #globale Installation
2 > cordova create App vai.eqx2 EQXMobileApplication
```

Ausschnitt 5.38: Installation von Cordova und Projekterzeugung

5.4.2 Aufbau und Konfiguration

Das erzeugte App-Verzeichnis besteht ursprünglich aus den in Ausschnitt 5.39 zu sehenden Verzeichnissen und Dateien. Projektspezifische Verzeichnisse und Dateien werden später hinzugefgt. Die Konfigurationsdatei config.xml deckt globale Projekteinstellungen ab. In Kapitel 5.4.2.1 sind die wichtigsten Positionen in XML-Notation angeführt.

```
1 hooks
2 platforms
3 plugins
4 www
5 config.xml
```

Ausschnitt 5.39: Erzeugte verzeichnisstruktur

5.4.2.1 Teile der Konfigurationsdatei config.xml

In der Konfigurationsdatei wird der Name der Cordova Applikation im Element name angegeben. Die Position „content“ definiert die Indexdatei im www-Verzeichnis, welche beim Start der Applikation angezeigt wird. Das Whitelist Plugin ist unerlässlich für den Zugriff auf externe Applikationen und Ressourcen und daher bereits bei der Erzeugung des Projektes in der Konfigurationsdatei vorgemerkt. Durch die Angabe der darauf folgenden Positionen ist die Applikation berechtigt, auf externe Resourcen zugreifen zu können. Die Plattformspezifischen Konfigurationen erlaubt iOS den Zugriff auf den App Store und Android den Zugriff auf den Google Play Store. Zudem wird das Android API Level definiert.

```
<name>EQXMobileApplication</name>
<content src="index.html" />
<plugin name="cordova-plugin-whitelist" version="1" />
<access origin="*" />
<allow-intent href="http://*/*" />
<allow-intent href="https://*/*" />
<!--Plattformspezifische Konfigurationen-->
<platform name="android">
        <allow-intent href="market:*" />
        <preference name="android-minSdkVersion"
                value="17"/>
        <preference name="android-targetSdkVersion"
                value="22"/>
</platform>
<platform name="ios">
        <allow-intent href="itms:*" />
        <allow-intent href="itms-apps:*" />
</platform>
```

Ausschnitt 5.40: Teile der Konfigurationsdatei confix.xml

Im *hooks*-Verzeichnis werden Trigger gespeichert. Vor der Durchführung einzelner Befehle kann eine Prozedur in JavaScript oder Batch ausgeführt werden. Diese Funktionen werden im Rahmen dieser Arbeit nicht verwendet. Im *platforms*-Verzeichnis entsteht für jede Plattform ein Verzeichnis in dem sich plattformspezifische Konfigurationsdateien und native Komponenten befinden. Zudem werden dort Quellcodedateien zur Erstellung einer Webview gespeichert (siehe Kapitel 2.5.2). Im *plugins*-Verzeichnis sind nativ implementierte Plugins gespeichert (siehe Kapitel 2.5.2). Im *www*-Verzeichnis befindet sich die Webapplikation. Die von Cordova angefertigte Webview zeigt diese zur Laufzeit an.

Im Ausschnitt 5.41 sind weitere Komponenten zu sehen. Diese sind vom Cordova-Projekt unabhängig. Alle mit dem Package-Manager Bower (siehe Kapitel 2.9.1) installierten Bibliotheken werden im *bower_components*-Verzeichnis gespeichert. Die Implementierung der Webapplikation erfolgt im *src*-Verzeichnis. Mit dem Taskrunner *gulp* (siehe Kapitel 2.9.2) wird die Webapplikation optimiert und in das *www*-Verzeichnis übertragen. Die notwendigen Bibliotheken für *gulp* befinden sich im *node_modules*-Verzeichnis. Die verwendeten Tasks sind in der Konfigurationsdatei gulpfile.js definiert.

```
bower_components
node_modules
src
gulpfile.js
```

Ausschnitt 5.41: Wichtige projektspezifische Module

5.4.3 Plattformen

Eine Plattform kann mit dem Befehl in Ausschnitt 5.42 in Zeile 1 hinzugefügt werden. Der Quellcode wird dadurch nicht beeinflusst. Hinzugefügt wird nur die Plattform Android. Cordova erzeugt dafür ein eigenständiges Androidprojekt. Dieses Projekt kann in nativen Entwicklungsumgebungen verändert und gestartet werden. Eine gepackte und installierbare Applikation (*.apk) entsteht erst mit dem Kompilierbefehl in Ausschnitt 5.44. Die iOS-Plattform kann unter Windows nicht hinzugefügt werden. Die iOS-Applikation wird unter MacOS erzeugt. Dazu muss das *App*-Verzeichnis kopiert werden und die iOS-Plattform mit dem Befehl in Ausschnitt 5.42 in Zeile 2 unter MacOS hinzugefügt werden. Die gesamte Implementierung der Webapplikation findet dennoch unter Windows statt.

```
1 App> cordova platform add android      #Unter Windows
2 App$ cordova platform add ios          #Unter MacOS
```

Ausschnitt 5.42: Cordova Befehle für das Hinzufügen der Plattformen

5.4.4 Verwendete Plugins

Für spezielle native Zugriffe sind Plugins (siehe Kapitel 2.5.2) notwendig. Diese können mit dem Befehl in Ausschnitt 5.43 in Zeile 1 hinzugefügt werden. Jedes Cordova-Projekt benötigt das Whitelist-Plugin für externe Zugriffe. Dieses ist standardmäßig in der Konfigurationsdatei config.xml angeführt. Bei der Kompilierung werden alle in der Konfigurationsdatei enthaltenen Plugins heruntergeladen, falls sie noch nicht vorhanden sind. Für den Zugriff auf die native Kamera wird ein Plugin benötigt. Um NFC Tags (siehe Kapitel 2.6.2) lesen zu können, wird das selbst implementierte NFC-Plugin (siehe Kapitel 5.5) verwendet. Diese Funktion ist nur auf NFC-fähigen Android-Geräten verfügbar. IOS-Geräte bieten die NFC-Funktion nicht an. Mit den Befehlen in Ausschnitt 5.43 in Zeile 1 und 3 werden die erforderlichen Plugins heruntergeladen. Eine Liste aller im Projekt existierenden Plugins wird mit dem Befehl in Ausschnitt 5.43 in Zeile 6 angezeigt.

```
1 App> cordova plugin add https://github.com/apache/cordova-
     plugin-camera
2 App> cordova plugin add https://github.com/davidHaunschmied
     /cordova-nfc-plugin
3
4 App> cordova plugins
5 cordova-plugin-whitelist 1.0.0 "Whitelist"
6 org.apache.cordova.camera 0.3.6 "Camera"
7 nfc.plugin.NfcVPlugin 0.1.0 "NfcVPlugin"
```

Ausschnitt 5.43: Cordova Befehle für das Hinzufügen der Plugins

5.4.5 Kompilierung

Cordova erzeugt mit dem Befehl in Ausschnitt 5.44 Cordova für jede im *platforms*-Verzeichnis existierende Plattform ein Applikationspaket. Unter Windows existiert in dem Projekt nur die Android-Plattform, daher wird nur ein Paket für Android (*.apk) angefertigt. Das iOS-Applikationspaket (*.ipa) wird unter MacOS angefertigt. Die Applikationspakete werden unter *outputs* in den verschiedenen Plattformverzeichnissen gespeichert. Neben Konfigurationsdateien und Ressourcen besteht ein Applikationspaket aus einer Webview in nativer Sprache (Android: Java, iOS: Objective C) und der kopierten Webapplikation.

```
1 App> cordova build
```

Ausschnitt 5.44: Cordova Befehl zur Erzeugung eines Applikationspaketes

5.4.6 Ausführen der Applikation

Die Webapplikation kann im Browser ausgeführt und getestet werden. Dazu stellt der Taskrunner *gulp* einen Webserver zur Verfügung. Native Funktionen wie NFC oder die Kamera können im Browser nicht getestet werden. Um die Cordova Applikation zu testen wird ein Android Gerät benötigt. Das Applikationspaket kann auf dem Gerät installiert und ausgeführt werden. Mit dem Befehl in Ausschnitt 5.45 in Zeile 1 wird die Applikation angefertigt, auf dem Gerät installiert und ausgeführt. Dazu muss das Gerät mit einem USB-Kabel an den PC angeschlossen sein. In Google Chrome kann unter dem Link *chrome://inspect/* die laufende Applikation auf Fehler untersucht werden. Die iOS-Applikation kann unter MacOS getestet werden. [Del13] Dazu wird der Apple Safari Browser verwendet. Mit der Entwicklungsumgebung XCode kann die iOS-Applikation auf dem verbundenen Apple Gerät installiert und ausgeführt werden.

```
1 App> cordova run android      #Unter Windows
```

Ausschnitt 5.45: Cordova Befehl zur Ausführung auf einem Androidgerät

5.5 NFC Plugin

5.5.1 Ausgangssituation

Das Plugin soll ermöglichen, Bauteile über NFC (siehe Kapitel 2.6.1) identifizieren zu können. Dazu werden diese mit NFC-Tags ausgestattet (siehe Kapitel 2.6.2).

5.5.1.1 Existierende Plugins

Es existieren bereits mehrere Plugins für die Bereitstellung der NFC-Funktion für Cordova unter Android. Das Plugin *phonegap-nfc* [Sol15] wurde hinzugefügt und getestet. Dazu wurden die von Primetals (Verweis auf Auftraggeber) zur Verfügung gestellten NFC-Tags verwendet. Jedoch bietet das Plugin keine Lese- und Schreibfunktionen für diese NFC-Tags. Der Grund dafür liegt darin, dass sich NFC-Tags im Typ beziehungsweise in den angebotenen Technologien (siehe Kapitel NFC-Technologien) unterscheiden. Beide Plugins verwenden die Ndef-Technologie (NFC Data Exchange Format) von Android zum Lesen und Beschreiben von NFC-Tags. Die eingesetzten NFC-Tags bieten die Ndef-Technologie jedoch nicht an, daher können diese existierenden Plugins nicht verwendet werden. Als Lösung wurde gemeinsam mit dem Auftraggeber Primetals entschieden, ein eigenes NFC-Plugin für Android zu implementieren. Dieses soll einen Wert auf den verwendeten Tags speichern und lesen können.

5.5.1.2 Verwendete Tags

Die Bauteile einer Stranggießanlage (siehe Kapitel 2.1) sind mit einem NFC-Tag versehen. Diese Tags speichern abhängig vom Bauteil eine ID, welche ausgelesen beziehungsweise überschrieben werden kann. Die Technologie und der Ablauf des Datenaustausches der verwendeten Tags ist abhängig von deren Typ. Die eingesetzten Tags haben den NFC-Tag Typ ISO 15693 (siehe Kapitel 2.6.2).

5.5.2 Entwicklung des NFC Plugins

5.5.2.1 Aufbau des Cordova Plugins

In Abschnitt 5.46 sind die notwendigen Verzeichnisse und Dateien zu sehen. Diese befinden sich im davor angelegten *cordova-nfc-plugin*-Wurzelverzeichnis. Die Konfigurationen eines Plugins befinden sich in der Konfigurationsdatei plugin.xml. Im Ausschnitt 5.47 sind die wichtigsten Positionen in XML-Notation zu sehen. Die Schnittstelle zwischen der Webapplikation und den nativ implementierten Quellcodedateien ist die JavaScript-Datei *nfcvplugin.js*. Diese Datei befindet sich im www-Verzeichnis. Im src-Verzeichnis sind Unterverzeichnisse für die unterschiedlichen nativen Implementation enthalten. Das Plugin wird nur für das Android Betriebssystem entwickelt, da iOS die NFC-Funktion nicht anbietet. Aus diesem Grund ist im src-Verzeichnis ein android-Verzeichnis mit in Java geschriebenen Quellcode Dateien vorhanden.

```
1 src
2 www
3 plugin.xml
4 README.md
```

Ausschnitt 5.46: Ordnerstruktur

```
1  <js-module src="www/nfcvplugin.js" name="nfc">
2    <clobbers target="nfc" />
3  </js-module>
4
5  <!-- android -->
6  <platform name="android">
7    <config-file target="res/xml/config.xml" parent="/*">
8      <feature name="NfcVPlugin">
9        <param name="android-package" value="nfc.plugin.
           NfcVPlugin"/>
10       <param name="onload" value="true" />
11     </feature>
12   </config-file>
13
14   <source-file src="src/android/NfcVPlugin.java" target-dir
       ="src/nfc/plugin" />
15   <source-file src="src/android/NfcVHandler.java" target-
       dir="src/nfc/plugin" />
16 </platform>
```

Ausschnitt 5.47: Teile der Konfigurationsdatei plugin.xml

Der in Ausschnitt 5.47 in Zeile 1 bis 3 zu sehende XML-Tag definiert den relativen Quellpfad zur JavaScript-Datei *nfcvplugin.js* und das JavaScript-Modul *nfc*. Dieses Modul wird von der Webapplikation verwendet, um über JavaScript native Funktionen des Plugins aufzurufen. In den Zeilen 6 bis 16 sind spezielle Einstellungen für die Android-Plattform zu sehen. Die Unterelemente vom *config-file*-Tag werden bei der Erzeugung der Applikation in die definierte *config.xml* übertragen. Die erste angegebene Position unter dem *feature* NfcVPlugin ist notwendig, damit das Plugin beim Erzeugen der Applikation von Cordova gefunden werden kann. Die zweite Position gibt an, dass das Plugin bereits beim Start der Applikation geladen wird und nicht erst beim ersten Aufruf. Damit wird die Performance zur Laufzeit verbessert. Die Positionen in Zeile 14 und 15 spezifizieren den Pfad zu den nativen Quellcodedateien, sowie deren Zielverzeichnis im plattformspezifischen Cordova Projekt. In dieses Verzeichnis werden die Dateien beim Hinzufügen des Plugins beziehungsweise der Plattform kopiert.

5.5.2.2 Implementierung

Die Implementierung des Cordova Plugins (siehe Kapitel 2.5.2) umfasst die JavaScript Datei *nfcvplugin.js* und die Java Dateien *NfcVPlugin.java* und *NfcVHandler.java*. In der JavaScript Datei ist das Modul *nfc* definiert. In diesem JSON-Objekt sind mehrere Funktionen deklariert. Eine Funktion zum Lesen von NFC-Tags, eine zum Beschreiben eines NFC-Tags und eine um die Verfügbarkeit von NFC am Android-Gerät zu prüfen. Zudem sind Funktionen definiert, um das Lesen beziehungsweise Beschreiben von Tags zu stoppen. Diese Funktionen haben Parameter, darunter eine *success* und eine *error* Callbackmethode. Diese werden je nach Ergebnis des Plugins aufgerufen. Jede im JSON-Objekt definierte Funktion ruft die Cordova *execute*-Methode auf. Dieser Methode werden beide Callbackmethoden, der Name der nativen Klasse *NfcVPlugin* und ein String, der die durchzuführende Aktion definiert, übergeben. Beim Beschreiben eines NFC-Tags werden zudem Argumente als JSON-Array mitgegeben. Das Plugin unterscheidet die Funktionen anhand des *action*-Parameters. Im Ausschnitt 5.48 ist ein Beispiel für den Aufruf der Cordova *execute*-Methode zu sehen. Die mitgegebene Aktion ist *startReadingNfcV*.

```
1 addNfcVListener: function (success, error) {
2   cordova.exec(success, error, "NfcVPlugin", "
       startReadingNfcV", []);
3 }
```

Ausschnitt 5.48: Ausschnitt nfcvplugin.js

Die angegebene Klasse *NfcVPlugin* muss von der existierenden Klasse *CordovaPlugin* erben und die vordefinierte *execute*-Methode überschreiben. Dadurch kann Cordova diese Methode bei einem Aufruf ausführen. In Ausschnitt 5.49 ist der Methodenkopf der im eigenen Plugin überschriebenen Methode zu sehen. Der *action*-String wird mit vordefinierten Zeichenketten verglichen und danach die jeweilige Funktion aufgerufen. Das *args* Objekt enthält optional die mitgegeben Parameter. Mit dem Objekt der Klasse *CallbackContext* können die Callbackmethoden aufgerufen werden und Daten beziehungsweise Fehlermeldungen zurückgegeben werden.

```
1 @Override
2 public boolean execute(String action, JSONArray args,
     CallbackContext callbackContext) throws JSONException { }
```

Ausschnitt 5.49: NFCTechPlugin.java *execute*-Methode

Das NFC-Plugin bietet eine Funktion an, um die Verfügbarkeit von NFC am Gerät zu prüfen. Dazu wird der *NfcAdapter* des Android-Gerätes auf *null* und *isEnabled()* überprüft. Dadurch wird erkannt, ob das Android-Gerät überhaupt NFC-fähig ist, beziehungsweise ob NFC aktiviert ist.
Wenn ein Tag gefunden wurde, wird dieses Ereignis von Android nicht an das Plugin weitergeleitet. Die Aktion *startReadingNfcVTech* aktiviert die Weiterleitung der gefundenen NfcV-Tag Ereignisse an das Plugin. Dazu wird die Methode *enableForegroundDispatch* verwendet. Bei Eintritt dieses Ereignisses wird mithilfe der NfcV-Methode *transceive* ein

bestimmter Datenblock des Tags ausgelesen. Ein Block ist ein byte-Array der Größe 4. Dieses wird in einen Integer-Wert umgewandelt und zurückgegeben. Die Methode *transceive* erlaubt gezielte Lese- und Schreibfunktion auf einzelne Bytes des Tags. Das dazu verwendete Protokoll ist in den Grundlagen (siehe Kapitel 2.6.2) beschrieben. Jeder Datenaustausch mit dem Tag geschieht über diese Funktion. Der beschreibbare Speicher ist in 4 Byte große Blöcke unterteilt. In Ausschnitt 5.50 ist der Aufruf der Funktion und die dafür notwendigen Parameter zu sehen. Mit dem Flag-Byte können zusätzliche Aktionen definiert werden. Das Kommando im zweiten übergebenen Byte bestimmt die Aktion. Das dritte Byte bestimmt den zu lesenden Block.
Die Aktion *startWritingNfcVTech* aktiviert auch die Weiterleitung der NfcV-Tag Ereignisse an das Plugin. Zudem wird ein Wert an das Plugin mitgegeben, der auf einen Tag geschrieben werden soll. Dieser Integer-Wert wird in ein 4-Byte großes Array umgewandelt und auf den gleichen Block wie beim Lesen geschrieben. In Ausschnitt 5.51 ist der Aufruf der *transceive* Methode mit den notwendigen Parametern zum Beschreiben eines einzelnen Blocks zu sehen. Der Unterschied zum Lesen ist das übergebene Kommando. Zudem wird der in 4 Byte geteilte Wert mitgegeben.

```
1  byte[] request = new byte[]{
2          (byte)0x00,     // Flag
3          (byte)0x20,     // Kommando: READ ONE BLOCK
4          (byte)block     // Block
5  };
6  response = nfcv.transceive(cmd);
7  // 1. Byte: Block locking status
8  byte[] value = new byte[]{ response[2], response[3],
      response[4], response[5] };
```

Ausschnitt 5.50: Lesen eines Blocks mithilfe der NfcV transceive-Methode

```
1  byte[] request = new byte[7];
2  request[0] = 0x00;              // Flag
3  request[1] = 0x21;              // Kommando: WRITE ONE
      BLOCK
4  request[2] = (byte) block;      // Block
5
6  request[3] = (byte) data[0];
7  request[4] = (byte) data[1];
8  request[5] = (byte) data[2];
9  request[6] = (byte) data[3];
10 response = nfcv.transceive(cmd);
```

Ausschnitt 5.51: Beschreiben eines Blocks mithilfe der NfcV transceive-Methode

Die Aktionen *stopReadingNfcV* und *stopWritingNfcV* stoppen die Weiterleitung der NFC-Ereignisse an das Plugin. Dazu wird die Methode *disableForegroundDispatch* verwendet.

5.5.2.3 Veröffentlichung auf Github

Das NFC-Plugin ist auf Github veröffentlicht. Unter dem Link *https://github.com/davidHaunschmied/cordova-nfc-plugin* kann das Plugin heruntergeladen werden. Es ist ein Open Source Projekt und kann von Entwicklern genutzt beziehungsweise weiterentwickelt werden. Die Einbindung in das Cordova Projekt ist als Open Source Projekt ebenfalls einfacher. In Ausschnitt 5.43 ist zu sehen, wie das Plugin mithilfe der Kommandozeile über die Github URL hinzugefügt wird. Bei Veränderungen des Pluginquellcodes kann auf diese Weise die neue Version in das Cordova Projekt übernommen werden.

5.5.2.4 Verwendung des Plugins in der Applikation

Die Verwendung im Client funktioniert über das im Plugin erzeugte JavaScript Objekt *nfc*. Dieses Objekt ist nur für das Betriebssystem Android definiert. Für jede andere Plattform ist das Objekt *undefined*. Am Client kann die Funktion nur verwendet werden, wenn NFC verfügbar und aktiviert ist. In Ausschnitt 5.52 ist der Funktionsaufruf zum Lesen des am Tag gespeicherten Wertes mit Übergabe der zwei Callbackmethoden zu sehen. Je nach Pluginergebnis wird eine der beiden Callbackmethoden aufgerufen.
Jedem am EQX-Server gespeicherten Bauteil kann ein Tag zugewiesen sein. Mithilfe des am Tag gespeicherten Wertes wird das jeweilige Bauteil vom Server abgerufen. Die verfügbaren NFC-Funktionen sind davon abhängig, ob der Tag einem Bauteil zugewiesen ist oder ob dieser Tag keinem oder einem gelöschten Bauteil zugewiesen ist. Die Bedienung und verfügbaren NFC-Funktionen sind im Kapitel (siehe Kapitel Bedienung NFC) zu sehen.

```
1  nfc.addNfcVListener((tagValue:any) => {
2          this.toastService.showSuccess('Tag found');
3          this.refreshComponent(tagValue);
4  },
5  (error:any) => {
6          if (error == 'NFC_DISABLED'){
7                  this.toastService.showError("NFC is
                      disabled");
8          }else if(error == 'NO_NFC'){
9                  this.toastService.showError("This device
                      doesn't support NFC!");
10         }
11         console.log(error);
12         this.state.go("home");
13 });
```

Ausschnitt 5.52: Verwendung der Lesefunktion des Plugins in TypeScript

Ein gelesener Tag kann einem Bauteil zugewiesen werden. Dazu wird der gespeicherte Wert überschrieben. In Ausschnitt 5.53 ist der Aufruf der Schreibfunktion zu sehen. Zu

den Callbackmethoden werden zwei Werte übergeben. Der zuvor ausgelesene Wert des Tags und der zu speichernde Wert. Um den neuen Wert auf den Tag zu schreiben, muss das Gerät erneut zum NFC-Tag gehalten werden. Der neu gespeicherte Wert wird über die Callbackmethode zurückgegeben. Der *alte* Wert ist zur Überprüfung notwendig. Es wird sichergestellt, dass der zuvor gelesene Tag und der zu beschreibende Tag der gleiche ist. Wenn beim Beschreiben des Tags ein anderer Tag verwendet wird, wird dieser nicht beschrieben und die *error*-Callbackmethode aufgerufen.

```
1 nfc.addNfcVWriter(this.oldValue, this.newValue, (tagValue:
     any) => {
2         this.proceed(tagValue);
3 },(error:any) => {
4         console.log(error);
5         if (error == "FALSE_TAG"){
6                 this.toastService.showError("Invalid tag!")
                     ;
7         }
8         this.cancel();
9 });
```

Ausschnitt 5.53: Verwendung der Schreibfunktion Plugins in TypeScript

Kapitel 6

Beurteilung

Durch die Implementierung der *EQX Mobile* Applikation haben wir einige wertvolle Erfahrungen im Bereich des Projektmanagement gewonnen und auch unseren Technologischen Wissensstand verbessern können. Projektmeetings fanden alle drei Wochen statt, wodurch wir die agile Entwicklungsmethode SCRUM zum ersten mal tatsächlich anwenden konnten, da bei jedem Meeting immer gewisse neue Features zur Applikation hinzukamen.

Außerdem haben wir die Erfahrung gemacht, dass es in den meisten Fällen besser ist auf etablierte Technologien zu setzen. Als die Entwicklung im September 2015 startete, versuchten wir bereits den Nachfolger von AngularJS, Angular2 zu verwenden, obwohl sich das Framework damals noch in der Beta Phase befand. Doch Aufgrund des Mangels an Codebeispielen im Internet und auch an Zusatzbibliotheken von Drittanbietern beschlossen wir nach einigen Tagen, doch das für uns bereits bekannte AngularJS zu verwenden. Allerdings haben wir durch Angular2 das erste mal TypeScript verwendet und da diese Sprache im Vergleich zu JavaScript viele Vorteile bietet, auf die wir nicht mehr verzichten wollten haben wir uns entschlossen TypeScript in Kombination mit AngularJS zu verwenden.

Eine weitere neue Erfahrung für uns war die Zusammenarbeit mit einem größerem Unternehmen und vor allem die Verwendung und Weiterentwicklung eines bereits existierenden Softwareprodukts. Wir wurden vor die Herausforderung gestellt, dieses vorhandene System bestmöglich in unsere eigene Entwicklung einzubinden, um etwaige Änderungen möglichst einfach übernehmen zu können. Außerdem war auch die Auswahl der verwendeten Technologien von größerer Bedeutung, da das Produkt zum einen von Primetals übernommen werden soll und andererseits für ein kommerzielles Produkt Lizenzbestimmungen beachtet werden mussten.

Abbildungsverzeichnis

2.1 Stranggießen [Sta16] . . . 16
2.2 Inventaransicht und Detailansicht des Bauteils *N4* vom Bauteiltyp *Narrow Plate* . . . 17
2.3 Baumansicht der installierten Bauteile und Detailansicht des Bauteils *CC2* vom Bauteiltyp *Caster* . . . 18
2.4 Cordova Architektur [Fou15a] . . . 32
2.5 Cordova Plugin Architektur [Rip14] . . . 33
2.6 Aktiver-aktiver Kommunikationsmodus [Har13] . . . 35
2.7 Aktiver-passiver Kommunikationsmodus [Har13] . . . 36
2.8 Standard Request Format [STM15] . . . 36
2.9 Lesen eines einzelnen Blocks: Anfrage [STM15] . . . 37
2.10 Lesen eines einzelnen Blocks: Antwort des Tags [STM15] . . . 37
2.11 Beschreiben eines einzelnen Blocks: Antwort des Tags [STM15] . . . 37
2.12 Beschreiben eines einzelnen Blocks: Antwort des Tags [STM15] . . . 38

3.1 Skizzierung der Arbeitsabläufe . . . 54
3.2 Hautpmenü . . . 55
3.3 Inventaransicht . . . 57
3.4 Baumansicht der Stranggießanlage . . . 59
3.5 Detailansicht eines Bauteils . . . 60
3.6 Dokumentenverwaltung eines Bauteils . . . 61
3.7 NFC Bauteilerkennung . . . 62

4.1 Architektur . . . 64
4.2 UML Diagramm der Bauteil Klassenstruktur . . . 66

5.1 Ablauf der HTTP Request Behandlung . . . 75
5.2 Client Modulstruktur . . . 103

Ausschnittsverzeichnis

2.1 TypeScript Basistypen (vgl. [Mic16]) . . . 19
2.2 TypeScript Array (vgl. [Mic16]) . . . 19
2.3 TypeScript Enum (vgl. [Mic16]) . . . 19
2.4 TypeScript Datentyp Void (vgl. [Mic16]) . . . 20
2.5 TypeScript Klasse (vgl. [Mic16]) . . . 20
2.6 Erzeugter JavaScript Code [Mic16] . . . 20
2.7 TypeScript Vererbung, Zugriffsmodifikatoren (vgl. [Mic16]) . . . 21
2.8 TypeScript Modul (vgl. [Mic16]) . . . 22
2.9 Nicht übereinstimmende Datentypen . . . 23
2.10 Angular Beispiel index.html . . . 27
2.11 Angular Beispiel app.ts . . . 28
2.12 Spinner Direktive . . . 29
2.13 Verwendung Spinner Direktive . . . 29
2.14 SCSS Syntax (vgl. [SAS16]) . . . 39
2.15 SASS Syntax (vgl. [SAS16]) . . . 39
2.16 CSS Ergebnis (vgl. [SAS16]) . . . 39
2.17 SCSS Verschachtelung (vgl. [SAS16]) . . . 40
2.18 CSS Ergebnis (vgl. [SAS16]) . . . 40
2.19 Bootstrap Grid System . . . 41
2.20 Beispiel JSON eines "window" Objektes (vgl. [JSO16b]) . . . 44
2.21 Verwendung des Spring Dependency Injection Frameworks . . . 48
2.22 SockJS Client Beispiel (vgl. [VMw16b]) . . . 49
5.1 Beispiel Routemapping . . . 76
5.2 Routemapping mit Parameter . . . 77
5.3 Request Body Parameter . . . 77
5.4 HTTP Status Code . . . 78
5.5 HTTP Status Code für Exceptions . . . 78
5.6 Bauteil Klassendefinition . . . 79
5.7 Bauteil JSON Repräsentation . . . 79
5.8 Bauteil Detailreiche Klasse . . . 80
5.9 Mapper Klasse für Bauteiltypen . . . 81
5.10 Abstraktes DTO . . . 82
5.11 JSON Strings mit *className* Attribut . . . 83
5.12 CORS Filter . . . 87

5.13 Auslesen des Authorization HTTP Headers und überprüfen der Benutzerdaten . . . 88
5.14 *SessionFilter* für das Auslesen der verwendeten Einheitenkultur . . . 89
5.15 Verbindungsaufbau zum EQX Server . . . 90
5.16 Verwendung von Optional als Rückgabewerten in Repositories . . . 91
5.17 Beispielaufruf der Methode in Ausschnitt 5.16 . . . 91
5.18 Methode um alle Bauteiltypen zu erhalten . . . 91
5.19 Zuordnung zwischen Dokumenten und Bauteilen . . . 92
5.20 Ausschnitt aus der YAML Konfigurationsdatei *eqx.yml* . . . 93
5.21 *EqxProperties* Klasse zum Laden der Serverkonfiguration . . . 93
5.22 Verwendung der Konfigurationsdatei im Code . . . 94
5.23 SockJS Server Konfiguration . . . 94
5.24 Senden einer Nachricht vom Server zu allen verbundenen Clients . . . 95
5.25 Unit-Test für Einheitenumrechnungsbibliothek . . . 96
5.26 Ausschnitt aus der Units.xml Datei . . . 97
5.27 Methode die den *UnitConverter* der aktuellen HTTP Session zurückgibt . . . 98
5.28 ConvertableUnit DTO . . . 98
5.29 Registrierung der benutzerdefinierten Serialisierer und Deserialisierer . . . 100
5.30 Benutzerdefinierter Serialisierer für die Einheitenumrechnung . . . 100
5.31 Generiertes JSON für EU und US Einheiten . . . 101
5.32 Erstellung einer Route . . . 103
5.33 HTML Template documents.html . . . 104
5.34 HttpInterceptor . . . 106
5.35 Bauteil Klassendefinition in TypeScript . . . 107
5.36 Bauteil Klassendefinition in Typescript . . . 107
5.37 Verwendung der SockJS [VMw16b] und STOMP [Mes12] Bibliotheken . . . 108
5.38 Installation von Cordova und Projekterzeugung . . . 109
5.39 Erzeugte verzeichnisstruktur . . . 109
5.40 Teile der Konfigurationsdatei confix.xml . . . 110
5.41 Wichtige projektspezifische Module . . . 110
5.42 Cordova Befehle für das Hinzufügen der Plattformen . . . 111
5.43 Cordova Befehle für das Hinzufügen der Plugins . . . 111
5.44 Cordova Befehl zur Erzeugung eines Applikationspaketes . . . 112
5.45 Cordova Befehl zur Ausführung auf einem Androidgerät . . . 112
5.46 Ordnerstruktur . . . 114
5.47 Teile der Konfigurationsdatei plugin.xml . . . 114
5.48 Ausschnitt nfcvplugin.js . . . 115
5.49 NFCTechPlugin.java *execute*-Methode . . . 115
5.50 Lesen eines Blocks mithilfe der NfcV transceive-Methode . . . 116
5.51 Beschreiben eines Blocks mithilfe der NfcV transceive-Methode . . . 116
5.52 Verwendung der Lesefunktion des Plugins in TypeScript . . . 117
5.53 Verwendung der Schreibfunktion Plugins in TypeScript . . . 118

Literaturverzeichnis

[Boo16] Bootstrap. Bootstrap, 2016. URL http://getbootstrap.com/. Zuletzt besucht: 01.04.2016.

[Bor14] Ralf Borchers. Cross-Origin Resource Sharing (CORS) über Apache HTTP, 2014. URL http://www.communardo.de/home/techblog/2014/10/09/. Zuletzt besucht: 01.04.2016.

[Bow16] Bower. Bower, 2016. URL http://bower.io/. Zuletzt besucht: 01.04.2016.

[Del13] Michael Dellanoce. Debugging iOS PhoneGap Apps with Safari's Web Inspector, 2013. URL http://phonegap-tips.com/articles/debugging-ios-phonegap-apps-with-safaris-web-inspector.html. Zuletzt besucht: 01.04.2016.

[Ede14] Johannes Eder. Gradle: Build-Automatisierung mit Android Studio, 2014. URL http://mfg.fhstp.ac.at/allgemein/gradle-build-automatisierung-mit-android-studio/. Zuletzt besucht: 01.04.2016.

[Eis99] Verein Deutscher Eisenhüttenleute. *Stahl-Fibel*. Verein Deutscher Eisenhüttenleute, 1999.

[ET15] Inc. Embarcadero Technologies. Embarcadero, 2015. URL https://www.embarcadero.com/. Zuletzt besucht: 01.04.2016.

[Eva04] Eric J. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[Eva09] Clark C. Evans. YAML Ain't Markup Language, 2009. URL http://yaml.org/. Zuletzt besucht: 01.04.2016.

[Fou15a] Apache Software Foundation. Apache Cordova Architecture, 2015. URL https://cordova.apache.org/docs/en/latest/guide/overview/index.html. Zuletzt besucht: 01.04.2016.

[Fou15b] Apache Software Foundation. Apache Cordova Homepage, 2015. URL https://cordova.apache.org/. Zuletzt besucht: 01.04.2016.

[Fou16a] Apache Software Foundation. Apache Software Foundation, 2016. URL http://www.apache.org/. Zuletzt besucht: 01.04.2016.

[Fou16b] Apache Software Foundation. ApacheCommons, 2016. URL https://commons.apache.org/. Zuletzt besucht: 01.04.2016.

[Fra16] Fractal. Gulp, 2016. URL http://gulpjs.com/. Zuletzt besucht: 01.04.2016.

[Goo15] Google. Guava, 2015. URL https://github.com/google/guava. Zuletzt besucht: 01.04.2016.

[Goo16] Google. AngularJS, 2016. URL https://angularjs.org/. Zuletzt besucht: 01.04.2016.

[Har13] Michael J. Harnisch. Near Field Communication, 2013. URL https://www.gi.de/nc/service/informatiklexikon/detailansicht/article/near-field-communication-nfc/druckversion.html. Zuletzt besucht: 01.04.2016.

[HC15] Chris Eppstein Hampton Catlin, Natalie Weizenbaum. SASS, 2015. URL http://sass-lang.com/. Zuletzt besucht: 01.04.2016.

[Hel13] Martin Helmich. RESTful Webservices: Was ist das überhaupt?, 2013. URL https://www.mittwald.de/blog/webentwicklung-webdesign/webentwicklung/restful-webservices-1-was-ist-das-uberhaupt. Zuletzt besucht: 01.04.2016.

[Inc15] Xamarin Inc. Xamarin, 2015. URL https://www.xamarin.com/. Zuletzt besucht: 01.04.2016.

[JSO16a] JSON. Einführung in JSON, 2016. URL http://www.json.org/json-de.html. Zuletzt besucht: 01.04.2016.

[JSO16b] JSON. JSON Beispiel, 2016. URL http://json.org/example.html. Zuletzt besucht: 01.04.2016.

[Kim16] Kimtag. NFC Tags Explained, 2016. URL http://kimtag.com/s/nfc_tags. Zuletzt besucht: 01.04.2016.

[Ler12] Brian Leroux. PhoneGap, Cordova, and what's in a name?, 2012. URL http://phonegap.com/blog/2012/03/19/phonegap-cordova-and-whate28099s-in-a-name/. Zuletzt besucht: 01.04.2016.

[Mes12] Jeff Mesnil. STOMP Over WebSocket, 2012. URL http://jmesnil.net/stomp-websocket/doc/. Zuletzt besucht: 01.04.2016.

[Mic16] Microsoft. TypeScript, 2016. URL http://www.typescriptlang.org/. Zuletzt besucht: 01.04.2016.

[Mor16] Gunnar Morling. MapStruct: Java bean mappings, the easy way!, 2016. URL http://mapstruct.org/. Zuletzt besucht: 01.04.2016.

[NFC16] Open NFC. Tags compatibility, 2016. URL http://open-nfc.org/wp/home/documentation/tags-compatibility/. Zuletzt besucht: 01.04.2016.

[npm16] Inc. npm. Node Package Manager, 2016. URL https://www.npmjs.com/. Zuletzt besucht: 01.04.2016.

[Ora16] Oracle. Java Servlet Technology, 2016. URL http://www.oracle.com/technetwork/java/index-jsp-135475.html. Zuletzt besucht: 01.04.2016.

[Pet14] Nick Pettit. The 2014 Guide to Responsive Web Design, 2014. URL http://blog.teamtreehouse.com/modern-field-guide-responsive-web-design. Zuletzt besucht: 01.04.2016.

[Pro16] Greg Proehl. RFID und NFC, 2016. URL http://www.mouser.at/applications/rfid-nfc-introduction/. Zuletzt besucht: 01.04.2016.

[PS16a] Inc. Pivotal Software. Messaging with JMS, 2016. URL https://spring.io/guides/gs/messaging-jms/. Zuletzt besucht: 01.04.2016.

[PS16b] Inc. Pivotal Software. Spring Framework, 2016. URL https://spring.io/. Zuletzt besucht: 01.04.2016.

[Rah12] Frank Rahn. Einführung in das Spring Framework, 2012. URL https://www.frank-rahn.de/einfuehrung-spring-framework/#toggle-id-1. Zuletzt besucht: 01.04.2016.

[Rip14] Ben Ripkens. Ionic: An AngularJS based framework on the rise, 2014. URL https://blog.codecentric.de/en/2014/11/ionic-angularjs-framework-on-the-rise. Zuletzt besucht: 01.04.2016.

[SAS16] SASS. Sass Basics, 2016. URL http://sass-lang.com/guide. Zuletzt besucht: 01.04.2016.

[Sol15] Chariot Solutions. PhoneGap NFC Plugin, 2015. URL https://github.com/chariotsolutions/phonegap-nfc. Zuletzt besucht: 01.04.2016.

[Sta16] Wirtschaftsvereinigung Stahl. Stahl im Fluss: Vergießen des Stahls, 2016. URL http://www.stahl-online.de/index.php/themen/stahltechnologie/stahlerzeugung/3/. Zuletzt besucht: 01.04.2016.

[STM15] STMicroelectronics. 2048-bit EEPROM tag IC at 13.56 MHz, with 64-bit UID and kill code, ISO 15693 and ISO 18000-3 Mode 1 compliant, 2015. URL http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/CD00092187.pdf. Zuletzt besucht: 01.04.2016.

[STO12] STOMP. Simple Text Oriented Messaging Protocol, 2012. URL http://stomp.github.io/. Zuletzt besucht: 01.04.2016.

[Tec16] Primetals Technologies. Primetals Technologies, 2016. URL http://primetals.com/. Zuletzt besucht: 01.04.2016.

[VMw16a] Inc VMware. SockJS, 2016. URL https://github.com/sockjs. Zuletzt besucht: 01.04.2016.

[VMw16b] Inc VMware. SockJS-client, 2016. URL https://github.com/sockjs/sockjs-client. Zuletzt besucht: 01.04.2016.