HOCHSCHULE **TRIER**
Trier University of Applied Sciences
Informatik - Computer Science

Entwicklung mobiler Applikation zur zentralen Verwaltung WG-typischer Aufgaben

Development of an mobile application for central administration to manage flat sharing tasks

Tobias Barwig, Robert Raschel, Simon Ritzel

Bachelor-Projektarbeit

Betreuer: Prof. Dr. Georg Rock

Trier, 18.12.2014

Kurzfassung

In dieser Projektarbeit wird die Problematik der Verwaltung von Haushaltsaufgaben in einer Wohngemeinschaft mit meist jungen erwachsenen Studenten behandelt. Ziel ist das oftmals auftretende Chaos (durch Zettelwirtschaft und schlechter Kommunikation) möglichst gering zu halten und die anstehenden Aufgaben und Termine jederzeit klar definiert jedem Mitbewohner zugänglich zu machen. Dieses Ziel soll durch eine mobile Applikation auf Basis des von Google entwickelten Betriebssystems Android erreicht werden. Als Trägermedium dient hierbei ein handelsübliches Smartphone. Die App bietet jedem Mitbewohner die Möglichkeit Notizen sowie Kalendereinträge einzusehen und zu erstellen. Außerdem ist es Möglich eine Einkaufsliste zu verwalten in der Artikel hinzugefügt, gelöscht und als gekauft markiert werden können. Des Weiteren zeigt ein Putzplan die noch anstehenden bzw. bereits erledigten Aufgaben aus dem WG Haushalt an. Für den Putzplan muss vor dem ersten Gebrauch der WG-Administrator, der auch unter anderem die Benutzer verwaltet, selbst definierbare Aufgaben mit Beschreibung, Zyklus und Mitbewohner anlegen.

This project thesis displays the often problematic administration of the various household tasks within a flatsharing student community. Objective is to reduce the mostly chaotic atmosphere (be it due to jumble of bits of paper or bad communication) by defining due tasks and deadlines and publishing them to every flatmate in an easy manner. This should be reached by using a mobile application based in an operation system from Google, called ?Android?. Carrier medium should be a common smartphone. The application out every roommate into position to manage and create different notes and calender appointments. Furthermore it is possible to manage a shopping list where articles can easily be added, deleted or marked. In addition to this shows

Inhaltsverzeichnis

Abbildungsverzeichnis

1

Einleitung

Haushaltsaufgaben müssen sinnvoll und einfach auf alle Mitbewohner verteilt werden und der aktuelle Stand zu jedem beliebigen Zeitpunkt für jeden einsehbar sein.

Die App bietet jedem Mitbewohner die Möglichkeit Notizen einzusehen und zu erstellen. Außerdem ist es Möglich eine Einkaufsliste zu verwalten, in der Artikel hinzugefügt, gelöscht und als gekauft markiert werden können. Des Weiteren zeigt ein Putzplan die noch anstehenden bzw. bereits erledigten Aufgaben aus dem WG Haushalt an. Jedes WG-Mitglied kann sich in der App einloggen. Dass ein Smartphone als neues Trägermedium dient, liegt nahe, weil nahezu alle der durchweg jungen erwachsenen Personen der Zielgruppe solch ein Gerät besitzen. Außerdem bietet es mit der hohen Konnektivität die perfekte Grundlage alle Mitbewohner jederzeit auf dem gleichen Wissensstand zu halten. Als Betriebssystem kommt die von Google entwickelte Android Plattform zum Einsatz. Durch deren hohen Marktanteil von 68,2% in Deutschland wird hier die größte Anzahl an potentiellen Nutzern erreicht (siehe Abb. **??**).

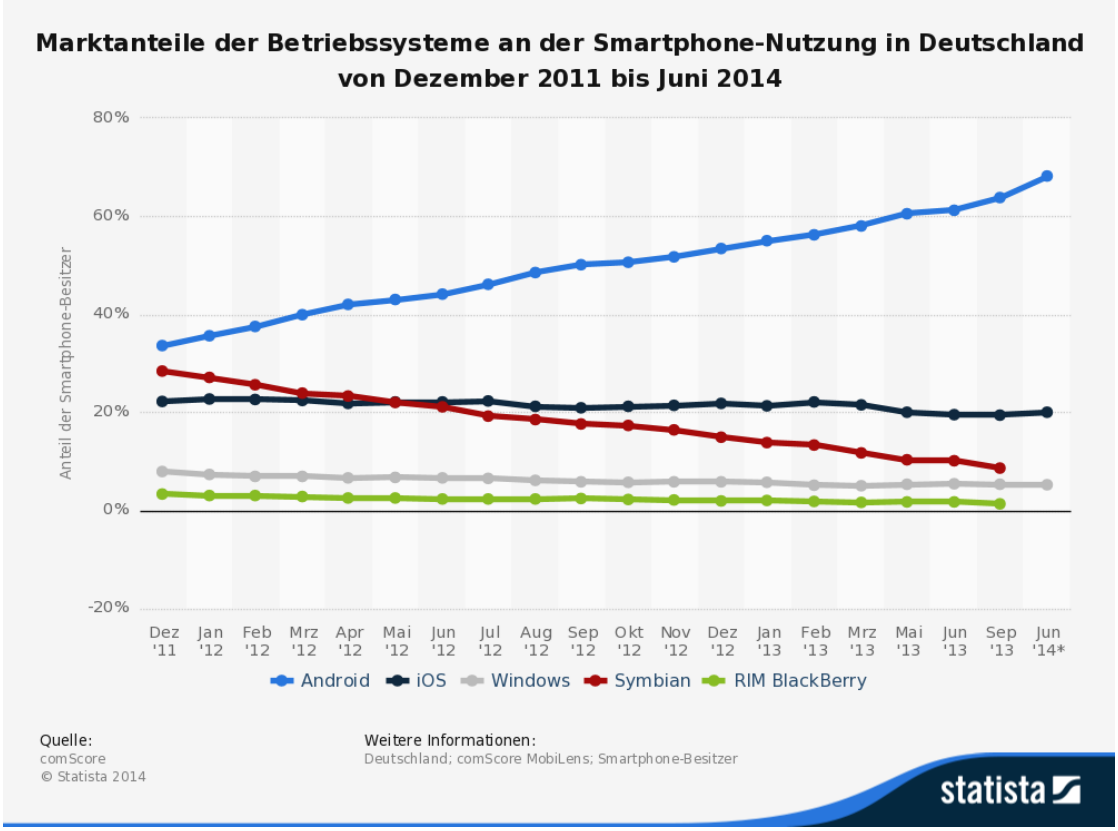

Abb. 1.1. Marktanteile der Betriebssysteme an der Smartphone-Nutzung in Deutschland von Dezember 2011 bis Juni 2014

1.1 Zielsetzung

Einer der WG-Mitbewohner erklärt sich bereit die Aufgaben des WG-Adminstrators zu übernehmen. Dieser WG-Administrator registriert sich und legt dabei eine neue WG an. Zu seinen Aufgaben gehört unter anderem die Verwaltung der Mitbewohner sowie das Pflegen des Putzplans. Neue Mitbewohner müssen vom WG-Administrator per E-Mail in die WG eingeladen werden. Für den Putzplan definiert er Aufgaben die in einem einstellbaren Rhythmus wiederholt werden und wählt zu jeder Aufgabe einen Mitbewohner aus. Nun beginnt der Rhythmus zu laufen und die Aufgaben wechseln nach Erledigung automatisch zu der nächsten Person aus der WG. Auf dem Schwarzen Brett können Einträge angezeigt, erstellt und gelöscht werden. In der Einkaufsliste können Artikel hinzugefügt und entfernt werden. Wurde ein Artikel gekauft, kann derjenige den Artikel als gekauft markieren. Alle Änderungen eines Mitbewohners ist für alle anderen Mitglieder der WG nach einer kurzen Synchronisation sichtbar. Alle Informationen einer WG werden serverseitig in einer Datenbank und clientseitig auf dem Smartphone des Benutzers gespeichert. Bei jedem Start der App wird ein Datenabgleich der auf Client-

und Serverseite gespeicherten Informationen durchgeführt und alle voneinander abweichenden Daten auf einer Übersichtsseite dem Benutzer als „Neu“ aufgelistet.

1.2 Ähnliche Apps

Es gibt bereits eine Auswahl an Apps, die sich jeweils an einem kleinen Teilbereich unseres Funktionsumfanges orientieren und dies gut umsetzen. Hierbei sind einzelne Apps für Einkaufslisten wie „Shopping List“[1] oder Putzpläne wie „Roomboard - Cleaning Roster“[2]. Jedoch gibt es eine weitere App die sich stark an unserer Idee mit dem Funktionumfang orientiert. Die App „Flatastic: Die WG-App“[3] bietet neben einer Einkaufsliste, einem Putzplan und einer Pinnwand zusätzlich einen Ausgabenrechner, womit alle für die WG getätigten Einkäufe zusammengerechnet werden können. Wir beschränken uns in dieser Ausarbeitung dennoch weiter auf unseren festgelegten Funktionsumfang und können uns nach der Fertigstellung nach wie vor dazu entscheiden weitere Zusatzfunktionen zu implementieren.

[1] „Shopping List“ in Google Play Store
[2] „Roomboard- Cleaning Roster“ in Google Play Store
[3] „Flatastic: Die WG-App“ in Google Play Store

2

Vorgehensweise

Um mögliche Probleme bei der Implementierung der App bereits früh zu identifizieren und den Arbeitsumfang der einzelnen Funktionen besser ermitteln zu können, haben wir vor Beginn der Implementierung die Risiken analysiert und uns für ein geeignetes Vorgehensmodell im Projektmanagement entschieden.

2.1 Projektmanagement

Vor der eigentlichen Implementierung eines Softwareprojektes liegt meist mehr schriftliche Arbeit, die meist mit Stift und Papier zu erledigen ist, als man denkt. Um in einem Team effektiv zu arbeiten, ist es nicht nur wichtig auf das gleiche Ziel hinzuarbeiten, sondern ebenso wichtig ist, gemeinsam auf dem gleichen Weg zum Ziel zu arbeiten. Dafür werden wir uns intern, sowie mit Absprache des Kunden verschiedene Meilensteine setzen. Zu jedem abgeschlossenen Meilenstein, ob nach einem großen oder kleinen Fortschritt, treffen wir uns im Team an einem Tisch und besprechen die Ergebnisse jeder Person um sie am Ende zusammenzutragen. Des Weiteren wird es in Absprache mit dem Kunden nach jedem abgeschlossenen großen Meilenstein ein Treffen bei dem Kunden geben. So können wir uns zu jedem Zeitpunkt des Projektes sicher sein, auf die Wünsche und Vorgaben des Kunden richtig einzugehen. Ausserdem ist der Kunde so direkt mit in die Entwicklung integriert.
Zu Beginn müssen wir für unser Projektmanagement das passende Modell finden. Nach reichlicher Überlegung haben wir uns dafür entschieden mit Use Cases, Mockups und Prototypen das Projekt zu beginnen. Dadurch war schnell mit dem "Prototyping"das passende Modell für unser Vorhaben gefunden. Es gibt verschiedene Arten von "Prototyping", die alle eine andere Herangehensweise bieten. Bei näherem Betrachten stellt sich heraus, dass das "horizontale Prototyping"genau die Art von "Prototyping"bietet, die wir bevorzugen. Dabei wird eine Ebene des Gesamtsystems möglichst genau entworfen oder implementiert und genau möchten wir mit den Mockups und dem Prototypen umsetzen, lediglich die Use Cases falle ein wenig aus dem Rahmen. Weitere Details finden Sie in den folgenden Unterkapiteln.
Da wir allerdings am Ende ein Lastenheft erstellen möchten und wir während

der Projektzeit Meilensteine definieren und zu Meilensteinsitzungen einladen werden, wird unser Management nicht nur alleine auf dem "horizontalen Prototyping"basieren, sondern zusätzlich Einflüsse aus dem "Wasserfallmodell"beinhalten. Dieses Modell zeichnet sich durch die Organisation in Phasen aus. Dabei werden kleine Meilensteine während einer Phase und große Meilensteine am Ende einer Phase definiert und in Meileinsteinsitzungen die Ergebnisse besprochen. Die wichtigsten Dokumente aus dem Wasserfallmodell ist unter anderem das Lastenheft.
Damit haben wir nun als Vorgehensmodell für unser Projektmanagement das "Horizontale Prototyping mit Einflüssen aus dem Wasserfallmodell".

2.1.1 Use-Cases

Ein Use-Case (dt. Anwendungsfall) schildert auf einfacher Ebene alle möglichen Szenarien die zu einem bestimmten Ziel führen können. Am Anfang steht ein Akteur (z.B. der Benutzer der App) der ein bestimmtes Ziel im System erreichen möchte (z.B. das erstellen einer Notiz). Der Use-Case beschreibt nun grob das System, die Umwelt in welchem das System arbeitet, die Anforderungen falls gegeben und letztendlich alle nötigen Schritte, welche vom Benutzer getätigt werden müssen, um das Ziel zu erreichen. Dazu können auch automatisierte Ereignisse vom System gehören.
Inwieweit der Use-Case ins Details geht ist nicht definiert. Der Ersteller kann hier selbst entscheiden wie sehr eine Aktion abstrahiert werden kann. Die Entscheidung kann von Use-Case zu Use-Case variieren. Es empfiehlt sich eine Vorlage mit relevanten Punkten zu erstellen, die dann bei jedem Use-Case möglichst einheitlich ausgefüllt werden kann. Für eine grafische Modellierung und der damit verbundenen Übersicht über das Gesamtsystem, können am Ende die textbasierten Use-Cases in ein Use-Case-Diagramm vereint werden. Hierfür ist in der Softwarentwicklung die Modellierungssprache Unified Modelling Language (dt. vereinheitlichte Modellierungssprache, kurz: UML) zu empfehlen. Sie wird für Version 2.4.1 unter anderem von der International Organization for Standardization (dt. Internationale Organisation für Normung, kurz: ISO, ISO/IEC 19505) standardisiert.

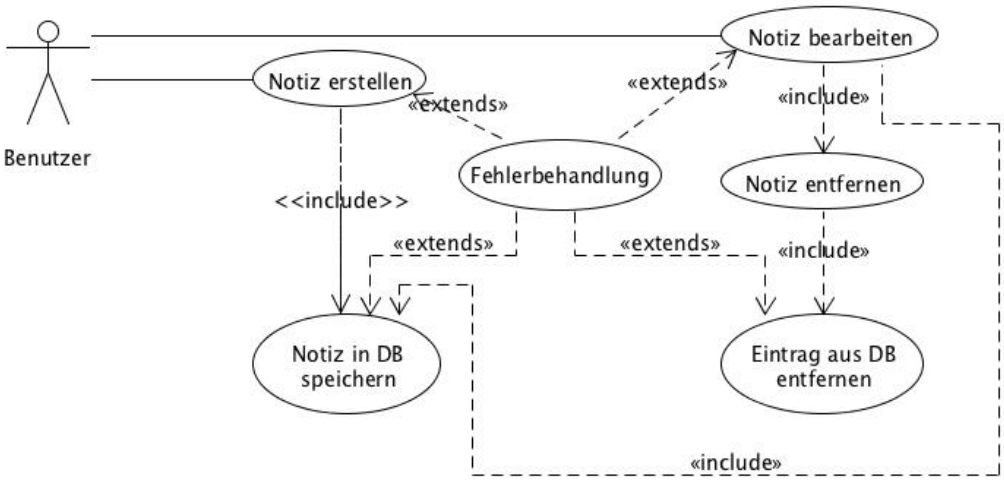

Abb. 2.1. Use-Case-Diagramm für das Blackboard

```
1  Use-Case: Notiz bearbeiten
2  Ziel: Unsortierte Liste von Notizen angezeigt
3  Kategorie: optional (nützlich, nicht unbedingt notwendig)
4  Vorbedingung: Benutzer ist eingeloggt (in App)
5  Nachbedingung Erfolg: Mitteilung an Benutzer, dass Eintrag erfolgreich
6  Nachbedingung Fehlschlag: Mitteilung an Benutzer, dass Eintrag fehlgeschlagen
7  Akteure: Benutzer der App (nur registrierte WG Mitglieder)
8  Auslösendes Ereignis: Erstellung einer neuen WG, Erstellung eines Eintrags
       von Benutzer
9  Beschreibung:
10 Notiz erstellen
11 1 Benutzer nimmt Änderungen an Notiz vor.
12 2 Benutzer möchte Notiz speichern.
13 3. Notiz in Datenbank speichern.
14 Alternativen:
15 2a Benutzer entschließ sich die Änderungen zu verwerfen. Benutzer gelangt auf
       vorherigen Bildschirm
16 2b Benutzer entschließt sich die Notiz zu löschen. Der Datenbankeintrag wird
       gelöscht.
```

Abb. 2.2. Beschreibung des Use-Cases *Notiz bearbeiten*

2.1.2 Mockups

2.1.3 Lastenheft

um Projektstart wird ein Lastenheft entwickelt, in dem das Konzept und alle Funktionen mit deren Anforderungen und dem voraussichtlichen Arbeitsaufwand beschrieben werden. Im Anschluss an das Lastenheft wird für jeden elementaren Screen der App ein Mockup angefertigten, an denen für alle Entwickler die Idee sowie eine grobe Einteilung der designtechnischen Elemente zu erkennen sein soll. Da die App mit einer Datenbankverbindung auf einen externen Server arbeiten

wird, haben wir uns darüber hinaus dazu entschieden, einen Prototypen zu entwerfen. Über die Verbindung zur externen Datenbank werden die Informationen der Benutzer gespeichert und geladen. Diese Vorgänge soll der Prototyp simulieren und somit über die möglichen Fehlerquellen und den zeitlichen Aufwand für die Implementierung näher Aufschluss geben.
Durch das Lastenheft, welches in der ersten Phase des "Wasserfallmodellsäuftritt, sowie der Mockups und dem Prototypen, welche im "Prototypingßu finden sind, entscheiden wir uns für das "Prototyping"mit Einflüssen aus dem "Wasserfallmodell".

2.1.4 Horizontales Prototyping

Durch das Management Modell "Prototypingßind wir frühzeitig in der Lage, die Realisierbarkeit einzelner Teilaspekte zu ermitteln. Bei "horizontalem Prototyping"handelt es sich um eine spezielle Uterart von "Prototyping", bei dem eine spezielle Ebene des Gesamtsystems möglichst genau entworfen wird. Diese kann dann im weiteren Verlauf der Implementierung weiter verwendet werden. Diese Unterart von "Prototyping"passt zu den Mockups, sowie zu dem Datenbank-Prototyp. Es standen noch weitere Unterarten von "Prototypingßur Auswahl, wobei letztendlich das "horizontale Prototypingäm besten auf unser Projektmanagement, so wie wires uns vorgestellt haben, gepasst hat.
In unseren Mockups entwerfen wir Vorab die GUI der App möglichst genau, ohne auf darunterliegende Ebenen näher einzugehen. Wir haben bereits n unsere Mockups das Screendesign einfließen lassen. Durch die Kombination von Mockup und Screendesign gelang es uns detailreiche Prototypen zu gestalten, auf denen im späteren Entwicklungsverlauf das Design basieren konnte.

Durch die Wir haben mit damit so weit wie Möglich ein nachträgliches Screendesign ersetzt. Wir Das gleiche gilt für den Datenbank-Prototyp in dem wir lediglich die Ebene der Verbindung mit dem Speichern und Laden von Daten implementieren. Der Fokus soll bei beiden Ausführungen auf das wesentliche beschränkt sein.

Die Verbindung über eine bestehende Internetverbindung zur verteilten Datenbank ist der Teilaspekt mit dem größten Fehlerpotential. Darum entwerfen wir für diesen Bereich der Implementierung einen Prototypen. Weitere Details zum Prototypen finden Sie im Kapitel **??** auf Seite **??**.

2.2 Risiken

- Datenbankverbindung - keine Internetverbindung - Server der Datenbank nicht erreichbar -

3

Prototyp

Um vermeidbare Verzögerungen in der späteren Entwicklung möglichst auszuschließen, wurde der Einsatz von Prototypen entschieden. Dazu werden die benötigten Komponenten genauer betrachtet und auf ihre Umsetzbarkeit hin untersucht. Bei einer Android-App, die auf Inhalte aus einer Datenbank zugreift, ist das die Schnittstelle zur Datenbank, beziehungsweise die Netzwerkverbindung.
Da eine direkte Verbindung zum Datenbankserver aus dem Android Betriebssystem nicht möglich ist, wurde die Authentifizierung des Benutzers als weitere Problemstelle markiert. Den die Verwendung von Datenbankbenutzern fällt dadurch weg und muss andersweitig gelöst werden. Um bei diesen kritischen Stellen nicht in bredouille zu geraten, wurde dafür ein Prototyp entwicklt.

3.1 Prototyp Implementierung

Als eine einfache und schnelle Art der Umsetzung haben wir uns für die PHP-Variante entschieden. Dabei werden die Daten von einem PHP-Skript aus der Datenbank gelesen und in eine JSON-Datenformat gebracht. Dieses Objekt wird in einem HTTP-Paket an die App übertragen. In der App sorgt dann ein JSON-Parser für das Auslesen der Daten, welche anschließend direkt verwertet werden oder zunächst in der lokalen SQLite-Datenbank vorgehalten werden.
Alternativ zu dieser Lösung wäre ein RESTful Web Service gewesen. Dieser Lösungsansatz wäre jedoch mit einem größeren programmiertechnischen Aufwand, sowie einer umfangreicheren Serverkonfiguration verbunden gewesen.
Da beide Schwerpunkte in einem Prototyp getestet wurden, wird auf eine weitere Trennung verzichtet.

3.1.1 Datenbankstruktur

Begonnen wurde die Umsetzung mit der Definition der Datenbankstruktur sowie deren Umsetzung. Dazu wurden zwei einfache Tabellen angelegt. Die Tabelle `tp_test??` wird für die Schreib- und Lesevorgänge verwendet. Um alle nötigen Datentypen testen zu können wurden verschiedene Spalten verwendet. Dadurch konnte auch der spätere Einsatz besser simuliert werden.

```
1  CREATE TABLE IF NOT EXISTS `test_tp` (
2      `uid` VARCHAR(23) NOT NULL,
3      `msg` TEXT,
4      `nmbr` INT(11) DEFAULT NULL,
5      `created_at` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
6  ) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

Abb. 3.1. Aufbau der Tabelle tp_test

Um die Benutzerverwaltung für den Prototypen zu simulieren wurden außerdem noch eine Tabelle `users ??` angelegt. Darin wurden Informationen zu den Benutzern hinterlegt. Zum Beispiel eine eindeutige ID, sowie Vor- und Nachname. Zum Authentifizieren wurde die E-Mailadresse und ein beliebiges Passwort verwendet. Um das Passwort nicht im Klartext zu speichern, wurde es zusammen mit einem Salt als Hash-Wert abgelegt.

```
1  CREATE TABLE IF NOT EXISTS `users` (
2    `uid` int(11) NOT NULL AUTO_INCREMENT,
3    `unique_id` varchar(23) NOT NULL,
4    `firstname` varchar(50) NOT NULL,
5    `lastname` varchar(50) NOT NULL,
6    `username` varchar(20) NOT NULL,
7    `wgId` int(11) NOT NULL,
8    `email` varchar(100) NOT NULL,
9    `encrypted_password` varchar(80) NOT NULL,
10   `salt` varchar(10) NOT NULL,
11   `created_at` datetime DEFAULT NULL,
12   PRIMARY KEY (`uid`)
13 ) ENGINE=MyISAM  DEFAULT CHARSET=utf8 AUTO_INCREMENT=5 ;
```

Abb. 3.2. Aufbau der Tabelle users

3.1.2 PHP-Skripte

Der Aufbau der PHP-Schnittstelle ist simpel umgesetzt, da nicht viele Funktionen für den Prototyp benötigt werden. Trotzdem wurde auf eine übersichtliche Datei- und Ordnerstruktur, als auch auf einen modularen Aufbau geachtet. Wie in Abbildung ?? ?? zu sehen, wurden der Aufbau zunächst in zwei Kategorien aufgeteilt. Funktionen, die direkt auf der Datenbank ausgeführt werden, sowie das Verbinden und Bereitstellen des Datenbankobjekts übernehmen, sind im Ordner

```
php
├── include
│   ├── config.php ... Verbindungsparameter für die
│   │                  Datenbank.
│   ├── DB_Connect.php ... Funktionen zum Aufbau
│   │                      und dem Bereitstellen der
│   │                      Datenbankverbindung.
│   └── DB_Functions.php ... Methoden für
│                            Datenbankoperationen; zum
│                            Beispiel getTable.
├── db.php ... Verarbeitung des
│              HTTP-Requests, auswerten der
│              Anfrage, sammeln der Daten und
│              abschließend Daten verpacken
│              und an Client zurücksenden.
└── login.php ... Authentifizieren des
                  Benutzers.
```

Abb. 3.3. Struktur der PHP-Skripte im Dateisystem

`php/include` untergebracht. Alle weiteren Funktionen die aus der App heraus erreichbar sein sollen, befinden sich im Hauptordner `php`.

Die vollständigen Skripte befinden sich im Anhang und werde hier nur in Auszügen dargestellt.

Der Ablauf zum Aufrufen der einzelnen PHP-Funktionen ist immer derselbe. Dazu wird ein POST-Request an den Server geschickt, der die entsprechende Ressource, in diesem Fall entweder `db.php` oder `login.php` anfordert. Als POST-Parameter werden neben den Werten für die Funktion, zum Beispiel E-Mailadresse und Passwort für den Login, noch ein TAG-Parameter angehängt. Der TAG-Parameter ist für den Login-Prozess zum Beispiel leer, zum Ändern des Passworts kann dazu `chgpass` eingesetzt werden. Nachdem die zum TAG passende Funktion gefunden wurde, werden die übertragenen Werte ausgelesen und entsprechend verarbeitet.

Eine wichtige Funktion der PHP-Skripte ist der Login, beziehungsweise die Authentifizierung des Benutzers.

Dazu wird die Adresse `http://test.app1.raschel.org/php/db.php` mit den Parametern `TAG=''`, `EMAIL=''ritzels@fh-trier.de''` und `PASSWORD=''ritzels''` aufgerufen (Vgl. **??**). Sind alle Parameter korrekt übertragen worden, wird die Funktion `getUserByEmailAndPassword($email, $password)`, die sich im Skript `DB\_Functions.php` befindet, aufgerufen. Zum Authentifizieren wird aus der Datenbank der zur E-Mailadresse passende Datensatz geladen, siehe Auszug **??**. Das in der Datenbank gespeicherte `SALT` wird mit dem übertragenen Passwort an die Funktion `checkhashSSHA($salt, $password)` übergeben, welche den Hash aus `SALT` und Passwort bildet und zurückgibt. Dieser generierte Hash wird dann mit dem in der Datenbank gespeichertem `encrypted\_password` verglichen. Stimmen die beiden Hashs überein, wird der geladene Datensatz an die Login-Funktion zurückgeliefert, ansonsten liefert die Funktion `getUserByEmailAndPassword() -> false`.

```
1  <?php
2    /**
3     PHP API for Login, Register, Changepassword,
4     Resetpassword Requests and for Email Notifications.
5   **/
6    [...]
7    if (isset($tag) && $tag != '') {
8    // Include Database handler
9    require_once 'include/DB_Functions.php';
10   $db = new DB_Functions();
11   // response Array
12   $response = array("tag" => $tag, "success" => 0, "error" => 0);
13   // check for tag type
14   if ($tag == 'login') {
15   // Request type is check Login
16   $email = $_POST['email'];
17   $password = $_POST['password'];
18   // check for user
19   $user = $db->getUserByEmailAndPassword($email, $password);
20     if ($user) {
21       // user found
22       // echo json with success = 1
23       $response["success"] = 1;
24       $response["user"]["user_id"] = $user["user_id"];
25       $response["user"]["wg_id"] = $user["wg_id"];
26       $response["user"]["crea"] = $user["crea"];
27
28       echo json_encode($response);
29     } else {
30       // user not found
31       // echo json with error = 1
32       $response["error"] = 1;
33       $response["error_msg"] = $IncorrectEMail;
34       echo json_encode($response);
35     }
36   }
37   else if ($tag == 'chgpass') {
38   [...]
```

Abb. 3.4. Auszug aus login.php

Im Kapitel refsec:App werden noch weitere Funktionen erläutert, die in diesem Kapitel noch keine Rolle spielen. Die Skripte befinden sich in vollem Umfang noch als Anhang an diese Arbeit.

```
1       [..]
2       private $db;
3
4      // constructor
5      function __construct() {
6          require_once 'DB_Connect.php';
7          // connecting to database
8          $this->db = new DB_Connect();
9          $this->db->connect();
10     }
11       /**
12       * Verifies user by email and password
13       */
14      public function getUserByEmailAndPassword($email, $password) {
15          $resultSQL = mysql_query("SELECT * FROM user WHERE email = '$email'")
16              or die(mysql_error());
17          // check for result
18          $no_of_rows = mysql_num_rows($resultSQL);
19
20          if ($no_of_rows > 0) {
21              $result = mysql_fetch_array($resultSQL);
22              $salt = $result['salt'];
23              $encrypted_password = $result['passwort'];
24              $hash = $this->checkhashSSHA($salt, $password);
25              // check for password equality
26              if ($encrypted_password == $hash) {
27                  // user authentication details are correct
28                  return $result;
29              }
30          }
31          else {
32              // user not found
33              return null;
34          }
35      }
36      [..]
```

Abb. 3.5. Auszug aus DB_Functions.php

3.1.3 Prototyp-App

Die Implementierung der Prototyp-App wurde in zwei Projekte aufgeteilt. In dem Projekt `DatabaseConnectionLib` wurden die Funktionen zum Schreiben und Lesen der MySQL-Datenbank ausgelagert, da diese mit Sicherheit für die spätere Implementierung wieder Verwendung finden werden. Das Anzeigen und Manipulieren der Daten wurde im Projekt `DBPrototyp` zusammengefasst.

DBPrototyp

Das Projekt wurde noch in der Entwicklungsumgebung Eclipse erstellt und umfasst deshalb eine etwas tiefere Ordnerhirarchie. Da nicht alle für diese Implementierung relevant sind, werden nur die wichtigsten Ordner und Dateien näher erläutert.
Der Ordnerbaum **??** zeigt die wichtigen Ordner und Dateien, und erläutert kurz deren Funktion.

Für die Funktion des Prototyp und das Umsetzen der definierten Schwerpunkte sind in diesem Projekt die beiden Klassen `LoginActivity.java` und `MainActivity.java` von Bedeutung. Beim Start der App wird zunächst die

```
DBPrototyp
├── src
│   └── de.hst.teamprojekt.dbprototype
│       ├── LoginActivity.java ... Java-Klasse mit den Funktionen zum Login.
│       └── MainActivity.java ... Java-Klasse mit den Funktionen zum Lesen und
│                                  Schreiben der Daten.
├── res
│   ├── layout
│   │   ├── activity_login.xml ... Layout-Datei für die Login-Oberfläche.
│   │   └── activity_main.xml ... Layout-Datei für die Standard-Oberfläche.
│   ├── menu ... Beinhaltet die Menüstruktur der einzelnen
│   │            Oberflächen.
│   └── values ... Beinhaltet XML-Dateien, in denen Werte
│                  hinterlegt werden können die nicht im Quellcode
│                  sein sollten.
└── AndroidManifest.xml ... Allgmeine Parameter der App, wie unterstützte
                            API-Version oder benötigte Ressourcen, bzw.
                            Erlaubnis für die Ressourcen.
```

Abb. 3.6. Verzeichnisstruktur des DBPrototyp-Projekts

`MainActivity.java` mit dem Layout `activity_main.xml` geladen **??**.
Das geschieht in Zeile **??** durch `setContentView(R.layout.activity_main)` in der `onCreate()` Methode.

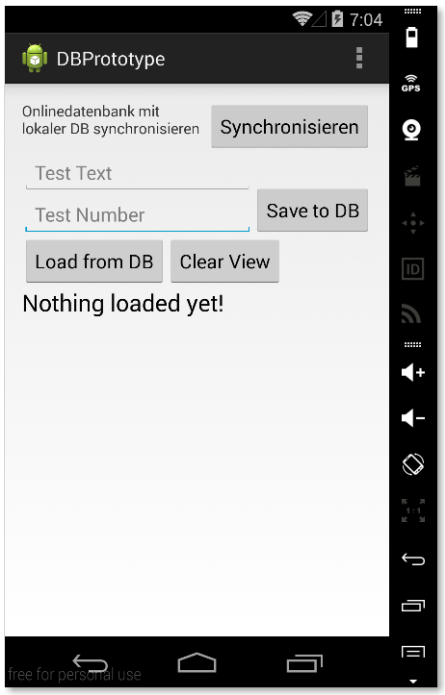

Abb. 3.7. Start-Oberfläche

Ist alles vollständig geladen, kann man sich über das Menü anmelden. Dafür wird in der `onOptionsItemSelected(MenuItem item)` ein `startActivity()` aufgerufen, welches die `LoginActivity.java` startet **??**.

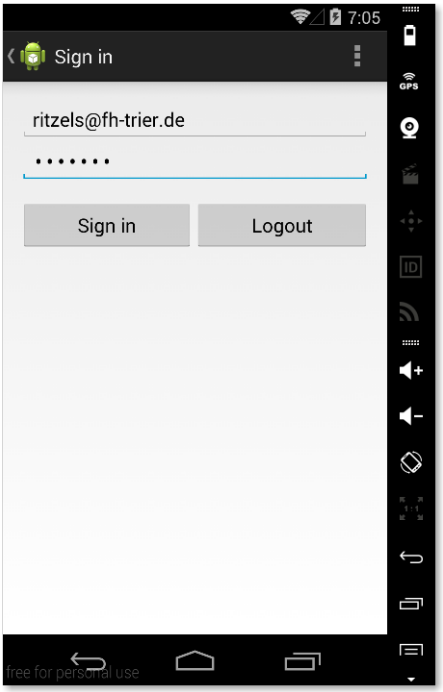

Abb. 3.8. Login-Oberfläche

Nach der erfolgreichen Authentifizierung wird über den Button `Synchronisieren` die Funktion `synchronizeDbs(View view)` **??** aufgerufen. Dies generiert ein Objekt der Klasse `SyncRemoteDatabase.java` **??**. Dabei handelt es sich um eine von `AsyncTask` abgeleitete Klasse, welche die Testdaten über das PHP-Skript aus der Datenbank abruft und anschließend in die lokale `SQLite`-Datenbank schreibt. Diese Klasse wurde jedoch in die `DBConnectionLib` **??** ausgelagert und anschließend an diese Kapitel genauer erläutert.

Wurden die Tabellen erfolgreich synchronisiert, kann über die beiden Felder Testwerte eingegeben werden. Durch einen klick auf den Button `Save to DB` werden die Daten zunächst in der Methode `saveToDb(View view)` **??** in *BasicNameValuePair*-Objekte überführt (Zeile **??** - **??**). Zum Speichern werden diese Objekte an eine Instanz der Klasse `InsertIntoDatabase.java` übergeben. Diese ist, wie die Klasse `SyncRemoteDatabase.java`, eine Subklasse von `AsyncTask` und wird ebenfalls im Kapitel **??** genauer erklärt. Zum Anzeigen der lokalen Tabelleninhalte kann der Button `Load from DB` betätigt werden, wodurch die Funktion `loadFromDb(View view)` ausgeführt wird. Dabei werden die Daten zunächst in einem *Cursor*-Objekt bereitgestellt, welches vom `DatabaseHandler`**??** bereitgestellt wird (vgl. Zeile **??**). Die Darstellung wird durch eine *ListView* übernommen, die im Layout der Aktivität hinterlegt und über den Befehl //*

```
1  [...]
2  @Override
3  protected void onCreate(Bundle savedInstanceState) {
4    super.onCreate(savedInstanceState);
5    setContentView(R.layout.activity_main);
6  [...]
7  public void synchronizeDbs(View view) {
8    {...}
9    SyncRemoteDatabase queryTask =
10     new SyncRemoteDatabase(MainActivity.this, this.creds);
11   queryTask.execute(Constants.TABLE_TEST);
12 }
13 [...]
```

Abb. 3.9. Auszug aus der MainActivity.java

`findViewById(R.id.listView1)` in Zeile **??** referenziert wurde. Um den *Cursor* in der *ListView* anzeigen zu können, muss dieser in einem *SimpleCursorAdapter* für die *ListView* zugänglich gemacht werden. Dazu wird in Zeile **??** dem Konstruktor des *SimpleCursorAdapter* zunächst der aktuelle *Context*, das gewünschte Layout der einzelnen Zeilen (`R.Layout.list_item`) sowie der *Cursor* `result` übergeben. Damit der Adapter weiß, welche Werte er in welche *TextView* packen soll, werden dem Konstruktor noch `from` und `to` übergeben.

Dabei handelt es sich bei dem Objekt `from` um eine *String*-Liste mit den Namen der Tabellenspalten. Die *int*-Liste `to` enthält die IDs der *TextViews* im verwendeten Zeilenlayout. Die Reihenfolge muss dabei beachtet werden. Abschließend erhält der Konstruktor noch eine Flag mit der das Verhalten bei Änderung der Datengrundlage gesteuert wird. Dieser Adapter wir mittels dem Befehl `.setAdapter(sca)` der *ListView* übergeben. Wodurch nun die Daten auf der Oberfläche sichtbar werden.

Zum Leeren der *ListView* wird der Button `Clear View` gedrückt, wodurch mittels der Methode *clearResultListView(View view)* **??** die *ListView* vom Adapter getrennt und unsichtbar gemacht wird.

```
1  [...]
2  public void saveToDb(View view) {
3    {...}
4    BasicNameValuePair tablename =
5      new BasicNameValuePair("table", Constants.TABLE_TEST);
6    BasicNameValuePair testText = new BasicNameValuePair("msg",
7          ((EditText) findViewById(R.id.editTestText)).getText().toString());
8    BasicNameValuePair testNumber = new BasicNameValuePair("nmbr", ((EditText)
9    findViewById(R.id.editTestNmbr)).getText().toString());
10
11   InsertIntoDatabase saveTask =
12     new InsertIntoDatabase(MainActivity.this, this.creds);
13   saveTask.execute(tablename, testText, testNumber);
14 }
15 public void loadFromDb(View view) {
16   {...}
17   Cursor result = db.getTestRow(1);
18
19   if (result.getCount() > 0) {
20       String[] from = new String[]
21       { Constants.KEY_UID, Constants.KEY_TEST_STRING,
22         Constants.KEY_TEST_INT, Constants.KEY_CREATED_AT };
23
24       int[] to = new int[]
25       { R.id.uid, R.id.msg, R.id.nmbr, R.id.created_at };
26
27       SimpleCursorAdapter sca = new SimpleCursorAdapter
28         (this, R.layout.list_item, result, from, to, 0);
29       ListView lv = (ListView) findViewById(R.id.listView1);
30       lv.setAdapter(sca);
31       lv.setVisibility(ListView.VISIBLE);
32   {...}
33 }
34 public void clearResultListVew(View view) {
35   TextView txtView = (TextView) findViewById(R.id.textView2);
36   txtView.setText(R.string.txtNothingLoaded);
37   txtView.setVisibility(TextView.VISIBLE);
38   ListView lv1 = (ListView) findViewById(R.id.listView1);
39   lv1.setAdapter(null);
40   lv1.setVisibility(ListView.INVISIBLE);
41 }
42 [...]
```

Abb. 3.10. Auszug aus der MainActivity.java

DatabaseConnectionLib

Das Projekt `DatabaseConnectionLib` wurde, wie bereits erwähnt, zum Auslagern von wiederverwendbaren Modulen erstellt. Aus diesem Grund befinden sich darin keine Layout, Values oder ähnliche Ressourcen die zur Darstellung und Bedienung nötig sind. Da die Ordner- und Dateistruktur der Projekte nahezu identisch ist, wird von einer weiteren Aufführung dieser Details abgesehen. In diesem Kapitel werden dementsprechend nur die Klassen näher erläutert, die im Kapitel **?? ??** schon erwähnt wurden.

Klasse SyncRemoteDatabase.java

Instanziiert wird diese Klasse in der `MainActivity.saveToDb()` Funktion. Um die Bedienbarkeit der Oberfläche nicht zu verhindern oder die App zum Einfrieren zu bringen, was durch eine langanhaltende Operation passieren könnte, wur-

de diese Klasse von der `AsyncTask`-Klasse abgeleitet. Diese ermöglicht eine komfortable Möglichtkeit mit Threads zu arbeiten und somit die Operationen vom GUI-Thread auszulagern. Der Vorteil gegenüber dem Ausführen eines einfachen `Runnable`-Objekts in einem gesonderten Thread ist, dass die `AsyncTask`-Klasse ableitbare Methoden besitzt mit denen bevor und während der Thread läuft, sowie nach dem Beenden des Threads, Operationen ausgeführt werden können. Im Fall der `SyncRemoteDatabase`-Klasse wird vor dem Ausführen des Task, mittels der Methode *onPreExecute()*, ein `ProgressDialog` erstellt (vgl. Zeilen **??** ff.). Anschließend wird im asynchronen Teil *doInBackground(String... params)*, die Funktion `syncRemoteTable(creds, params[0])??` aufgerufen und das damit erzeugte `JSONObject`-Objekt an die Methode *onPostExecute(JSONObject json)***??** übergeben. Da diese Methode, wie die *onPreExecute*, im GUI-Thread läuft, kann der `ProgressDialog` mit einer neuen Nachricht versehen werden **??** und die erhaltenen Daten aus dem `JSONObject`-Objekt extrahieren (Zeile **??**). Im selben Schritt werden die Testdaten der Testtabelle als neuen Datensatz angefügt (siehe Zeile **??**).

```
1  [...]
2  @Override
3  protected void onPreExecute() {
4    super.onPreExecute();
5    pDialog = new ProgressDialog(appContext);
6    pDialog.setTitle("Contacting Servers");
7    pDialog.setMessage("Query database ...");
8    pDialog.setIndeterminate(false);
9    pDialog.setCancelable(true);
10   pDialog.show();
11 };
12 @Override
13 protected JSONObject doInBackground(String... params) {
14 /** {... **/
15   JSONObject json = userFunction.syncRemoteTable(creds, params[0]);
16   return json;
17 }
18 @Override
19 protected void onPostExecute(JSONObject json) {
20 /** {... **/
21   pDialog.setMessage("Loading Test Data");
22   pDialog.setTitle("Getting Data");
23   JSONArray json_array = json.getJSONArray("result");
24   /** {...} **/
25     db.dropSyncTable();
26     for (int i = 0; i < Integer.parseInt(res); i++) {
27       JSONObject json_data = json_array.getJSONObject(i);
28       db.addTestRow(json_data.getInt(Constants.KEY_UID),
29         json_data.getString(Constants.KEY_TEST_STRING),
30         json_data.getInt(Constants.KEY_TEST_INT),
31         json_data.getString(Constants.KEY_CREATED_AT));
32     }
33 [...]
```

Abb. 3.11. Auszug aus SyncRemoteDatabase.java

Die angesprochene Funktion *syncRemoteTable()* befindet sich in der Klasse `UserFunctions.java` **??**. Darin wurden die Methoden zusammgefasst, die vom

Benutzer oder mit dem Benutzer zusammenhängen. Neben den Methoden *loginUser(...)* und *loggedInUser(...)*, enthält sie auch die Methode *syncRemoteTable(AuthCredentials creds, String table)* **??**, die zum Abrufen der Testdaten aus der MySQL-Datenbank verwendet wird.
Dazu werden zunächst die *POST*-Parameter in eine Liste aus `BasicNameValuePair`-Objekten gepackt, wie in den Zeilen **??** bis **??** zu sehen.

```
1  [...]
2  public JSONObject syncRemoteTable(AuthCredentials creds, String table) {
3    List<BasicNameValuePair> params = new ArrayList<BasicNameValuePair>();
4    params.add(new BasicNameValuePair("tag", syncTag));
5    params.add(new BasicNameValuePair("email", creds.getEmail()));
6    params.add(new BasicNameValuePair("password", creds.getPassword()));
7    params.add(new BasicNameValuePair("table", table));
8    JSONObject json = jsonParser.getJSONFromUrl(syncUrl, params);
9    return json;
10 }
11 [...]
```

Abb. 3.12. Auszug aus UserFunctions.java

Die so verpackten Parameter werden schließlich an den `JSONParser` **??** übergeben. Die darin enthaltene Funktion
getJSONFromUrl(String url, List<BasicNameValuePair> params) der **??** schickt die erhaltenen Attribute mittels `DefaultHttpClient()`- und `HttpPost`-Objekt zur angegebenen *url* (Zeile **??** ff). Die Serverantwort, die vom Socket über den `InputStream` erreichbar wird, wird durch einen `BufferedReader` lesbar. Der `BufferedReader` ist hier besonders geeignet, da dadurch ein zeilenweises Lesen ermöglicht wird, was bei *HTTP*-Kommunikationen das Standardverfahren ist (Zeile **??** und **??**). Jede Zeile wird an einen *StringBuilder* gehängt und nach dem Lesen der letzten Zeile von einem `String` (Zeile **??**) in das gewünschte *JSONObject* umgewandelt.

Die so verpackte Tabelle, wird dann mit einer `for`-Schleife, siehe Zeile **??** im **??** **??**, in die einzelnen Zeilen zerlegt und von der Funktion *addTestRow(...)* **??** in die lokale `SQLite`-Datenbank geschrieben (Zeile **??**). Dazu werden die einzelnen Werte in zunächst mit den zugehörigen Spaltennamen in ein sogenanntes `ContentValues`-Objekt geschrieben, was im Groben einer Tabellenzeile entspricht. Eine solche Kapselung ist theoretisch nicht nötig, da die Möglichkeit besteht, die Werte durch ein zusammengesetzten SQL-Statement in die Datenbank zu schrieben. Diese Lösung ermöglicht aber die Verwendung der *insert()*-Methode wie in Zeile **??**, wodurch der Code nicht nur weniger fehleranfällig, sondern auch übersichtlicher und sicherer wird.

Klasse InsertIntoDatabase

Ebenfalls aus der `MainActivity` heraus wird diese Klasse instantziiert und ausgeführt. Sie dient zum Schreiben von Testdaten in die `MySQL`-Datenbank und ist

```
1  public JSONObject getJSONFromUrl(String url,
2      List<BasicNameValuePair> params) {
3  [...]
4    DefaultHttpClient httpClient = new DefaultHttpClient();
5    HttpPost httpPost = new HttpPost(url);
6    httpPost.setEntity(new UrlEncodedFormEntity(params));
7    HttpResponse httpResponse = httpClient.execute(httpPost);
8    HttpEntity httpEntity = httpResponse.getEntity();
9    is = httpEntity.getContent();
10 [...]
11   BufferedReader reader =
12     new BufferedReader(new InputStreamReader(is, "iso-8859-1"), 8);
13   StringBuilder sb = new StringBuilder();
14   String line = null;
15   while ((line = reader.readLine()) != null) {
16     sb.append(line + "\n");
17   }
18   is.close();
19   json = sb.toString();
20 [...]
21 else {
22   jObj = new JSONObject(json);
23 }
24 [...]
```

Abb. 3.13. Auszug aus JSONParser.java

```
1  [...]
2  public void addTestRow(int id, String text, int number,
3      String created_at) {
4    SQLiteDatabase db = this.getWritableDatabase();
5    ContentValues vals = new ContentValues();
6    if (id > 0 & created_at != null & !created_at.equals("")) {
7      vals.put(Constants.KEY_UID, id);
8      vals.put(Constants.KEY_TEST_STRING, text);
9      vals.put(Constants.KEY_TEST_INT, number);
10     vals.put(Constants.KEY_CREATED_AT, created_at);
11   }
12   db.insert(Constants.TABLE_TEST, null, vals);
13   db.close();
14 }
15 [...]
```

Abb. 3.14. Auszug aus der DatabaseHandler.java

somit auch eine Subklasse von `AsyncTask`.
Analog zur **??** wird auch hier zunächst ein Objekt der Klasse erstellt, dem dann mit der *execute(...)*-Methode die Testwerte als Parameter übergeben werden. Nachdem der `ProgressDialog` erstellt wurde, werden die Daten in Schlüssel- und Wertevariablen gekapselt und der *insertIntoRemoteTable(...)*-Methode übergeben.
Um einen standardkonformen *HTTP*-Request abzuschicken, werden dort wieder die *POST*-Parameter in einer Liste gekapselt und dem `JSONParser` **??**, zum Abschicken der Anfrage übergeben. Die Antwort wird wieder analog zur **??** in der *onPostExecute()*-Methode ausgewertet und eine Benachrichtigung angezeigt, ob die Übertragung erfolgreich war oder nicht.

4

Implementierung

Die App wurde zum Großteil in der aktuell von Google empfohlenen Umgebung Android Studio umgesetzt. Zu Beginn fand die Entwicklung noch in Eclipse mit dem entsprechenden Plug-In statt. Jedoch erschwerten die Bugs und die Behäbigkeit der Eclipse-IDE den zügigen Fortschritt. Aus diesem Grund wurde nach dem Legen des Grundsteins das Projekt auf die neue Entwicklungsumgebung migriert, wo es auch fertiggestellt wurde. ...

4.1 App

4.2 Putzplan

4.2.1 Umsetzung

4.2.2 Probleme und Lösungen

5

Zusammenfassung und Ausblick

In diesem Kapitel soll die Arbeit noch einmal kurz zusammengefasst werden. Insbesondere sollen die wesentlichen Ergebnisse Ihrer Arbeit herausgehoben werden. Erfahrungen, die z.B. Benutzer mit der Mensch-Maschine-Schnittstelle gemacht haben oder Ergebnisse von Leistungsmessungen sollen an dieser Stelle präsentiert werden. Sie können in diesem Kapitel auch die Ergebnisse oder das Arbeitsumfeld Ihrer Arbeit kritisch bewerten. Wünschenswerte Erweiterungen sollen als Hinweise auf weiterführende Arbeiten erwähnt werden.

A

Glossar

DisASTer	DisASTer (Distributed Algorithms Simulation Terrain), A platform for the Implementation of Distributed Algorithms
DSM	Distributed Shared Memory
AC	Linearisierbarkeit (atomic consistency)
SC	Sequentielle Konsistenz (sequential consistency)
WC	Schwache Konsistenz (weak consistency)
RC	Freigabekonsistenz (release consistency)

B

Erklärung der Kandidaten

☐ Die Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen- und Hilfsmittel verwendet.

☐ Die Arbeit wurde als Gruppenarbeit angefertigt. Meine eigene Leistung ist ...

Diesen Teil habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Namen der Mitverfasser: ...

Datum

Unterschrift der Kandidaten