ulm university universität
uulm

Fakultät für
Ingenieurwissenschaften
und Informatik
Institut für
Datenbanken und
Informationssysteme

Konzeption und Entwicklung eines Moduls für mobile Arbeitslisten

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Tran Bao Dat Nguyen
tran.nguyen@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Dr. Rüdiger Pryss

2017

Fassung 10. März 2017

Satz: PDF-LaTeX 2_ε

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation und Zielsetzung . . . 2
- 1.2 Überblick und Aufbau der Arbeit . . . 3

2 Projektanforderung **4**
- 2.1 Funktionale Anforderungen . . . 4
  - 2.1.1 Serverseitige funktionale Anforderungen . . . 4
  - 2.1.2 Clientseitige funktionale Anforderungen . . . 5
- 2.2 Nicht-funktionale Anforderungen . . . 6
- 2.3 Zusammenfassung . . . 7

3 Architektur **8**
- 3.1 Projektaufbau . . . 8
  - 3.1.1 Webserver für mobile Arbeitsliste . . . 8
  - 3.1.2 Mobiler Client - Eine Android-App für die Arbeitsliste . . . 9
  - 3.1.3 Dokumentation - Voraussetzung und Entwicklungsumgebung 9
- 3.2 Datenbank . . . 10
  - 3.2.1 Beschreibung . . . 10
  - 3.2.2 Weitere Anmerkungen zur Datenbank . . . 13
- 3.3 Zusammenfassung . . . 14

4 Client- und Serverimplementierung **17**
- 4.1 RESTful Webservice in Java . . . 17
  - 4.1.1 de.ulm.uni.bachelor.nguyen.worklist_rest_api.database . . . 17
  - 4.1.2 de.ulm.uni.bachelor.nguyen.worklist_rest_api.model . . . 19
  - 4.1.3 de.ulm.uni.bachelor.nguyen.worklist_rest_api.resource . . . 21
  - 4.1.4 de.ulm.uni.bachelor.nguyen.worklist_rest_api.service . . . 24
  - 4.1.5 de.ulm.uni.bachelor.nguyen.worklist_rest_api.exception.* . . . 27
  - 4.1.6 REST Endpoints und Nachrichtenformat . . . 28
- 4.2 Client - WorklistApp in Android . . . 34
  - 4.2.1 MainActivity . . . 35

4.2.2 Anmerkung und Zusammenfassung 40
4.2.3 Vorstellung der Android-Applikation - WorklistModul 42

5 Abgleich der Projektanforderungen 55
5.1 Serveranforderungen 55
5.2 Clientanforderungen 55
5.3 Fehlende Komponenten 56
5.4 Ausnahmebehandlungen und Robustheit 56

6 Verwandte Arbeiten 58

7 Zusammenfassung 60

Literaturverzeichnis 61

1 Einleitung

Der Begriff Mobilität spielt eine zunehmend wichtige Rolle in unserer heutigen Informationsgesellschaft. Im Zuge der Vernetzung und Digitalisierung haben mobile Endgeräte, wie Smartphones oder Tablet-PCs, sich nicht nur im privaten Alltag etabliert, auch werden sie vermehrt in Produktions- und Dienstleistungsprozessen von Unternehmen integriert und eingesetzt. Aktuelle Themen und Inhalte in der Unternehmenspolitik z.B. mobile Sicherheit oder BYOD[1] zeigen, dass sich Unternehmen und andere Institutionen intensiv mit den Möglichkeiten und Risiken der Integration von mobilen Endgeräten in Geschäftsabläufen auseinandersetzen. Dahingehend stellt sich die Frage: Welchen Einfluss hat der Einsatz von mobilen Endgeräten auf die Geschäftsprozesse in Organisationen?

Mit der fortschreitenden Entwicklung und Integration von *Prozessmanagementsystemen*[2] und *Workflow-Managementsystemen*[3], gelangt der Industrie ein wichtiger Schritt, um langfristig Geschäftsprozesse anpassungsfähiger, kostengünstiger und effizienter zu gestalten. Trotz teilautomatisierte Prozessabläufe oder die Integration von PMTS oder WFMTS in Desktop-Systemen, erweist sich die Prozessdurchführung in der Realität oft als schwierig. Aktivitäten[4], die nicht automatisiert bearbeitet werden können, müssen daher einem Endanwender *Prozessaktivitäten* in Form von Aufgaben in einer Arbeitsliste[5] zugeteilt werden. Anschließend kann dieser eine vorgeschlagene Aufgabe bzw. Aktivität aus seiner Arbeitsliste entnehmen, durchführen und so den Prozessablauf fortsetzen. Jedoch hat das System einen entscheidenden Mangel: aufgrund fehlender, kontextabhängiger Informationen z.B. Standort, ist es schwierig kontextbezogene Aktivitäten einer passenden Arbeitsliste vorzuschlagen. Auch sind neue Anpassungen an dynamischen Geschäftsabläufen für Mitarbeiter häufig aufwendig und komplex umzusetzen. Um zumindest die kontextabhängigen Probleme, während der aktiven Prozessausführung oder Aufgabenzuteilung

[1] Bring Your Own Device - Integration von privaten Mobilgeräten in Organisationen
[2] abk. PMTS
[3] abk. WFMTS
[4] Prozessaktivität, *en.* Activity
[5] *en.* Worklist

zu lösen, sprechen Gründe wie gerätespezifische Funktionen z.B. Kamera oder GPS-Funktionen, Mobilität unabhängig der Arbeitsumgebung, Flexibilität und die Möglichkeit kontextbezogene Informationen in Echtzeit abzurufen, für einen Wandel vom bisherigen Desktop-Arbeitsplätzen zu mobilen Endgeräten. Aufbauend auf der Idee von mobilen, kontextabhängigen Aktivitäten in Arbeitslisten und dem Konzept kontextbezogene Daten aus Mobilgeräten mit PMTS oder WFMTS zu kombinieren, beschäftigt sich diese Arbeit mit dem *Konzept und der Entwicklung eines Moduls für mobile Arbeitslisten*.

1.1 Motivation und Zielsetzung

Ziel dieser Bachelorarbeit ist ein einfaches Modul und eine mobile Arbeitsliste, mittels *Prototyping*, zu konzipieren und zu entwickeln. Dabei sollen ebenfalls die Probleme und Herausforderungen einer mobilen Anwendung mit Prozessaktivitäten thematisiert werden. Weiterhin werden in dieser Abschlussarbeit Ansätze und Konzepte angeführt, um das Gesamtsystem robuster und funktionaler zu gestalten.

Grundsätzlich zielt der folgende Inhalt dieser Arbeit auf zwei Implementierungsaspekte des Projekts ab: Client- und Server-Implementierung. Dabei sollen die Konzeption und die Entwicklungsschritte der Implementierungen dokumentiert werden. Statt eines Prozessmanagementsystems, wird die Komponente in dieser Arbeit durch eine REST-Schnittstelle und relationale Datenbank ersetzt und simuliert. Diese soll für die Datenhaltung, Verwaltung, Logik und Interaktion auf Arbeitslisten und Aktivitäten zuständig sein. Anschließend soll das Modul mit diesem Backend-System kommunizieren. Zudem berücksichtigt diese Arbeit den Einsatz von *Beacons BLE*[6], welche dem Modul eine ortsbezogene Kontextinformation darstellt und sich auf die dargestellten Inhalte der Worklists auswirken. Mit Hilfe von Beispielcodes werden die funktionale Umsetzungen der REST-Schnittstelle und des Moduls hervorgehoben und erklärt. Weitere Anforderungen für das Projekt werden in im Kapitel *Funktionale Anforderungen* ausgeführt.

[6] Beacon mit Bluetooth Low Energy

1.2 Überblick und Aufbau der Arbeit

Nach einer kurzen Einführung und Motivation für die Entwicklung des Projekts, folgt nun eine kurzer, inhaltlicher Gesamtüberblick der Arbeit. Im zweiten Kapitel werden die funktionalen Anforderungen der REST-Schnittstelle, relationalen Datenbank und des Moduls aufgelistet und vorgestellt. Auch werden die nicht-funktionalen Aspekte des Projekts erwähnt, welche weitere Gesichtspunkte des Prototyps abdecken sollen. Anschließend wird im nachfolgenden Kapitel die Projektstruktur und der Entwurf des Gesamtsystems abgebildet und das Datenbankmodell beschrieben. Anhand von Beispielcodes wird die Serverimplementierung im Unterkapitel 4.1 erklärt. Neben der Darstellung von RESTful Webservices in Java, umfasst das Kapitel auch REST-Endpoints und das Nachrichtenformat des Webservices. Danach werden die Implementierungsaspekte des Moduls in Kapitel 4.2 vorgestellt und auf die Herausforderungen mit Kontextinformationen eingegangen. Wie im vorherigen Kapitel werden auch hier exemplarisch einige bedeutende Codebeispiele ausgewählt und erklärt. Bedienung sowie die Vorstellung der Applikation werden im Kapitel 4.2.3 zusammengefasst und anschaulich dargestellt. Im darauffolgenden Teil der Arbeit werden auf die Projektanforderungen und Herausforderungen des Projekts eingegangen. Auch werden im selben Abschnitt die Ausnahmebehandlungen und fehlenden Komponenten des Prototyps thematisiert und weitere Implementierungsvorschläge und Referenzen eingeführt, die nicht in der Projektanforderungen umgesetzt werden konnten. Im Abschluss der Arbeit folgt eine kurze Zusammenfassung des Projekts und ein Ausblick auf die zukünftige Entwicklung von Applikationen mit Prozessunterstützung auf mobilen Endgeräten.

2 Projektanforderung

Im ersten Abschnitt dieses Kapitels werden die funktionalen Anforderungen des Projekts aufgeführt. Die Liste umfasst wichtige, serverseitige Anforderungen, welche die REST-Schnittstelle und die dazugehörige Datenbank erfüllen müssen, um vereinfachte Aktivitäten aus einem Prozessmanagementsystem zu simulieren. Hingegen erfassen clientseitige Anforderungen Funktionen, die das Modul besitzen sollte, damit eine Interaktion mit den Aktivitäten zwischen verschiedenen Worklists möglich ist. Für die Lauffähigkeit des Projekts sind nicht-funktionale Anforderungen nicht notwendig, können jedoch zur Benutzerfreundlichkeit und Bedienbarkeit des Projekts beitragen. Ziel des Kapitels ist, einen überschaubaren Gesamtüberblick auf die geforderten Funktionen und Implementierung des Webservices und der mobilen Applikation zu vermitteln. Auf den funktionalen Anforderungen aufbauend, welche sich an der Dissertation von Rüdiger Pryss [9, S. 25-27] orientiert, wird anschließend das Projekt konzipiert und entworfen.

2.1 Funktionale Anforderungen

Im Folgenden werden die funktionalen Anforderungen in zwei Abschnitte unterteilt. Der erste Punkt setzt sich mit den Spezifikationen der Serverimplementierung auseinander. Anschließend wird im zweiten Teil des Kapitels die Anforderungen des Moduls aufgeführt und zusätzliche Funktionen der Applikation spezifiziert.

2.1.1 Serverseitige funktionale Anforderungen

Anhand dieser Anforderungen sollen Softwarearchitektur, Logik und das Design der REST-Schnittstelle und Datenbank entworfen werden:

- Die serverseitige Implementierung enthält einen RESTful Webservice, welche eine Navigation und Interaktion auf Arbeitslisten, mittels HTTP-Operationen, ermöglicht.

- Dahingehend kann man zwischen zwei Arten von Arbeitslisten unterscheiden: *QueueActivities* und *MyActivities*
- emphQueueActivities verwaltet alle verfügbaren Aktivitäten. Sämtliche Aktivitäten in dieser Liste werden vom System in QueueActivities bereitgestellt. Je nach Kontextinformation, hat jeder Endanwender Zugriff auf die Aktivitäten dieser Liste. Aus QueueActivities können Benutzer die vorgeschlagenen Aktivitäten in ihre persönliche Arbeitsliste hinzufügen.
- Jeder Anwender verwaltet und besitzt eine eindeutig, zugewiesene Worklist MyActivities. Diese Arbeitsliste zeigt alle hinzugefügten Aktivitäten an, die derzeit vom entsprechenden Endanwender bearbeitet werden. Sobald der Benutzer eine Aktivität aus QueueActivities entnimmt, ist sie nur in MyActivities des entsprechenden Benutzers enthalten. Werden Aktivitäten in bestimmten Zuständen aus MyActivities gelöscht, müssen sie allen anderen Benutzern wieder in QueueActivities angezeigt und freigegeben werden.
- Kontextabhängige Variablen nehmen Einfluss auf die Zuteilung und Sichtbarkeit von bereitgestellten Aktivitäten in QueueActivities.
- Das Datenformat für Worklist- und Activity-Objekte soll in JSON umgesetzt werden.
- Generische Fehler und Ausnahmen werden mit einer einfachen Nachricht in JSON-Format an den Client geschickt. In diesem Prototyp werden prozessabhängige Fehlerbehandlungen nicht berücksichtigt, da hier kein PMTS eingesetzt wird. Jedoch wird im Kapitel 5.4 auf diesen Gesichtspunkt verwiesen, die dieses Thema ausführlicher behandelt.
- Das Lesen und Schreiben von Objekten *Worklist*, *Activity* und User wird mit CRUD-Operationen auf einer relationalen Datenbank umgesetzt.

2.1.2 Clientseitige funktionale Anforderungen

Die clientseitigen Anforderungen fassen die wichtigsten Funktionen zusammen, welche in der mobile Applikation bzw. Modul am Ende des Implementierung umsetzt werden:

- Listenansicht der Arbeitsliste QueueActivities soll alle verfügbaren Aktivitäten, unter der Berücksichtigung bereitgestellter Kontextinformationen, anzeigen.

- Listenansicht der Arbeitsliste MyActivities enthält eine Menge der ausgewählten Aktivitäten eines Benutzers.
- Kontextinformationen sollen die Verfügbarkeit der Aktivitäten in einer Arbeitsliste beeinflussen.
- Die Anwendung reagiert auf die Erkennung eines Beacons und zeigt 'versteckte' bzw. mobile Aktivitäten in der Worklist QueueActivities zusätzlich an. Es wird manuell nach Beacons gesucht und der Anwender bei Erfolg benachrichtigt.
- Der Anwender kann mit den Elementen seiner Arbeitsliste interagieren. Hierbei können Aktivitäten gelesen, freigegeben bzw. gelöscht oder verändert werden.
- Ein Wechsel der Sicht zwischen verschiedenen Arbeitslisten d.h. QueueActivities und MyActivities soll implementiert werden.

2.2 Nicht-funktionale Anforderungen

In diesem letzten Abschnitt dieses Kapitels werden zusätzliche Anforderungen erwähnt, die zur besseren Bedienung des Prototyps und einem übersichtlichen Design des Gesamtprojekts beitragen:

- Die automatische Fehlermeldungen vom Backend-System soll in diesem Prototyp an das Modul gesendet und angezeigt werden.
- Halbwegs robuste Ausführung der Applikation bzw. Moduls soll gewährleistet sein.
- Zur besseren Übersicht, unterstützt das Modul Pagination von Aktivitäten.
- Die Applikation verwendet eine einheitliche, graphische Benutzeroberfläche, z.B. Google Android Material Design.
- Weiterhin müssen Benachrichtigungen und Bedienung der Applikation erkennbar sein.
- Das Design des Prototyps soll überschaubar und einfach zu bedienen sein.
- Die App soll sich an Design Guidelines richten und dabei die optimale Layout- und Textgröße darstellen.

2.3 Zusammenfassung

Anhand der Anforderungen, erkennt man, dass der Entwurf einer mobilen Arbeitsliste erforderlich ist. Im weiteren Verlauf dieser Arbeit wird daher das Gesamtprojekt in zwei Bestandteilen zerlegt. Auf der Serverseite müssen RESTful Service und relationale Datenbank ein simples PMTS repräsentieren. Hierbei soll das Backend-System Arbeitslisten verwalten und Aktivitäten erstellen, lesen, modifizieren und löschen können. Die HTTP-Operationen POST, GET, PUT und DELETE werden unter den Anforderungen implementiert, sodass sie die CRUD-Operationen[1] abbilden. Mit Worklist- und Activity-Models sollen sowohl der Zugriff auf Daten, als auch die Darstellung in JSON-Datenformat erleichtert werden. Im zweiten Teil des Projekts muss ein Modul bzw. die mobile Applikation die Client-Anfragen umsetzen. Aufbauend auf dem JSON-Datenformat und die HTTP-Operationen der REST-Schnittstelle, liest das Modul die Arbeitslisten aus. Die Applikation soll mindestens zwei Typen von Listen anzeigen: QueueActivities und MyActivities[2] [9, S. 75]. Anschließend können aus den Worklists alle dazugehörigen Aktivitäten dargestellt werden. Die App soll ebenfalls die Möglichkeit haben mit den angezeigten Aktivitäten zu interagieren, indem man diese löschen, lesen oder modifizieren kann. Mit einem Beacon werden ortsbezogene Kontextinformationen simuliert, sodass spezielle Aktivitäten aus QueueActivities bei Erkennung dargestellt und ausgewählt werden können. Zusätzlich wurde, während der Implementierung, versucht auch auf die nicht-funktionalen Aspekte einzugehen und diese ansatzweise umzusetzen.

[1] Basisoperationen auf der Datenbank: Create, Read, Update, Delete
[2] Bezeichnung aus der Dissertation von Rüdiger Pryss

3 Architektur

Nachdem die Anforderungen für das Projekt ausgearbeitet wurden, widmet sich dieses Kapitel dem Aufbau und Entwurf des Projekts. Hierbei sollen die folgenden Abschnitte einen klaren Gesamtübersicht des Projekts liefern und die verwendete Software und wichtige Bibliotheken präsentieren.

3.1 Projektaufbau

Für die Ausführung eines Webservices wird ein Servlet-Container, in welche die REST-Schnittstelle ausgeführt wird, eingesetzt. Der mobile Client soll mithilfe der realisierten Android-Applikation mit dem RESTful Webservice per HTTP kommunizieren. Auf einer relationalen Datenbank, sollen anschließend Ressourcen und Daten gelesen und geschrieben werden. Die Abbildung 3.1 zeigt den Aufbau der Backend-Architektur als laufender Webserver und einen mobilen Client, welcher mit den Server über HTTP kommuniziert. Es folgt nun eine konzeptionelle Beschreibung zu den verwendeten Softwarekomponenten des Servers und des mobilen Clients. Außerdem werden hier die Mindestvoraussetzungen und eine Installationsbeschreibung der notwendigen Bibliotheken erwähnt, welche für die Entwicklung und Lauffähigkeit des Projekts notwendig sind.

3.1.1 Webserver für mobile Arbeitsliste

Für diese Arbeit wurde die Ausführungsumgebung *Apache Tomcat 8.0* [2] gewählt, da die prototypische REST-Schnittstelle in Java realisiert und der Apache Server leicht zu bedienen ist. Mithilfe der Bibliotheken JAX-RS 2.0[1] [3][6] und Jersey [4], wurden der Webservice implementiert. Grundsätzlich arbeitet die REST-Schnittstelle mit 3 verschiedenen Klassen: *Resource-*, *Service-* und *Object-Klassen*[2].

[1] Java API for RESTful Web Services

[2] im weiteren Verlauf der Arbeit auch: Model-Klassen

Damit man mit HTTP-Anfragen auf die URL-Ressourcen von Worklist-und Activity-Objekten zugegriffen werden kann, wurden Resource-Klassen implementiert, welche die adressierten Pfade realisieren. Service-Klassen werden bei erfolgreicher HTTP-Anfrage ausgeführt, dienen als Schnittstelle zwischen Service und Datenbank, und sollen u.a. Ressourcen auslesen bzw. schreiben. Zusätzlich liefern die Services entsprechende Responses an dem Client zurück. Die Bibliotheken *JAXB*[3] und *Jackson JSONParser* finden insbesondere in Object-Klassen Verwendung, welches erlaubt, Objekte automatisch in JSON-Format zu parsen. Ein weiterer Gesichtspunkt von Object-Klassen ist die Transformation der Datenbank-Entitäten in die entsprechenden Klassenobjekte und umgekehrt. Um jedoch Zugriff auf die Datenbank zu implementieren, wird deswegen die Java-Klasse DatabaseHandler als Schnittstelle eingeführt.

3.1.2 Mobiler Client - Eine Android-App für die Arbeitsliste

Das Modul bzw. die App wird für das Betriebssystem Android 5.0 implementiert. Dabei stellt das Modul automatisch eine Kommunikation zum Backend-System her und soll über einen Nachrichtenformat JSON kommunizieren. Aktivitätsklassen stellen die View und Navigation der Applikation dar. Auf Anfrage zeigt diese die entsprechenden Ressourcen an. Die Logik und Interaktion auf der View werden in den Adapter-Klassen realisiert. Einer der Adapter *BeaconAdapter*[4], soll dabei Beacons erkennen und dem Client entsprechend mitteilen. Diese sollen hier einen mobilen Kontext simulieren und eine Anfrage an den Server stellen, um ortsabhängige Aktivitäten zu laden.

3.1.3 Dokumentation - Voraussetzung und Entwicklungsumgebung

Voraussetzung für das Projekt ist eine Java Runtime Environment[5] und empfehlenswert eine Entwicklungsumgebung für die REST-Schnittstelle z.B. *Eclipse IDE for Java EE Developers*. Mit dem Build-Management-Tool Apache Maven werden Dependencies bzw. Bibliotheken des Projekts verwaltet. Möchte man ohne Apache Maven arbeiten, so ist es notwendig die Bibliothek Jersey separat in das angelegte

[3] Java Architecture for XML Binding
[4] In der Implementierung: BluetoothAdapter
[5] Mindestvoraussetzung: JRE 7

Projekt einzubinden. Erforderlich ist ebenfalls ein Webserver z.B. Apache Tomcat[6] oder Glassfish, welche die Ausführung von WAR-Dateien ermöglicht. In dieser Arbeit wurden Datenbanksysteme wie MySQL oder die Open Source-Variante PostgreSQL für die Speicherung, Lese- und Schreibzugriffe von Daten genutzt. Beide Datenbanksysteme bieten zudem eine graphische Benutzeroberfläche an, welche die Entwicklung und Verwaltung von Datenbankensystemen erleichtern. Für die Implementierung der mobilen App kommt in diesem Projekt die Entwicklungsumgebung *Android Studio*[7] zum Einsatz.

3.2 Datenbank

Im zweiten Unterkapitel wird das Datenbankmodell beschrieben, welche Worklist-, Activity-Objekte und Nutzerinformationen speichert. Anzumerken ist, dass die entsprechende Datenbanktabelle *User* in der Implementierung keine Funktionalität hat. Zur Vollständigkeit wird sie dennoch hinzugefügt und im späteren Kapitel 5 diskutiert. Ein entsprechendes EER-Diagramm in Abbildung 3.2 soll einen Gesamtüberblick der Datenbank-Entitäten liefern und die Relationen der Tabellen ausarbeiten. Man erkennt, dass das Schema des Datenbanksystems aus *worklist*, *activity* und *user* besteht.

3.2.1 Beschreibung

Anhand des EER-Diagramms werden im Folgenden die Datenbank-Entitäten beschrieben und erklärt. Zu Beginn wird die Tabelle, in welcher die Benutzerinformationen hinterlegt werden, erläutert. Obwohl diese Tabelle in der späteren Implementierung nicht genutzt wird, sollte sie darauf hinweisen, dass eine Implementierung von Benutzerkonten notwendig sind, um die Funktionalität von Arbeitslisten auszubauen.

Unter der Tabelle 3.1, findet man Informationen über die entsprechenden Benutzer der Arbeitsliste wieder, darunter der Benutzername *username*, das Benutzerkennwort *password*, die E-Mail-Adresse *email* und eine Benutzerrolle *roleid*. Die Idee ist, dass jeder Endanwender über einen Account verfügt, welche ihm seine persönliche Arbeitsliste zuweisen sollte. Gewöhnlicherweise registriert sich der Benutzer

[6] Mindestvoraussetzung: Apache Tomcat 8.0
[7] Android Studio 2.23

Attribut	Typ	Merkmal	Beschreibung
username	Varchar(20)	Primary Key	Name für ein Benutzerkonto; System verwaltet QueueActivities
password	Varchar(20)	Not null	Password des Benutzers
email	Varchar(32)	Unique, Not null	E-Mail-Adresse des Benutzers
roleid	Int	Not null	Zugewiesene Rolle des Benutzers

Tabelle 3.1: Tabelle user - Verwaltung von Benutzerkonten

mit einer E-Mail-Adresse und wählt einen eindeutigen, verfügbaren Benutzernamen und ein Kennwort. Wie bei einem Benutzerkonto sind Name und Kennwort für eine erfolgreiche Anmeldung notwendig. Die zugewiesene Benutzerrolle sollte dem Endanwender weitere Zugriffsrechte auf Aktivitäten einräumen. Ein Spezialfall eines Benutzers ist der Systemaccount mit dem Benutzernamen *System*, welche die Arbeitsliste *QueueActivities* verwaltet.

Attribut	Typ	Merkmal	Beschreibung
worklistid	Int	Primary Key	Identifikator einer Arbeitsliste; QueueActivities ist 0
accessstatus	Int	Not null	Zustand einer Arbeitsliste
description	Varchar(100)		Beschreibung einer Arbeitsliste
username	Varchar(20)	Foreign Key	Account der Arbeitsliste; Spezialfall für Account System

Tabelle 3.2: Tabelle worklist - Verwaltung von Arbeitslisten

Jeder registrierter Endanwender besitzt eine persönliche Arbeitsliste *MyActivities* mit einer ID *worklistid*, sodass sie auf dieser Weise eindeutig identifiziert werden kann. Einmalig ist die Arbeitsliste QueueActivities mit worklist 0, welche vom Benutzer *system* verwaltet wird. Weiterhin besitzt sie die Besonderheit, dass alle Endanwender die Aktivitäten der Arbeitsliste eingeschränkt auslesen und in ihre persönliche Liste MyActivities hinzufügen können. Neben der Identifikationsnummer erlaubt *accessstatus*, Arbeitslisten in spezielle Zustände u.a. aktiv, gesperrt oder in Wartung zu setzen. Bei Bedarf soll somit den Zugriff auf diese Arbeitslisten einschränkt werden. Letztlich können Endanwender in *description* eine Anmerkungen und Beschreibungen an ihrer Arbeitsliste hinzufügen. Die Tabelle 3.2 stellt eine Kurzübersicht der Datenbanktabelle *worklist* dar.

In der letzten genannten Tabelle 3.3 wird die Datenbank-Entität *activity* anschaulich präsentiert. Unter Verwendung vom Attribut *activityid* können Aktivitäten eindeutig identifiziert werden. Um der Aktivität einer Liste zuweisen zu können, setzt man

Attribut	Typ	Merkmal	Beschreibung
activityid	Int	Primary Key	Identifikator einer Aktivität
activityname	Varchar(45)	Not null	Name der Aktivität
type	TinyInt	Not null	Flag zur Unterscheidung zwischen Aktivitäten in der Liste QueueActivities oder MyActivities
roleid	Int	Not null	Zugriff auf die Aktivität in der entsprechenden Rolle; Keine Einschränkung im Default
description	Varchar(100)		Kurzbeschreibung einer Aktivität
duration	Int	Not null	Dauer einer auszuführenden Aktivität in Minuten
activitystatus	Int	Not null	Zustand einer Aktivität
batterystatus	Int	Not null	Erforderliche Akkuleistung zur Ausführung einer Aktivität
connectstatus	Int	Not null	Kontextinformation für die standortbasierten Aktivitäten
address	Varchar(45)		MAC-Adresse eines Senders; Wert verfügbar bei connectstatus
worklistid	Int	Foreign Key	Aktivität zugewiesene Arbeitliste

Tabelle 3.3: Tabelle worklist - Verwaltung von Arbeitslisten

die entsprechende Referenz auf das Attribut *worklistid* der entsprechenden Liste. Abhängig von der Arbeitsliste kann man zwischen zwei Arten von Aktivitäten, *type*, differenzieren. Befindet sich die Aktivität in QueueActivities, wird type auf 0 gesetzt. Wird die Aktivität ausgewählt und einer MyActivities zugeordnet, so setzt man den Flag auf 1. Bei Bedarf kann somit auf alle aktiven Aktivitäten aller MyActivities zugegriffen werden. Des Weiteren hat jede Aktivität einen Namen, sodass man diese in einer mobilen Arbeitsliste alphabetisch sortieren und suchen kann. Das Attribut *roleid* ermöglicht es, bestimmte Aktivitäten nur für die Zielgruppe von Endanwendern in der passenden Benutzerrolle darzustellen. Keine entsprechenden Einschränkungen hat die Aktivität, falls roleid dem Default-Wert 0 entspricht. Ein weiterer Parameter für die Prozessaktivität ist *duration*, welche die ungefähre Dauer der Aktivität bestimmtt. Zustände der ausgewählten Prozessaktivitäten werden unter *activitystatus* gespeichert. Dabei repräsentieren die Integer-Werte des Attributs einen Zustand der Aktivität z.B aktiv, pausiert, wartend, delegiert. Das zusätzliche Attribut *batterystatus* gibt die erforderliche Akkuleistung eines mobilen Endgeräts an, um die Aktivität auszuführen. Damit Aktivitäten auch ortsabhängige Kontextinformation beinhalten können, wird aus diesem Grund *connectstatus* eingeführt. Der Wert die-

ses Attributs entscheidet, ob Aktivitäten vom Ausführungsort des Moduls abhängig sind. Dementsprechend wird in diesem Projekt connectstatus für diesen Zweck implementiert, um einen ortsbezogenen Kontext einfügen zu können, welche nur über bestimmte Beacons zugänglich sind. Aufbauend auf den standortbasierten Aktivitäten, muss unterschieden werden, welche nur über bestimmte Beacons zu erreichen sind. Mit Hilfe der eindeutigen MAC-Adresse *address* können diese Aktivitäten bei Erkennung von Beacons mit passenden Adresse geladen und angezeigt werden.

3.2.2 Weitere Anmerkungen zur Datenbank

Aus der Beschreibung im vorherigen Abschnitt erschließt man, dass das Schema nur die wichtigsten Aspekte einer PMTS berücksichtigt und deshalb unvollständig ist. In diesem Projekt werden die Objekte fest in der Datenbank gespeichert und sollen den REST-Service beim Lesen und Schreiben von Ressourcen unterstützen. Zudem gilt, dass die Zugriffe weder effizient, noch alle Fehler- und Ausnahmefälle behandelt wurden. Realisiert werden lediglich einige Fälle, wie das Löschen von bestimmten *worklist*- und *user*-Einträgen. Ein Fallbeispiel für die Implementierung einer Datenbankfunktion, ist das Entfernen einer MyActivities-Arbeitsliste, gefüllt mit Aktivitäten. In diesem Fall sollen die Aktivitäten aus der besagten Liste entfernt und in QueueActivities wieder freigeben werden. Ein Ansatz, um diese Anforderung umzusetzen, ist die Einführung und Implementierung von Trigger-Funktionen. Es folgt nun ein Beispiel eines MySQL-Statements, welches den beschrieben Vorgang umsetzt:

```
CREATE TRIGGER worklistDelete_and_activityUpdate
BEFORE DELETE ON worklist
FOR EACH ROW
UPDATE activity AS newAct
SET newAct.worklistid = 0, newAct.type = 0
WHERE newAct.worklistid = old.worklistid;
```

Obwohl der Benutzer des Prototyps keine Arbeitsliste löschen kann, soll dieses Fallbeispiel deutlich machen, dass die Implementierung von Datenbankfunktionen in diesem Projekt notwendig sind, um den korrekten Umgang mit einer mobilen Arbeitsliste zu simulieren. Außerdem ist noch anzumerken, dass die Syntax, vom jeweils eingesetzten Datenbankmanagementsystem voneinander abweichen kann.

3.3 Zusammenfassung

Ziel dieses Kapitel ist es, eine Übersicht über die Struktur des Projekts zu erhalten. Neben dem Entwurf des Gesamtprojekts und ihre Abhängigkeiten, soll sich der erste Abschnitt einen kurzen Einblick in die verwendeten Technologien verschaffen. Zudem soll die Beschreibung des Datenbankmodells zeigen, welche Entitäten notwendig sind, um die wichtigsten Gesichtspunkte einer mobilen Arbeitsliste abbilden zu können. In erster Linie soll die Beschreibung dazu beitragen, ein grundlegendes Verständnis des Projekts zu vermitteln.

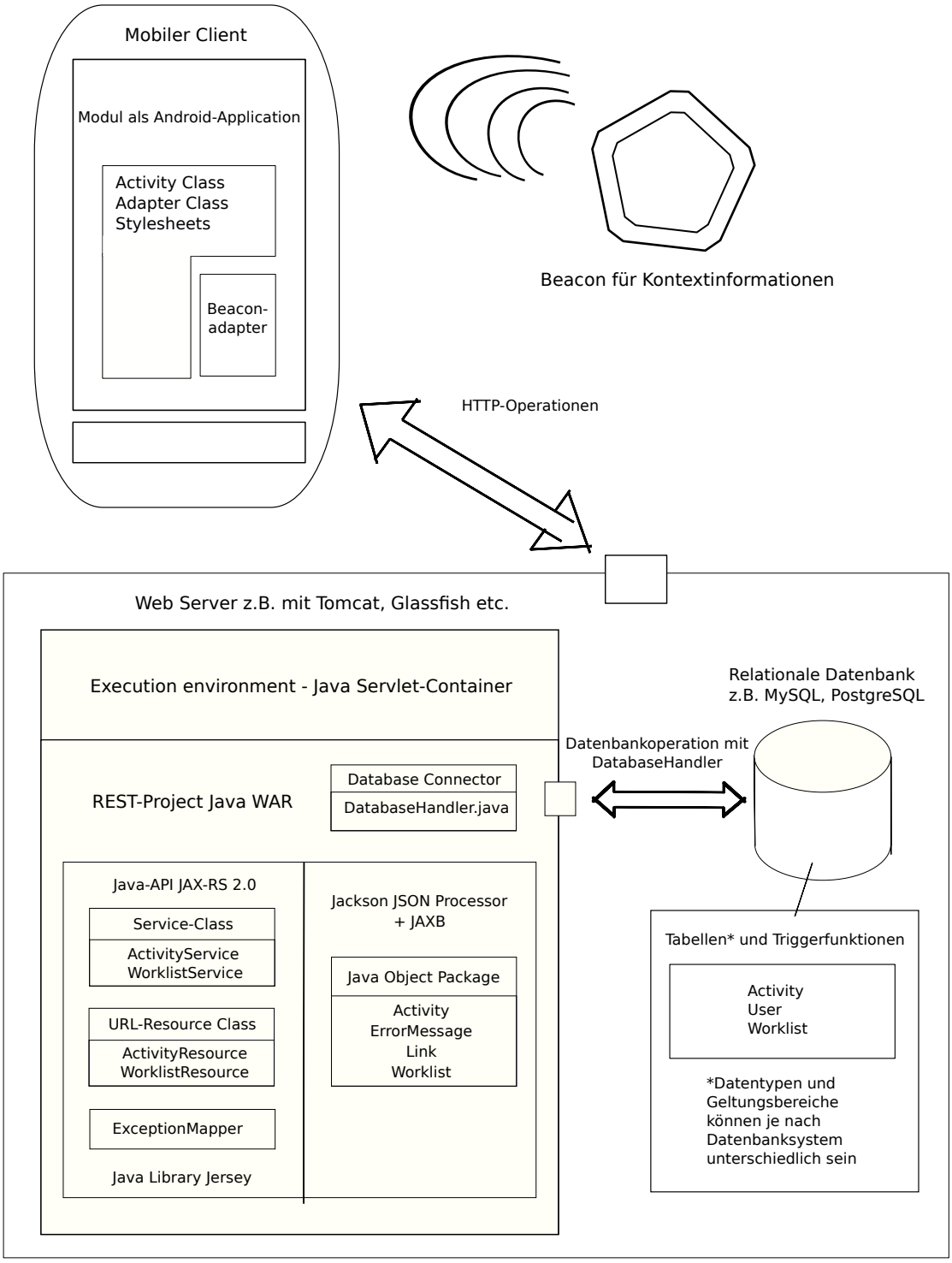

Abbildung 3.1: Architektur - Gesamtübersicht der verwendeten Komponenten

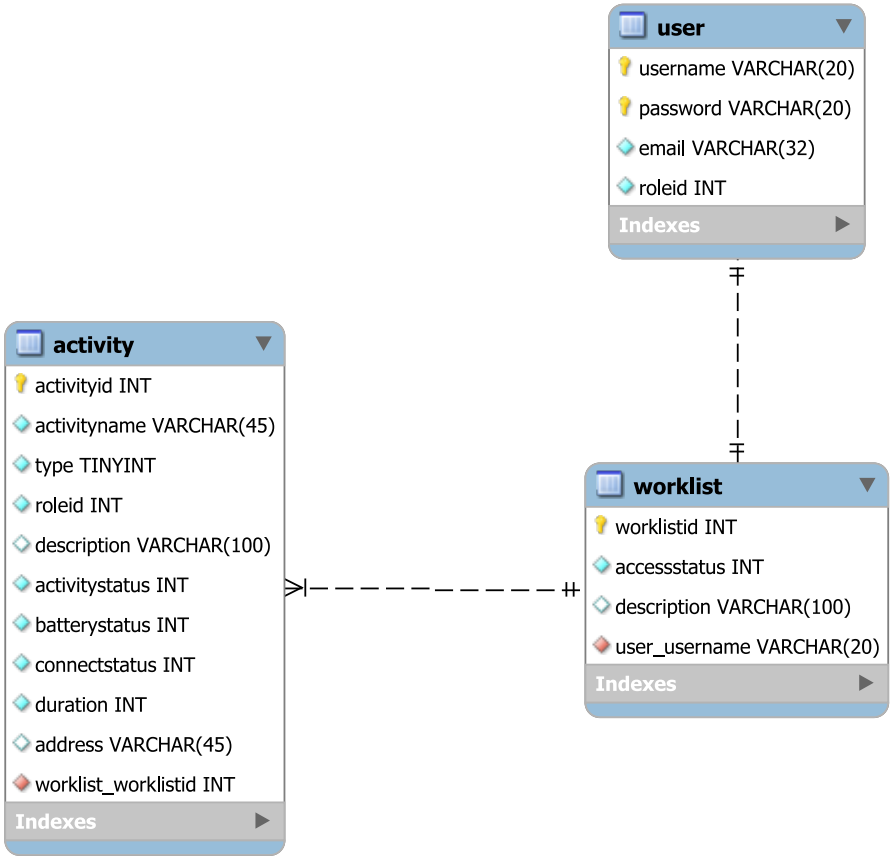

Abbildung 3.2: EER-Diagramm - Datenbankschema zur mobilen Arbeitsliste

4 Client- und Serverimplementierung

Nachdem der Aufbau des Projekts ausführlich beschrieben und die Struktur aufgezeigt wurden, setzt sich der Inhalt dieses Kapitel mit dem wichtigsten Aspekten der Client- und Serverimplementierung auseinander. Unter anderem werden hier ausgewählte Implementierungstechniken des Projekts dokumentiert und beschrieben. Aufbauend auf dem vorherigen Kapitel wird zunächst die serverseitige Implementierung betrachtet. Anschließend werden clientseitige Implementierungsaspekte im zweiten Teil dieses Kapitels dargestellt und erläutert.

4.1 RESTful Webservice in Java

In diesem Unterkapitel werden alle implementierten Packages, Klassen, Interfaces und Methoden des Java-Projekts aufgelistet und erklärt. Dabei beschränken sich die folgenden Beispiele der Dokumentation auf die wichtigsten Implementierungsaspekte der REST-Schnittstelle, realisiert mit JAX-RS 2.0 und Jersey.

4.1.1 de.ulm.uni.bachelor.nguyen.worklist_rest_api.database

Das Package verwaltet alle Datenbank-relevanten Klassen und Schnittstellen, die für eine Kommunikation zum Datenbankmanagementsystem benötigt werden. In diesem Projekt ist *DatabaseHandler* eine Java-Klasse, welche aus der Properties-Datei *db.properties* Treiber-und Datenbankinformationen liest und die Verbindung zur Datenbank herstellt. Ebenfalls werden in dieser Klasse die Methoden für Lese- und Schreibzugriffe auf die Datenbankeinträge realisiert. Der folgende Ausschnitt 4.1 aus der besagten Java-Klasse zeigt, wie mit der Methode *createConnection()* eine Verbindung zum Datenbankmanagementsystem hergestellt wird:

```
Properties properties = new Properties();

try {
       properties.load(Thread.currentThread().getContextClassLoader
          ().getResourceAsStream("db.properties"));

       Class.forName(properties.getProperty("jdbc.driver"));
       con = DriverManager.getConnection(properties.getProperty("
          jdbc.url"), properties.getProperty("jdbc.username"),
          properties.getProperty("jdbc.password"));
} catch (SQLException | IOException | ClassNotFoundException e) {
       e.printStackTrace();
       throw new DatabaseException("Database error");
}
return con;
```

Listing 4.1: Ausschnitt aus createConnection()

Es folgt nun eine Liste aller Methoden und eine kurze Beschreibung zur Funktionalität der Klasse Databasehandler:

- **Connection createConnection()** stellt eine Verbindung zum Datenbankmanagementsystem her.
- **Map<Integer, Worklist> getAllWorklists()** liest alle worklist-Einträge aus der Datenbank und liefert eine Key-Value-Liste[1] von Arbeitslisten zurück.
- **void addNewWorklist(Worklist worklist)** schreibt ein Worklist-Objekt in die Datenbank.
- **void updateWorklist(Worklist worklist)** aktualisiert[2] den entsprechenden Datenbankeintrag des übergebenen Worklist-Objekts.
- **void removeWorklist(int myWlId)** entfernt einen gültigen Worklist-Eintrag mit der angegebenen ID[3]. Implementierte Trigger in der Datenbank ermöglichen es den entsprechenden Besitzer zu löschen und die Aktivitäten in QueueActivities wieder freizugeben.

[1] Key = worklistId; Value = Worklist-Objekt
[2] Besitzer und ID können nicht verändert werden
[3] myWlId

- **Map<Integer, Activity> getAllActivitiesFromWlID(int myWlId)** liefert alle Aktivitäten der angegeben WorklistID in einer Key-Value-Liste[4] aus.
- **Map<Integer, Activity> getMyActivities(int wlId)** unterscheidet anhand der ID, ob Aktivitäten aus QueueActivities oder MyActivities gelesen werden.
- **void addActivity(Activity activity)** fügt ein Activity-Objekt der Datenbank hinzu. Sie wird nur erfolgreich ausgeführt, falls die Aktivität der Spezifikation der Datenbank *activity* entspricht.
- **void updateActivity(Activity activity)** überschreibt den Activity-Eintrag in der Datenbank. Attribut *activityid* kann hierbei nicht verändert werden.
- **void removeActivity(int wlId, int actId)** löscht den Subressource mit der entsprechenden *activityid* zur zugehörigen Arbeitsliste *worklistid.*
- **Map<Integer, Activity> getMyActivitiesWithBeacon(String myAddress)** liefert alle zugänglichen Aktivitäten aus QueueActivities inklusive die Kontextabhängigen mit der entsprechenden MAC-Adresse.
- **boolean checkIfExists(String table, String column, int id)** ist eine Hilfsfunktion und überprüft, ob ein entsprechender Eintrag[5] in der angegebenen Datenbanktabelle[6] existiert.
- **boolean checkUsernameExists(String username)** überprüft einen Benutzernamen.
- **boolean checkWlIdExists(int myId)** untersucht, ob die übergebene WorklistID bereits existiert.
- **boolean checkActIdExists(int myId)**, wie in den vorherigen beiden Punkten, überprüft stattdessen die ID einer Aktivität.

4.1.2 de.ulm.uni.bachelor.nguyen.worklist_rest_api.model

In diesem Unterabschnitt werden die Models Activity, ErrorMessage, Link und Worklist beschrieben. Insbesondere spielt die Package für JAXB eine entscheidende Rolle, denn die Klassenobjekte werden hier automatisch mit dem Jackson JSON Parser in den Datenformat JSON umgewandelt. Da der Aufbau in allen Klassen

[4] Key = activityid, Value = Activity-Objekt
[5] column und id
[6] table

fast identisch ist, zeigt ein Snipplet 4.2 aus der Klasse Worklist, wie JAXB realisiert wird. Ein wichtiges Schlüsselelement ist die Annotation *@XMLRootElement*. Damit kann ein Root-Element einer Klasse eindeutig assoziiert werden. Kombiniert man dies mit Jackson, so wird statt einem XML-Objekt ein Objekt in JSON gebildet. Für Jackson ist es zusätzlich erforderlich, dass eine leere Instanz angelegt wird.

```
@XmlRootElement
public class Worklist {

        private int worklistid;
        private String username;
        private int accessstatus;
        private String description;
        private Map<Integer, Activity> activities = new HashMap<>();
        private List<Link> links = new ArrayList<>();

        public Worklist() {

        }

        public Worklist(int worklistid, String username, int
            accessstatus, String description) {
        this.worklistid = worklistid;
        this.username = username;
        this.accessstatus = accessstatus;
        this.description = description;
        }
```

Listing 4.2: Ausschnitt aus der Klasse Worklist

Ähnlich wie im vorherigen Abschnitt werden nun die Strukturen und Besonderheiten der einzelnen Klassen beschrieben und aufgelistet, dagegen werden hier die generischen Getter- und Setter-Methoden ignoriert:

Worklist Neben den Attributen in Tabelle 3.2 , welche bereits in Kapitel *Datenbank* angesprochen wurde, enthält ein Worklist-Objekt Aktivitäten und Links.

- **Map<Integer, Activity> activities** zeigt eine Liste von zugeordneten Aktivitäten an.

- **List<Link> links** verwaltet Links[7], die eine Navigation durch die Worklist ermöglicht.
- **void addLink(String url, String rel)** setzt neue Link-Objekte in die Liste ein.

Activity Activity-Objekte enthalten alle Attribute aus der Tabelle 3.3 und zusätzlich ein Integer-Wert *priority*, welche die Priorität[8] einer Aktivität aus *batterystatus*, *duration* und *connectstatus* berechnet.

Links Links sind Objekte, die jeweils zwei String-Objekte, *link* und *rel*, für die URL halten können.

ErrorMessage ErrorMessage-Objekte werden erzeugt sobald ein Fehler auftritt, diese geben einen Fehlercode *errorCode* und eine Fehlernachricht *errorMessage* aus.

4.1.3 de.ulm.uni.bachelor.nguyen.worklist_rest_api.resource

Dieser Abschnitt der Arbeit beschreibt die realisierten REST Resource-Klassen mit speziellen JAX-RS/Jersey-Annotationen. Es erlaubt, auf HTTP-Anfragen an den entsprechenden URIs der REST-Schnittstelle, nach Ressourcen zu suchen oder diese zu bearbeiten. Die entsprechenden URI sind so in den Resource-Klassen konzipiert, sodass Aktivitäten als Subressourcen von Arbeitslisten auftreten. Zunächst wird die Resource-Klasse von Worklist *WorklistResource* beschrieben.

```
@Path("/worklists")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class WorklistResource {
...
}
```

Listing 4.3: Ausschnitt aus WorklistResource - Jersey Annotationen

[7] siehe HATEOAS
[8] Priorität wird im Modul nicht implementiert

WorklistResource Um die URL-Ressourcen festzulegen und HTTP-Requests zu behandeln, werden Jersey und JAX-RS Annotationen eingesetzt. Beispielcode 4.3 zeigt verschiedene Annotationen, die zu Beginn der Klasse implementiert werden: @Path, @Consume und @Produce.

- **@Path** definiert einen REST-Endpoint, auf dieser die Ressource als relative URI zu erreichen ist. Als allgemeiner Endpunkt für Worklist-Ressourcen wird */worklists* gewählt. Im nächsten Unterabschnitt 4.1.6 werden die Endpoints und Nachrichtenformate genauer betrachtet.
- **@Consumes** verarbeitet Anfragen nur, falls diese im angegebenen MIME-Type umgesetzt sind.
- **@Produces** sendet HTTP-Responses im angegebenen MIME-Type an den Client. Es ist ebenfalls möglich mehrere Repräsentationen von MIME-Types für eine Response anzubieten.

```
@GET
@Path("/{worklistid}")
public Worklist getWorklist(@PathParam("worklistid") int
   worklistid, @Context UriInfo uriInfo) {
       Worklist worklist = worklistService.getWorklist(
          worklistid);
       worklist.addLink(getUriForSelf(uriInfo, worklist), "
          self");
       worklist.addLink(getUriForActivities(uriInfo,
          worklist), "activities");
       return worklist;
}
```

Listing 4.4: Ausschnitt aus WorklistResource - GET /worklists/worklistid

- **@GET, @POST, @PUT, @DELETE** sind vordefinierte Annotationen, die auf entsprechende HTTP-Operationen reagieren und die darunterliegenden Methoden ausführen.
- **@PathParam** ermittelt annotierte Parameter aus der URL und liest diese als Funktionsparameter ein. Beispielsweise wird hier *worklistid* gelesen und als String-Parameter weiterverarbeitet.
- **@Context** können Methoden- und Parameterkontext aus HTTP-Anfragen gelesen werden. Ein Anwendungsfall für @Context zeigt der Ausschnitt 4.4, in-

dem die relative URI ermittelt und in ein Links-Objekt verarbeitet wird.

Ebenfalls sind die Annotationen in der Resource-Klasse *ActivityResource* vorhanden. Um die Subressource zu definieren, wird sie in der Klasse explizit definiert; wie im folgenden Codeausschnitt 4.5 beschrieben:

```
@Path("/{worklistid}/activities")
public ActivityResource getActivityResource() {
       return new ActivityResource();
}
```

Listing 4.5: Ausschnitt aus WorklistResource - Subressource Activity

Hinzu folgt nun eine Gesamtübersicht aller implementierten Resource-Methoden der Klasse *WorklistResource*:

- **@GET List<Worklist> getAllWorklist()** soll auf HTTP-GET in */worklists/ alle Arbeitslisten in einem JSON-Array auflisten.
- **@POST Response addWorklist(Worklist worklist, @Context UriInfo uriInfo)** legt mit POST */worklists/ eine neue Arbeitsliste an. Der Client wird mit einer Response in JSON über den Vorgang benachrichtigt.
- **@PUT Worklist updateWorklist(@PathParam(“worklistid“) int wlid, Worklist worklist)** ändert die Arbeitsliste mit HTTP-PUT und sendet die aktualisierte Liste in JSON an den Client zurück.
- **@DELETE Response deleteWorklist(@PathParam(“worklistid“) int wlid)** löscht mit HTTP-DELETE auf */worklists/<wlid>/ die Arbeitsliste mit der ID *wlid*.
- **@GET Worklist getWorklist(@PathParam(“worklistid“) int wlid, @Context UriInfo uriInfo)** holt sich nur die Arbeitsliste *wlid* mittels HTTP-GET und liefert diese als JSON-Objekt aus.
- **String getUriForActivities(UriInfo uriInfo, Worklist worklist)** baut einen relativen URI-Pfad zur Subressource auf.
- **String getUriForSelf(UriInfo uriInfo, Worklist worklist)** erstellt einen Link zur Eigenreferenz.
- **ActivityResource getActivityResource()** greift auf die Subressourcen der Arbeitsliste *wlid* zu. Abschließend soll *ActivityResource* alle Anfragen auf dieser Worklist behandeln.

ActivityResource Die Aktivitäten in den Listen können unter der URI **/worklists/<wlid>/activities* erreicht werden. Identisch zur *WorklistResource* werden hier die HTTP-Anfragen auf Aktivitäten der Arbeitslisten umgesetzt. Deshalb werden nun die implementierten Methoden von *ActivityResource* aufgelistet:

- **List<Activity> getMyActivities(@PathParam(“worklistid“) int wlid, @QueryParam(“start“) int start, @QueryParam(“size“) int size, @QueryParam(“address“) String address)** liefert alle Aktivitäten der übergebenen Listen-ID als JSON-Array aus. Die Annotationen *@QueryParam* sollen die QueryParameter des Requests auslesen und eine Pagination darstellen. Gibt man stattdessen die *address* an, so wird zusätzlich nach Aktivitäten mit der besagten MAC-Adresse gesucht.
- **@POST Activity addActivity(@PathParam(“worklistid“) int wlid, Activity activity)** legt eine neues Activity-Objekt in einer Arbeitsliste an. In diesem Projekt wird nicht unterschieden, in welcher Arbeitsliste die Aktivität angelegt wird, da der mobile Endanwender diese Funktion nicht ausführt.
- **@PUT Activity updateActivity(@PathParam(“worklistid“) int wlid, @PathParam(“activityid“) int activityid, Activity activity)** aktualisiert die Attributwerte der Aktivität und liefert dem Client das veränderte Aktivity-Objekt im JSON-Format aus.
- **@DELETE Response deleteActivity(@PathParam(“worklistid“) int wlid, @PathParam(“activityid“) int activityid)** soll den Löschvorgang auf eine Aktivität ausführen und den Client benachrichtigen.
- **@GET Response getActivity(@PathParam(“worklistid“) int wlid, @PathParam(“activityid“) int activityid)** selektiert eine bestimmte Aktivität aus einer Arbeitsliste und antwortet dem Client mit de, Activity-Objekt.

4.1.4 de.ulm.uni.bachelor.nguyen.worklist_rest_api.service

Die Aufgabe der Services ist die Verarbeitung und Wiedergabe von Datenbank-Einträgen. Des Weiteren wird hier die Logik der HTTP-Operationen auf Ressourcen implementiert. Die Service-Klassen sollen bei der zukünftigen Implementierung Fehler und Ausnahmen behandeln und somit das System robuster gestalten. Dementsprechend werden nun die Implementierungsaspekte der Service-Klassen dargestellt und erläutert. Der erste Unterabschnitt setzt sich mit der Klasse *Work-*

listService auseinander. Anschließend wird *ActivityService* im zweiten Teil dieses Unterkapitels dargestellt.

WorklistService soll eine Schnittstelle vom REST-Service zur Datenbank repräsentieren und alle Operationen, die Worklist-Objekte betreffen, ausführen. Neben der Logik werden hier die Fehler und Ausnahmen behandelt. Bei einem Fehler- oder Ausnahmefall wird der Client mit einer entsprechenden Nachricht in Kenntnis gesetzt. Der nächste Unterkapitel 4.1.5 wird diesen Gesichtspunkt detaillierter betrachten. Welche Funktionen in dieser Klasse umgesetzt werden, wird nun im folgenden Absatz beschrieben:

- **List<Worklist> getAllWorklists()** gibt wird eine Liste von Worklist-Objekten aus. Zudem gilt, dass beim Starten des Service, alle Arbeitslisten einmalig aus der Datenbank ausgelesen werden.
- **Worklist getWorklist(int id)** sucht nach der Arbeitsliste mit der entsprechenden ID. Wird diese nicht gefunden, wird eine Fehlernachricht ausgeworfen.
- **Worklist addWorklist(Worklist worklist)** schreibt eine neue Worklist in die Datenbank. Dabei wird beachtet, ob die ID oder Nutzer bereits existiert, und ob dem angegebenen Benutzer schon eine Worklist zugewiesen wurde. Der Ausschnitt 4.6 zeigt wie die Methode realisiert wurde.
- **Worklist updateWorklist(Worklist worklist)** überprüft die ID der Liste und aktualisiert diese in der Datenbank.
- **Response removeWorklist(int id)** löscht die Arbeitsliste, auch wenn diese nicht mehr existiert und liefert dem Client eine entsprechende Response auf den Löschvorgang.

```
public Worklist addWorklist(Worklist worklist) {

        if (worklists.containsKey(worklist.getWorklistid())) {
                throw new WorklistExistsException("ID already exists
                    - please choose a new ID");
        }
        if (!DatabaseHandler.checkUsernameExists(worklist.
            getUsername()) || worklist.getUsername().equals(null)) {
                throw new CannotProcessValueException("Cannot process
                     value: Username");
        }
```

```
        for (Entry<Integer, Worklist> i : worklists.entrySet()) {
                Worklist w = i.getValue();
                if (w.getUsername().equals(worklist.getUsername())) {
                        throw new CannotProcessValueException("User
                            already has a worklist");
                }
        }
        DatabaseHandler.addNewWorklist(worklist);
        worklists.put(worklist.getWorklistid(), worklist);
        return worklist;
}
```

Listing 4.6: Ausschnitt aus WorklistService - Methode addWorklist(Worklist worklist)

ActivityService führt Methoden aus, die von der Resource-Klasse *ActivityResource* ausgelöst werden. Statt Arbeitslisten wird hier die Logik und Ausnahmebehandlung auf Activity-Objekte implementiert:

- **ArrayList<Activity> getMyActivities(int worklistid)** ermittelt die Aktivitäten der Arbeitsliste *worklistid* und speichert das Resultat in einem Array.
- **Response getActivity(int worklistid, int activityid)** überprüft, ob die besagte Aktivität existiert und liest ggf. ein Activity-Objekt aus der Arbeitsliste aus.
- **Activity addActivity(int worklistid, Activity activity)** fügt bei Erfolg einer bestimmten Arbeitsliste ein neues Activity-Objekt hinzu.
- **Activity updateActivity(int worklistid, Activity activity)** aktualisiert die Aktivität der entsprechenden Arbeitsliste.
- **Response deleteActivity(int worklistid, int activityid)** löscht die Aktivität und benachrichtigt dem Client mit einer entsprechenden Response.
- **List<Activity> getAllActivitiesPaginated(int start, int size, int worklistid)** gibt eine Liste von Activity-Objekten seitenweise aus. Dabei müssen Startindex und Größe der Seite angegeben werden.
- **List<Activity> getAllActivitiesWithBeacon(String address)** zeigt zusätzlich die Aktivitäten mit der MAC-Addresse *address* von QueueActivities an.
- **void checkNewActivity(Activity act)** überprüft und validiert die neue Aktivität und wirft bei einem Fehler eine Nachricht aus.

4.1.5 de.ulm.uni.bachelor.nguyen.worklist_rest_api.exception.*

Die Packages können mehrere individuelle Fehlernachrichten umsetzen. Da jedoch die Nachrichten auf der Model-Klasse *ErrorMessage* aufbauen, wird sich dieser Abschnitt mit einem allgemeinen Format der implementierten Fehlerbehandlungen beschäftigen. Grundsätzlich besteht eine individuelle *Exception* aus zwei Komponenten: *Exception* und *ExceptionMapper*.

```
public class SampleException extends RuntimeException {

	private static final long serialVersionUID = 1L;

	public SampleException(String message) {
		super(message);
	}
}
```

Listing 4.7: Beispiel einer Exception-Klasse

Jeder der implementierten Exception-Klassen erbt von der Superklasse *RuntimeException*. Der oben-gezeigte Beispielcode 4.7 zeigt die generische Struktur einer Exception-Klasse. ExceptionMapper-Klassen erweitern Ausnahmebehandlung mit individuellen Exception-Klassen und können somit auf die Ausführungsumgebung von JAX-RS während Laufzeit Einfluss nehmen. Um dies zu realisieren, wird *@Provider* eingesetzt. Wie die ExceptionMapper in diesem Projekt umgesetzt und die Fehlernachricht an in JSON geparst werden, zeigt das folgende Beispiel 4.8:

```
@Provider
public class SampleExceptionMapper implements ExceptionMapper<
   ExceptionException> {

	@Override
	public Response toResponse(SampleException ex) {
		ErrorMessage errorMessage = new ErrorMessage("An
		   individual error message", ex.getMessage());
		return Response.status(Status.OK).entity(errorMessage
		   ).type(MediaType.APPLICATION_JSON).build();
	}
}
```

Listing 4.8: Beispiel einer ExceptionMapper-Klasse

4.1.6 REST Endpoints und Nachrichtenformat

Nach der Beschreibung aller wichtigen, serverseitigen Implementierungsaspekte des RESTful Webservice-Projekts, widmet sich nun dieser Abschnitt dem REST-Endpoints und dem Nachrichtenformat der Schnittstelle. Aufbauend auf den vorherigen Punkten, werden in diesem Kommunikation zwischen Client und der Serverschnittstelle aufgezeigt. JSON wird hier dabei als Nachrichtenformat genutzt, um den Kommunikation kompakter gestalten und von den Funktionen des Jackson JSON Parsers und JAXB profitieren zu können. Es folgt eine Gesamtübersicht der Endpunkte und das notwendige Nachrichtenformat auf HTTP-Operationen:

GET */worklists Die folgende Response ist die Ausgabe eines JSON Arrays aller Arbeitslisten mit HTTP 200 OK:

```
[
{
        "worklistid": 0,
        "username": "system",
        "accessstatus": 0,
        "description": "QueueActivities - Owner: system",
        "links": []
},
{
        "worklistid": 2,
        "username": "nguyen",
        "accessstatus": 2,
        "description": "MyActivities - Owner: nguyen",
        "links": []
}
]
```

GET */worklists Die folgende Response ist die Ausgabe eines JSON Arrays aller Arbeitslisten mit HTTP 200 OK:

```
[
{
        "worklistid": 0,
        "username": "system",
        "accessstatus": 0,
        "description": "QueueActivities - Owner: system",
        "links": []
},
{
        "worklistid": 2,
        "username": "nguyen",
        "accessstatus": 2,
        "description": "MyActivities - Owner: nguyen",
        "links": []
}
]
```

POST */worklists Möchte man eine neue Arbeitsliste anlegen, so sollte POST-Body das folgende Format besitzen:

```
{
        "worklistid": 4,
        "username": "kuloglu",
        "accessstatus": 0,
        "description": "MyActivities - Owner: kuloglu"
}
```

Falls die Operation erfolgreich durchgeführt wird, erhält man anschließend die Response mit HTTP 201 Created:

```
{
  "worklistid": 4,
  "username": "kuloglu",
  "accessstatus": 0,
  "description": "MyActivities - Owner: kuloglu",
  "links": []
}
```

Bei fehlerhaften Eingaben wird eine entsprechende Fehlernachricht verschickt z.B. vorhandene ID einer Worklist:

```
{
        "errorCode": "Unexpected error",
        "errorMessage": "ID already exists - please choose a new ID"
}
```

PUT */worklists/<worklistid> Änderungen der Worklist mit ID *<worklistid>* werden hier durchgeführt. Das untere Beispiel zeigt einen Request im PUT-Body, um die Beschreibung der Arbeitsliste auf **/worklists/4* zu verändern:

```
{
        "username":"kuloglu",
        "accessstatus": 0,
        "description": "MyActivities - Owner: kuloglu - Updated"
}
```

Sowie die entsprechende Response mit den aktualisierten Informationen, HTTP 200 OK:

```
{
        "worklistid": 4,
        "username": "kuloglu",
        "accessstatus": 0,
        "description": "MyActivities - Owner: kuloglu - Updated",
        "links": []
}
```

DELETE */worklists/<worklistid> Um eine Arbeitsliste zu löschen, wird die DELETE Operation auf den entsprechenden Pfad der Liste angewendet und man erhält als Response HTTP 202 Accepted.

GET */worklists/<worklistid> Mit HTTP GET unter diesem REST-Endpoint erhält man die gewählte Arbeitsliste mit weiteren relativen Links als JSON-Objekt.

```
{
        "worklistid": 2,
```

```
        "username": "nguyen",
        "accessstatus": 2,
        "description": "MyActivities - Owner: nguyen",
        "links": [
        {
                "link": "http://localhost:8080/worklist-rest-api/
                   worklists/2",
                "rel": "self"
        },
        {
                "link": "http://localhost:8080/worklist-rest-api/
                   worklists/2/activities/",
                "rel": "activities"
        }
        ]
}
```


GET */worklists/<worklistid>/activities Der Client erhält ein JSON-Array aller Aktiväten der Liste *worklistid*. Falls man eine URL mit Pagination aufrufen möchte z.B. **/worklists/2/activities?start=1&size=2*, so hat die Response den folgenden Format:


```
[
{
        "activityid": 8,
        "worklistid": 2,
        "activityname": "Bestandskontrolle",
        "type": true,
        "roleid": 1,
        "description": "Inventur im Lagerabschnitt 03",
        "activitystatus": 2,
        "connectstatus": 0,
        "batterystatus": 20,
        "duration": 60,
        "address": "",
        "priority": 80
},
{
```

```
        "activityid": 9,
        ...
        "priority": 80
}
]
```

POST */worklists/<worklistid>/activities Eine Aktivität wird in dieser Liste angelegt. Wie das Format der Nachricht im Body erfolgen muss, wird mit diesem Beispiel unter **/worklists/2/activities* dargestellt:

```
{
        "activityid": 14,
        "activityname": "Sonderlieferung - Produkt H",
        "type": false,
        "roleid": 1,
        "description": "Nachlieferung soll bis zum 20.04 erfolgen!",
        "activitystatus": 0,
        "connectstatus": 1,
        "batterystatus": 50,
        "duration": 100,
        "address": "1F.D1:3f:52:3D:A9"
}
```

Anschließend wird der Client bei Erfolg, mit der neuen Aktivität benachrichtigt:

```
{
        "worklistid": 0,
        "activityname": "Sonderlieferung - Produkt H - Ersatz
           gesucht!",
        "type": false,
        "roleid": 1,
        "description": "Nachlieferung soll bis zum 20.04 erfolgen!",
        "activitystatus": 0,
        "connectstatus": 1,
        "batterystatus": 50,
        "duration": 100,
        "address": "1F.D1:3f:52:3D:A9"
        "priority": 300
}
```

PUT */worklists/<worklistid>/activities/<activityid> Die Aktivität *activityid* wird aktualisiert. Damit die Aktualisierung im richtigen Format durchführt werden kann, soll das folgende Beispiel unter **/worklists/2/activities/14* veranschaulichen:

```
{
        "activityid": 14,
        "activityname": "Sonderlieferung - Produkt H",
        "type": false,
        "roleid": 1,
        "description": "Nachlieferung soll bis zum 20.04 erfolgen!",
        "activitystatus": 0,
        "connectstatus": 1,
        "batterystatus": 50,
        "duration": 100,
        "address": "1F.D1:3f:52:3D:A9"
}
```

Wird die Änderung durchgeführt erhält der Client in der Response die neue Aktivität in JSON:

```
{
        "activityid": 14,
        "worklistid": 0,
        "activityname": "Sonderlieferung - Produkt H - Ersatz
           gesucht!",
        "type": false,
        "roleid": 1,
        "description": "Nachlieferung soll bis zum 20.04 erfolgen!",
        "activitystatus": 0,
        "connectstatus": 1,
        "batterystatus": 50,
        "duration": 100,
        "address": "1F.D1:3f:52:3D:A9",
        "priority": 300
}
```

DELETE */worklists/<worklistid>/activities/<activityid> löscht auf Anfrage die Aktivität in der Liste und liefert bei Erfolg HTTP 202 Accepted.

GET */worklists/<worklistid>/activities/<activityid> Ähnlich zur Worklist kann hier das Activity-Objekt der entsprechenden Arbeitsliste in JSON angezeigt werden. Die folgende Response stammt aus der GET-Anfrage auf **/worklists/0/activities/1*:


```
{
        "activityid": 1,
        "worklistid": 0,
        "activityname": "Rohstoffeinkauf - Produkt B",
        "type": false,
        "roleid": 0,
        "description": "Nachbestellung des Produkts B bis 29.03",
        "activitystatus": 0,
        "connectstatus": 0,
        "batterystatus": 0,
        "duration": 10,
        "address": "",
        "priority": 10
}
```


4.2 Client - WorklistApp in Android

Nachdem die wichtigsten Punkte der serverseitigen Implementierung ausgeführt und dokumentiert wurden, liegt nun der Schwerpunkt im zweiten Teil das Kapitels auf die wichtigsten Implementierungsaspekte des Moduls. Zur Demonstration einiger Operationen mit dem REST-Webservice, beschränkt sich der Prototyp der Android-Applikation auf insgesamt 2 Aktivitäten: der Hauptaktivität *MainActivity* und einer Nebenaktivität *MyWorklistActivity*. Dazu muss angemerkt werden, dass das Ziel des Prototyps nicht die Implementierung aller Operationen auf des REST Services ist. Die Implementierungen sollen lediglich einige kontextabhängige Informationen des mobilen Clients berücksichtigen, welche sich auf der Arbeitsliste QueueActivities widerspiegelt. Insbesondere liegt der Fokus dieser Arbeit auf die prototypische Erkennung und den Umgang von registrierten Beacons auf den Arbeitslisten. Während der Entwicklung und den Testvorgängen wurde ein Gimbal Qualcomm Beacon, welches BLE umsetzt, genutzt. Als Testgerät diente dazu ein Sony Xperia Z C6603 mit dem Android-Betriebssystem Lollipop 5.1.1. Da jedoch die Technologie BLE ab der Android Version Jelly Bean 4.3 verfügbar ist, sollte ein mobiles Endgerät mindestens diese Version unterstützen können.

4.2.1 MainActivity

Weil sich die Hauptaktivität und die Nebenaktivität der App gering unterscheiden, liegt Schwerpunkt der gezeigten Implementierungsaspekte auf der Klasse *MainActivity*. Dem Benutzer wird zum Start der Applikation diese Aktivität gezeigt. Hierbei beschränkt sich das Modul zunächst auf 4 wichtige, graphische Bestandteile: eine *ListView* und 3 *Buttons* für die Interaktionen. Mit einem ArrayAdapter werden die darunterliegenden Activity-Daten auf der ListView *queueActivityList* präsentiert. Für die Implementierung der Listenelemente wird vorerst String als Datentyp verwendet. Eine bessere Alternative für die Umsetzung der erhaltenen Activity-Objekte wäre die individuelle Darstellung der Listenelemente, geparst aus JSON-Objekten. Zusätzlich zur Präsentation der Aktivitäten können die Elemente durch einen simplen Klick selektiert und bearbeitet werden, wie im folgenden Beispielcode dargestellt:

```
queueActivityList.setOnItemClickListener(new AdapterView.
   OnItemClickListener() {
       @Override
       public void onItemClick(AdapterView<?> adapterView, View
          view, int position, long l) {
              Object obj = queueActivityList.getItemAtPosition(
                 position);
              String description = (String) obj;
              String[] parts = description.split("\\:");
              String id = parts[0];
              for(int k = 0; k < queueActJson.size(); k++) {
                     JSONObject check = queueActJson.get(k);
                     try {
                            if(check.getString("activityid").
                               equals(id)) {
                                result = check;
                            }
                     } catch (JSONException e) {
                     e.printStackTrace();
              }
       }
       new ActivityToMyActivities().execute(id);
       listAdapter.remove(listAdapter.getItem(position));
}
```

```
});
```

Listing 4.9: Ausschnitt aus MainActivity - OnItemClick auf die Listenelemente

Neben der Liste ermöglicht das Modul, mittels der Button *btnAct* und *btnActBlue*, die angezeigte Liste verschieden darzustellen. Mit *btnAct* sollen die allgemeinen, kontextunabhängigen Aktivitäten in die Liste geladen werden:

```
btnAct.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
                queueActivities = new ArrayList<String>();
                queueActJson = new ArrayList<JSONObject>();
                new GetQueueActivities().execute(actLink);
        }
});
```

Listing 4.10: Ausschnitt aus MainActivity - OnClick auf Button 'Scan for tasks'

Im ArrayList<String> *queueActivities* werden die String-Listenelemente der ListView zwischengespeichert. Die JSON-Objekte, welche durch einen Hintergrund-Thread AsyncTask<String...> der Klasse *GetQueueActivities* mit HTTP GET ermittelt werden, werden in der Liste *queueActJson* verwaltet.

```
btnActBlue.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
                macAddress = new ArrayList<String>();
                queueActivities = new ArrayList<String>();
                queueActJson = new ArrayList<JSONObject>();
                if (mBluetoothAdapter == null || !mBluetoothAdapter.
                   isEnabled()) {
                        Intent enableBtIntent = new Intent(
                           BluetoothAdapter.ACTION_REQUEST_ENABLE);
                        startActivityForResult(enableBtIntent,
                           REQUEST_ENABLE_BT);
                } else {
                if (Build.VERSION.SDK_INT >= 21) {
                        mLEScanner = mBluetoothAdapter.
                           getBluetoothLeScanner();
                        settings = new ScanSettings.Builder()
```

```
                        .setScanMode(ScanSettings.
                            SCAN_MODE_LOW_LATENCY)
                        .build();
                        filters = new ArrayList<ScanFilter>();
                }
                scanLeDevice(true);
        }
}
});
```

Listing 4.11: Ausschnitt aus MainActivity - OnClick auf Button 'Scan with BT'

Dagegen berücksichtigt der Aufruf auf dem Button *btnActBlue*, dass versteckte Aktivitäten durch erkannte BLE-Geräte in der Umgebung, wie im Ausschnitt abgebildet, zusätzlich angezeigt werden. Es wird überprüft und der Benutzer aufgefordert die Bluetooth-Funktion des Endgeräts anzuschalten. Für die Ausführung der Bluetooth-Funktionen ist die Implementierung eines BluetoothAdapter *mBluetoothAdapter* notwendig. Sind die Voraussetzungen erfüllt, wird die Methode *scanLeDevice(Boolean enable)* ausgeführt. Die Funktion aus dem unterem Codeabschnitt zeigt, dass es um einen verzögerten Thread handelt, der durch einen Callback*mScanCallback* alle ermittelten Hardware-Adressen der Bluetooth-Geräte in unmittelbarer Nähe anzeigt. Da sich jedoch im Testverlauf die MAC-Adresse des Gimbal Beacons stets verändert, wurde deswegen eine weitere, festgelegte Adresse hinzugefügt. Für weitere Details zum Umgang mit BLE auf Android-Applikationen sind die offizielle Dokumentation von Android [1] oder der Eintrag von Mohit Gupt [5] zu empfehlen.

```
private void scanLeDevice(final boolean enable) {
        if (enable) {
                mHandler.postDelayed(new Runnable() {
                @Override
                public void run() {
                        if (Build.VERSION.SDK_INT < 21) {
                                mBluetoothAdapter.stopLeScan(
                                    mLeScanCallback);
                        } else {
                        mLEScanner.stopScan(mScanCallback);
                        //Test added a fixed MAC address 1F.D1:3f
                            :52:3D:A9
```

```
                    //macAddress.add("1F.D1:3f:52:3D:A9");
                    for(int i = 0; i < macAddress.size(); i++) {
                            String newURL = actLink+"?address="+
                               macAddress.get(i);
                            new GetQueueActivitiesBlue().execute(
                               newURL);
                    }
            }               }
        }, SCAN_PERIOD);
        if (Build.VERSION.SDK_INT < 21) {
                mBluetoothAdapter.startLeScan(mLeScanCallback);
        } else {
        mLEScanner.startScan(filters, settings, mScanCallback);
}
} else {
if (Build.VERSION.SDK_INT < 21) {
        mBluetoothAdapter.stopLeScan(mLeScanCallback);
} else {
mLEScanner.stopScan(mScanCallback);
}
}
```

Listing 4.12: Scannt nach Bluetooth Low Energy Geräten

Werden die MAC-Adressen erfolgreich ermittelt, so wird führt die AsyncTask-Klasse *GetQueueActivitiesBlue(String...)* die GET-Operation über alle gefundenen Adressen auf. Das folgende Codebeispiel veranschaulicht den Aufbau einer AsyncTask:

```
public class GetQueueActivitiesBlue extends AsyncTask<String,
   String, String> {
        @Override
        protected String doInBackground(String... params) {
                HttpURLConnection con = null;
                BufferedReader reader = null;
                try {
                        URL url = new URL(params[0]);
                        con = (HttpURLConnection) url.openConnection
                           ();
                        con.connect();
```

```
                    reader = new BufferedReader(new
                        InputStreamReader(con.getInputStream()));
                    StringBuffer strbuffer = new StringBuffer();
                    String readline = "";
                    while ((readline = reader.readLine()) != null
                        ) {
                        strbuffer.append(readline);
                    }
                    return strbuffer.toString();
                } catch (MalformedURLException e) {
                e.printStackTrace();
        } catch (IOException e) {
        e.printStackTrace();
} finally {
if (con != null)
con.disconnect();
try {
        if (reader != null)
        reader.close();
} catch (IOException e) {
e.printStackTrace();
}
}
return null;
}

@Override
protected void onPostExecute(String result) {
        super.onPostExecute(result);
        try {
                JSONArray activities = new JSONArray(result);
                for(int i = 0; i < activities.length(); i++) {
                    JSONObject activity = activities.
                        getJSONObject(i);
                    String activityName = activity.getString("
                        activityname");
                    int connectStatus = activity.getInt("
                        connectstatus");
```

```
                    int activityId = activity.getInt("activityid"
                        );
                    String activityDescription = activityId+":"+
                        activityName+" - "+connectStatus;
                    if(!containsString(queueActivities,
                        activityDescription)) {
                            queueActivities.add(
                                activityDescription);
                            queueActJson.add(activity);
                            Log.i("Added", activityDescription);
                    }
                }
                listAdapter = new ArrayAdapter<String>(MainActivity.
                    this, android.R.layout.simple_list_item_1,
                    queueActivities);
                queueActivityList.setAdapter(listAdapter);
        } catch (JSONException e) {
        e.printStackTrace();
}
}
}
```

Listing 4.13: Scannt nach Bluetooth Low Energy Geräten

In der Methode *void doInBackground(String... params)* wird eine HTTP GET über die angegebene URL durchgeführt und die Ergebnisse als String auf einem UI-Thread in der Methode *void onPostExecute(String result)* weiterverarbeitet. Anschließend werden die Aktivitäten im JSON-Array in Datenformat String geparst und als ArrayList<String> auf der ListView queueActivityList angezeigt. Mit dem letzten Button *btnToMyAct* kann man anschließend zur Nebenaktivität *MyWorklistActivity* wechseln.

4.2.2 Anmerkung und Zusammenfassung

Die Nebenaktivität *MyWorklistActivity* ähnelt der Hauptaktivität, jedoch hat sie nur zwei Buttons: *btnMyAct* und *btnToQueueAct*. Mit *btnMyAct* werden die Aktivitäten aus befindlichen Arbeitsliste auf der ListView *myActivityList* ausgegeben. Klickt man dagegen auf , wird die Nebenaktivität beendet und der Benutzer kehrt zur

Hauptaktivität zurück. Ein weiterer Unterschied ist der Umgang mit den Listenelementen der ListView. Durch einen Klick auf das entsprechende Element, wird die gewählte Aktivität aus der Liste gelöscht und wieder in der ListView der Hauptaktivität MainActivity freigegeben. Nachdem nun die wichtigsten Implementierungsaspekte der Klasse MainActivity präsentiert wurden, folgt nun eine Zusammenfassung aller Operationen der Klasse MainActivity und MyWorklistActivity:

MainActivity Hier ist eine kurze Übersicht der wichtigsten Implementierungen zur Ausführung der Hauptaktivität:

- **void onCreate(Bundle savedInstanceState)** legt die Hauptaktivität an und stellt eine View anhand der definierten Layout-Ressourcen dar. Es wird eine Beschreibung im TextView der Arbeitsliste erstellt, eine leere ListView, und 3 Buttons mit Listener angelegt. Auch wird bereits überprüft, ob das verwendete Gerät Bluetooth unterstützt.
- **startMyActivities()** wechselt bzw. legt die Nebenaktivität MyWorklistActivity an.
- **void scanLeDevice(final boolean enable)** ist eine Methode, die einen Thread ausführt, um zunächst die MAC-Adressen-Liste der gefundenen BLE-Geräte zu ermitteln und anschließend die Aktivitäten anhand der Adressen zu bestimmen.
- **ScanCallback mScanCallback** ist keine Methode, jedoch dient der Callback dazu, in einem bestimmten Ausführungszustand des Scans, die ermittelten Informationen per Toast an den Client mitzuteilen und eine Liste von MAC-Adressen anzulegen.
- **class QueueWorklistInfo extends AsyncTask<String, String, String>** liest per GET-Request die Worklist-Ressource und baut anhand der Response eine Beschreibung der Arbeitsliste zusammen.
- **class GetQueueActivities extends AsyncTask<String, String, String>** ermöglicht über die angegebene URL alle zugänglichen Aktivitäten, somit kontextunabhängige Aktivitäten, zu ermitteln. Der String im Hintergrundprozess wird anschließend im UI-Thread verarbeitet und als Liste in der ListView dargestellt.
- **class GetQueueActivitiesBlue extends AsyncTask<String, String, String>** wird ähnlich aufgebaut wie *GetQueueActivities* und listet mit der ermittelten

Liste der MAC-Adressen *macAddress* zusätzlich alle versteckten Aktivitäten auf und zeigt diese in der Liste an.

- **class ActivityToMyActivities extends AsyncTask<String, String, String>** führt einen PUT-Request aus und setzt das selektierte Element der ListView in die Arbeitsliste MyActivities bzw. die ListView der Klasse MyWorklistActivity. Außerdem wird das entsprechende Element entfernt.

MyWorklistActivity In diesem Unterabschnitt werden nun die wichtigsten Methoden und Klassen der Nebenaktivität aufgelistet und erklärt:

- **void onCreate(Bundle savedInstanceState)** legt die Nebenaktivität an und stellt, wie in der Hauptaktivität beschrieben, eine View anhand der definierten Layout-Ressourcen dar. Lediglich werden hier die Beschreibung der MyActivities in der TextView, eine leere ListView und 2 weitere Buttons erstellt.
- **class MyWorklistInfo extends AsyncTask<String, String, String>** ist exakt identisch implementiert wie *QueueWorklistInfo*, liest die Beschreibung der übergebenen WorklistId aus und zeigt diese in der TextView an.
- **class GetMyActivities extends AsyncTask<String, String, String>** ist äquivalent zu *GetQueueActivities* und ermittelt alle Aktivitäten der übergebenen WorklistId. Im Gegensatz zur Hauptaktivität ist es hier nicht erforderlich die versteckten Aktivitäten mittels der MAC-Adressen zu ermitteln.
- **class ActivityToQueueActivities extends AsyncTask<String, String, String>** löscht das selektierte Element aus der ListView und führt einen PUT-Anfrage an den Server durch, um die Aktivität der Arbeitsliste QueueActivities wieder zur Verfügung zu stellen.

4.2.3 Vorstellung der Android-Applikation - WorklistModul

Im letzten Unterkapitel von Client- und Serverimplementierung werden hier nun die Interaktionen auf dem Prototyp präsentiert. Sobald man die Anwendung startet, wird man auf die Hauptaktivität mit einer leeren Liste geschickt (Abbildung 4.1). Die Beschreibung über der ListView wird beim Anlegen der Hauptaktivität geladen. Klickt man anschließend auf den Button 'Scan for tasks', so erhält man, wie im Bild 4.2 gezeigt, die Listenelemente oder Aktivitäten der Arbeitsliste QueueActivities. Möchte man zusätzlich zu den kontextunabhängigen Aktivitäten, die ortsabhängigen Prozessaktivitäten nachladen, so muss man den Button 'Scan with BT'

bestätigen. Falls das Bluetooth des Benutzers auf seinem Gerät ausgeschaltet ist, fordert die App diesen dazu auf die Bluetooth-Funktion freizuschalten (Abbildung 4.3). Nach einigen Sekunden, erscheinen Toast-Fenster mit den erkannten MAC-Adressen von BLE in der Umgebung (Abbildung 4.4). Sobald die Ausführung beendet wird, wird dem Benutzer in der Liste die zuvor versteckten Aktivitäten mit den registrierten MAC-Adressen angezeigt (Abbildung 4.5). Durch einen Klick auf einen Element der Liste, in Abbildung 4.6 dargestellt, wird die ausgewählte Aktivität unmittelbar in die Liste von MyActivities zugewiesen und aus der QueueActivities entfernt (Abbildung 4.7). Wechselt man mit dem Klick auf den Button 'MyActivities' wird die Nebenaktivität angelegt und abgebildet (Abbildungen 4.8 und 4.9. Auf 'Get my tasks' erhalten wir anschließend die Aktivitäten der Arbeitsliste MyActivities, wie in den Bildern 4.10 und 4.11 gezeigt. Um wieder auf die MainActivity zu kommen, klickt man auf den Button 'Return to QueueActivity'.

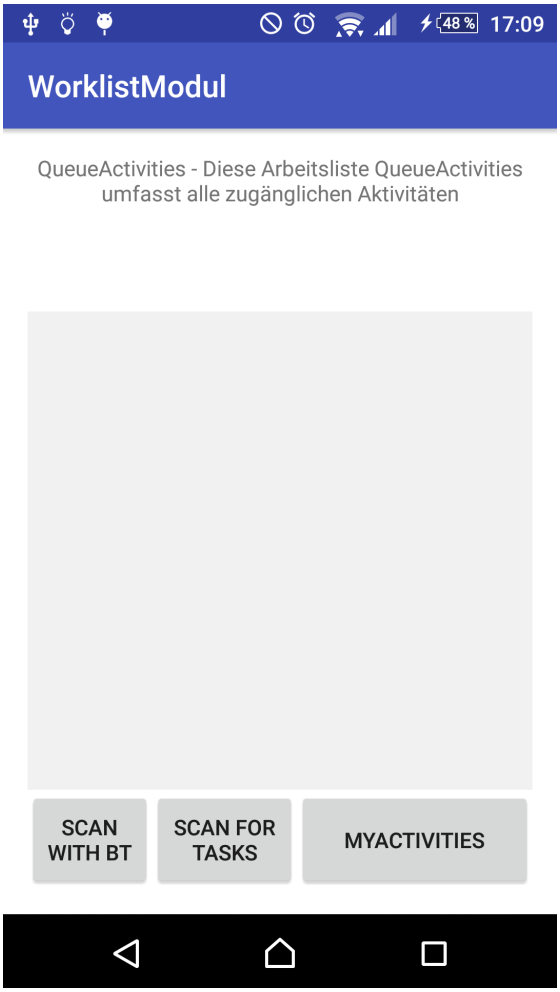

Abbildung 4.1: MainActivity - Nach Start der Anwendung

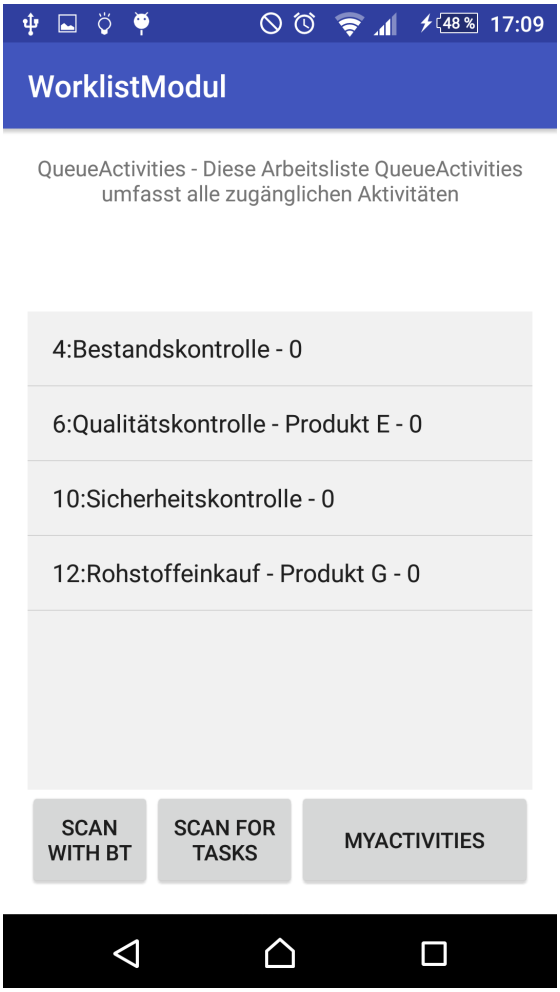

Abbildung 4.2: MainActivity - Nach Klick auf ’Scan for tasks’

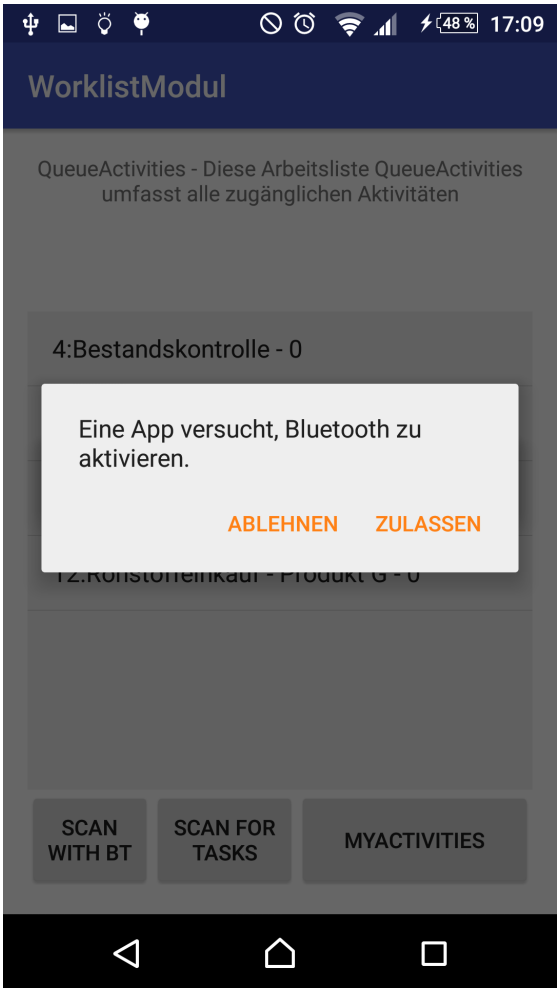

Abbildung 4.3: Architektur - Gesamtübersicht der verwendeten Komponenten

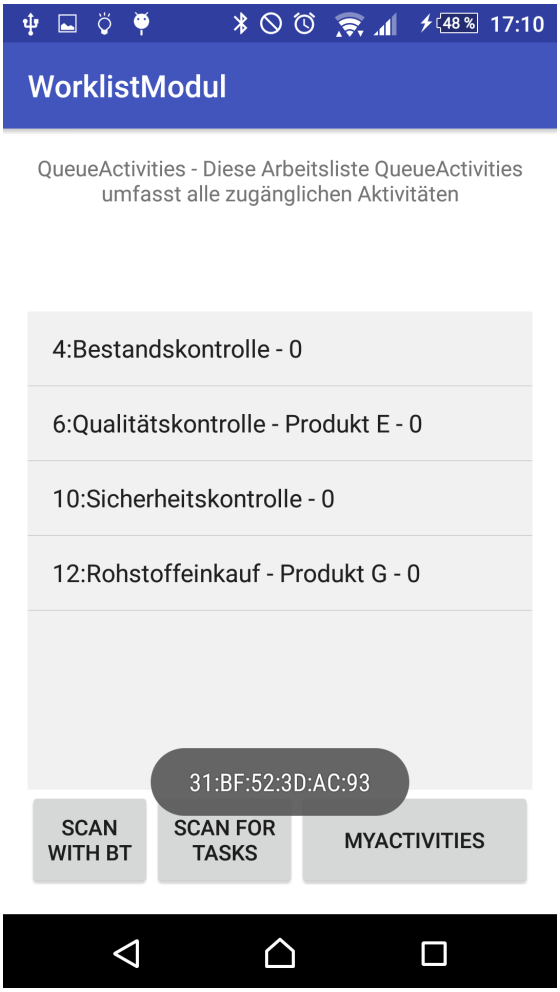

Abbildung 4.4: Architektur - Gesamtübersicht der verwendeten Komponenten

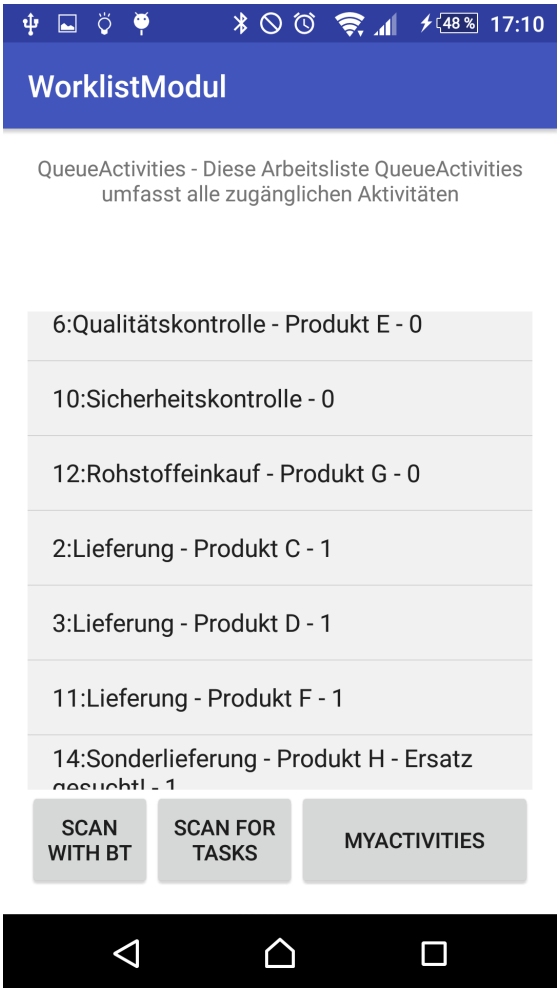

Abbildung 4.5: Architektur - Gesamtübersicht der verwendeten Komponenten

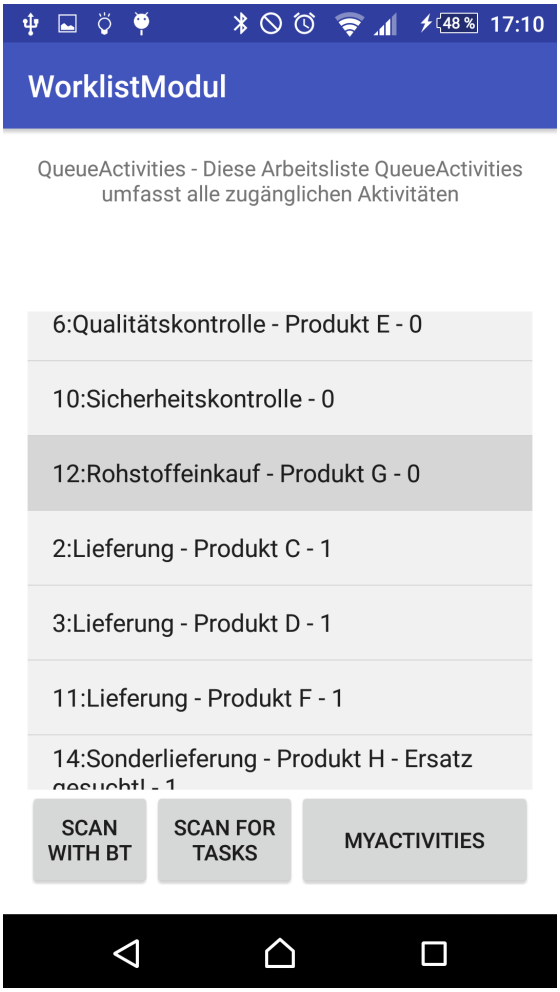

Abbildung 4.6: Architektur - Gesamtübersicht der verwendeten Komponenten

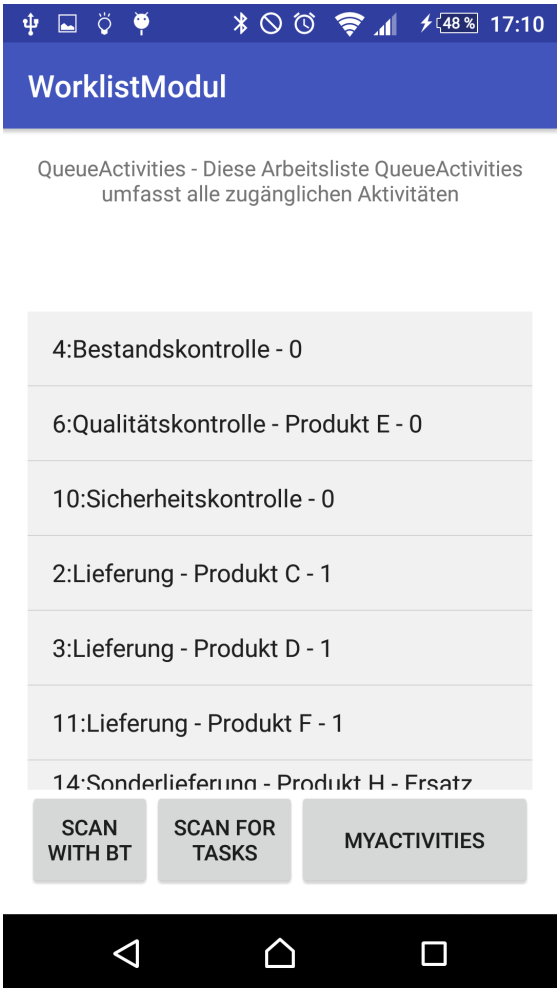

Abbildung 4.7: Architektur - Gesamtübersicht der verwendeten Komponenten

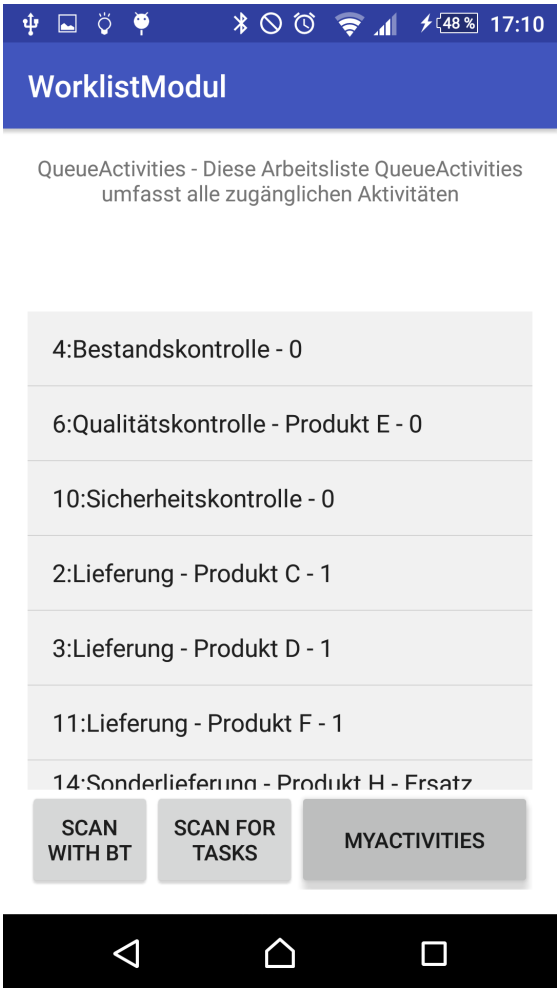

Abbildung 4.8: Architektur - Gesamtübersicht der verwendeten Komponenten

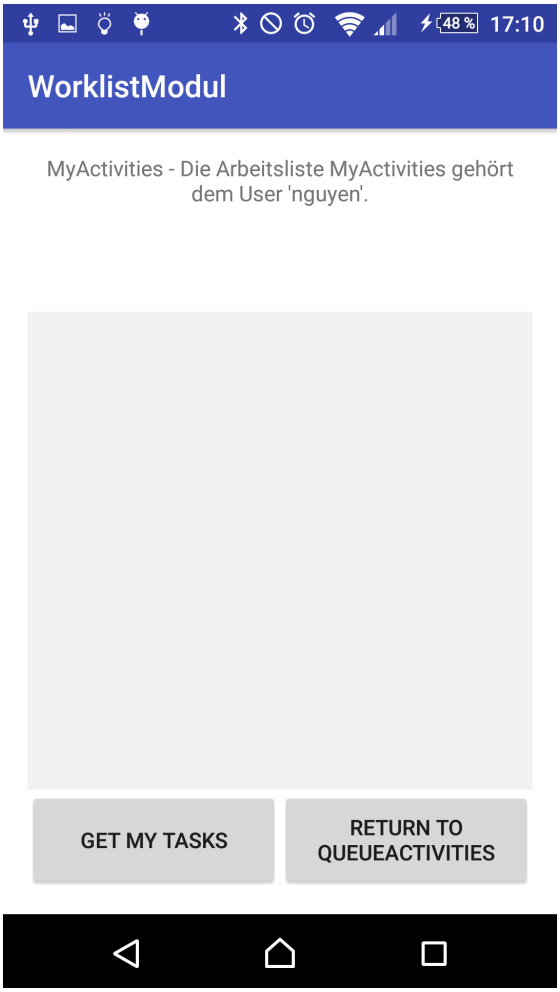

Abbildung 4.9: Architektur - Gesamtübersicht der verwendeten Komponenten

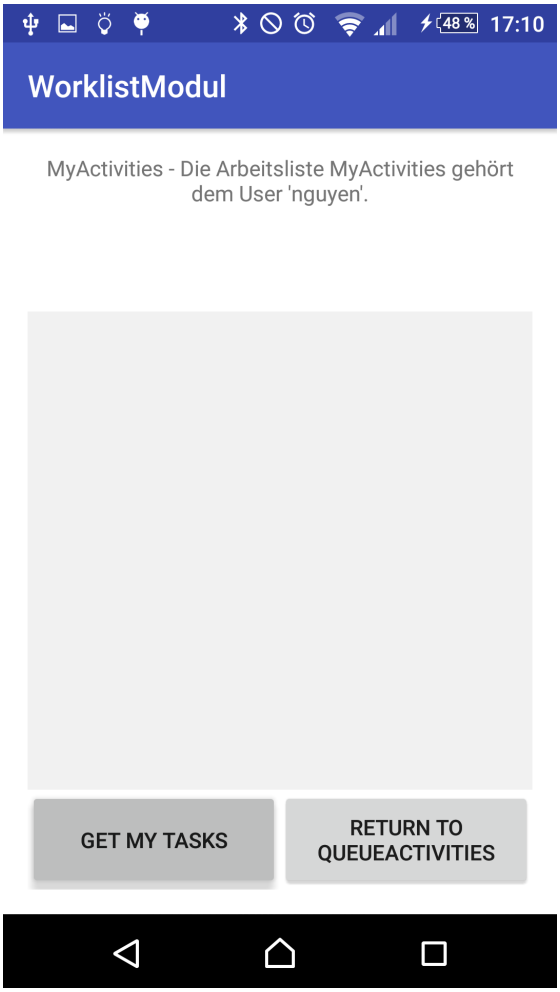

Abbildung 4.10: Architektur - Gesamtübersicht der verwendeten Komponenten

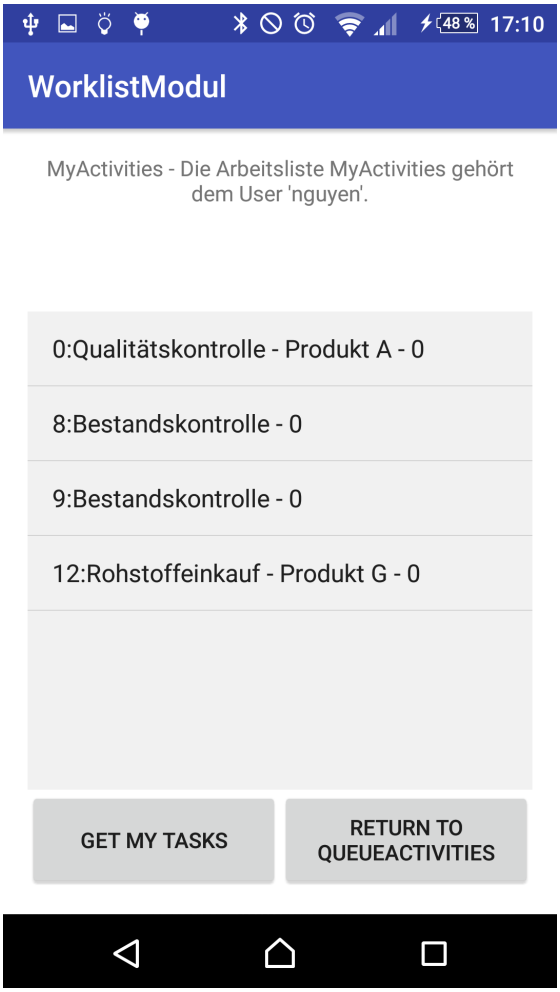

Abbildung 4.11: Architektur - Gesamtübersicht der verwendeten Komponenten

5 Abgleich der Projektanforderungen

Nachdem nun im vorherigen Kapitel die Implementierungsaspekte und das Modul ausführlich behandelt wurden, folgt nun der Abgleich der Projektanforderungen. Der folgender Inhalt setzt sich somit dem implementierten Prototyp und den anfänglichen Anforderungen in Kapitel 2 auseinander. Außerdem werden hier die Herausforderungen und Probleme während der Implementierung angesprochen.

5.1 Serveranforderungen

In der Implementierung wurde die Navigation zwischen den Arbeitslisten über REST berücksichtigt. Laut der Spezifikation kann man nicht zwischen zwei Arten von Arbeitslisten unterscheiden, stattdessen wurde festgelegt, dass die QueueActivities eine spezielle WorklistId besitzt. Unabhängig vom Typ den Arbeitslisten, verwaltet jede Liste seine eigenen Aktivitäten, die sich von Nutzer zu Nutzer unterscheiden. Jedoch ist zu berücksichtigen, dass nicht alle Kontextinformationen eingesetzt wurden, diese werden jedoch im Abschnitt 5.3 bedeutend sein. Die Schnittstelle erlaubt ebenfalls Worklist- oder Activity-Objekt eingeschränkt zu bearbeiten. Dabei erfolgen alle Kommunikationen zwischen dem Server und Client im Datenformat JSON. Jegliche HTTP-Operation wurde erfolgreich in CRUD der Datenbank realisiert. Auch einige Ausnahmen und Fehler in der Ausführung wurden mit generischen Fehlernachrichten abgefangen.

5.2 Clientanforderungen

Dagegen wurden im Modul jeweils eine Haupt- und Nebenaktivität erstellt, die eine Arbeitsliste QueueActivities und MyActivities repräsentieren. Mittels einer ListView

werden die Aktivitäten in String dargestellt. Allerdings wurde während der Implementierung nur eine ortsbezogene Kontextinformation berücksichtigt, die sich auf die Darstellung der Arbeitsliste QueueActivities auswirkt. Auf den angezeigten Elementen der Liste kann man mit einem Klick die gewählte Aktivität der entsprechenden Arbeitsliste hinzugefügt oder freigegeben. Dementsprechend startet die Anwendungen, bei einem Klick auf den Button *Scan for BT*, die Suche nach Beacon BLEs in unmittelbarer Nähe und präsentiert eine neue Liste von QueueActivities. Ein anderer Button ermöglicht hierbei die zwischen der Haupt- und Nebenaktivität zu wechseln.

5.3 Fehlende Komponenten

Trotz der ausführlichen Behandlung der funktionalen Anforderungen, konnte im Laufe des Prototypings nicht alle Komponenten und Anforderungen erfolgreich umgesetzt werden. Denn während der Entwicklung wurden ein RESTful Webservice und eine Datenbank verwendet statt einem Prozessmanagementsystem. Lediglich wurde in diesem Projekt ein mobiler Kontext betrachtet, welche den Ausführungsort ansatzweise simuliert. In der Dissertation [9, S. 25-28] von Rüdiger Pryss sind zudem mehr Anforderungen zum mobilen Kontext und der mobilen Aktivitäten zu berücksichtigen. Aus diesem Grund wurden in dem Projekt einige weitere kontextabhängige Informationen hinzugefügt. Diese sollen verdeutlichen, dass die Zuteilung und Ermittelung mobiler Aktivitäten den entsprechenden Werten abhängen. Ein weiterer Aspekt der in diesem Prototyp fehlt, ist die Verwaltung von Benutzerkonten, welche ausschlaggebend für den Zugriff und Zuteilung bestimmter Aktivitäten sind. Mit Fabian Meier findet man andere Arbeit, in welchem er sich mit der Entwicklung eines mobilen Workflow-Clients in Android auf AristaFlow BPM Suite und den Aspekt von Session und Benutzerkonten auseinandersetzte. Wie in seiner Arbeit wurde der Designaspekt während des Prototypings ignoriert.

5.4 Ausnahmebehandlungen und Robustheit

Neben den obengenannten Punkten, fehlt der Umgang und das Design von prozessabhängigen Fehlern, trotz der Einführung von generischen Ausnahmebehandlungen im RESTful Webservice. Wie man auf die kontextabhängigen Informationen von mobilen Prozessaktivitäten in Ausnahme- und Fehlerfällen reagiert, beschreibt

Ekaterina Speda in ihrer Ausarbeitung *Usability-Konzept für die Fehlerbehandlung von mobilen Prozess-Aktivitäten* [16]. Unter dem Design Guideline können somit die Fehlermeldungen angepasst und die Benutzerfreundlichkeit, sowie Verständlichkeit in der Fehlerbehandlung verbessert werden. Für die Robustheit des Moduls trägt die Ausarbeitung von Steffen Musiol zum Thema *A Process Engine Independent Architecture Enabling Robust Execution of Mobile Tasks in Business Processes* bei [8]. Dabei präsentiert er in seiner Arbeit einen Ansatz einer Schnittstelle bzw. Middleware, welche die Robustheit während der Ausführung von mobilen Prozessaktivitäten gewährleisten kann. Im nächsten Kapitel werden noch weitere verwandte Arbeiten präsentiert, die sich mit den Kernproblemen und Herausforderungen von mobilen Prozessaktivitäten beschäftigen.

6 Verwandte Arbeiten

In vorletzten Kapitel werden nun weitere Forschungsarbeiten angeführt, die sich intensiver mit dem Konzept und Einsatz von mobilen Prozessaktivitäten in Geschäftsprozessen auseinandersetzen. Die verwandten Arbeiten sollen dabei ein tieferes Verständnis in die Konzepte und Möglichkeiten von mobilen Prozessaktivitäten ermöglichen. Beispielsweise beschreiben Rüdiger Pryss, Steffen Musiol und Manfred Reichert in ihrer gemeinsamen Publikation *Integrating Mobile Tasks with Business Processes: A Self-Healing Approach* Ansätze für die Integration von mobilen Kontextinformationen in dynamischen Geschäftsprozessen [11]. Untersucht und veranschaulicht wird dabei der Gesichtspunkt, dass smarte, mobile Endgeräte genutzt werden, um kontextabhängige Informationen, während den dynamischen Geschäftsprozessen, zu ermitteln und diese im Ablauf zu integrieren. In ihrer weiteren Veröffentlichung *Extending Business Processes with Mobile Task Support: A Self-Healing Solution Architecture*, führen die selben Autoren weitere Paradigmen und Methoden zum Einsatz und Design von mobilen Aktivitäten in Prozessmanagementsystemen ein [12]. Ein Praktischer Anwendungsbereich für mobile Endgeräte in Geschäftsabläufen sind beispielsweise klinische, psychologische Abteilungen, welches im Konferenzpapier *Mobile Task Management for Medical Ward Rounds - The MEDo Approach* [10] von Rüdiger Pryss, David Langer, Manfred Reichert, Alena Hallerbach näher erläutert wird. Möchte man sich dagegen intensiver mit den technischen Möglichkeiten von mobilen Endgeräten in Prozessausführungen auseinandersetzen, baut die Publikation *BPM to Go: Supporting Business Processes in a Mobile and Sensing World* [14] auf den Bereichen Sensorik und Ortung von mit Mobilgeräten auf. Da diese Konzepte eine robuste und kontextbezogene Ausführung, insbesondere während einer aktiven Prozessausführung, voraussetzen, beschäftigen sich die Dissertation von Rüdiger Pryss [9] und die wissenschaftliche Arbeit *Robust Execution of Mobile Activities in Process-Aware Information Systems* [13] mit den Problemen, Konzepten, Design und Herausforderungen von mobilen Aktivitäten in Prozessmanagementsystemen. Dass anschließend menschliche Interaktionen in einer mobilen Infrastruktur Einfluss auf die Prozessabläufe nehmen können, wird in der gemeinsamen Publikation von Rüdiger Pryss, Manfred Rei-

chert, Marc Schickler und Thomas Bauer zum Thema *Context-Based Assignment and Execution of Human-Centric Mobile Services* dargelegt und beschrieben [15].

7 Zusammenfassung

Obwohl das Projekt nicht alle genannten Anforderungen und Konzepte umsetzen konnte, sollte diese Arbeit zeigen, dass die heutigen, mobilen Endgeräte durchaus in der Lage sind, Kontextinformationen in Echtzeit für Prozessmanagementsysteme bereitzustellen. Um diesen Aspekt zu untersuchen, wurde in der Konzeption des Moduls ein mobiler Kontext festgelegt: die ortsbezogene Kontextinformation, welches durch Beacons dargestellt wird. Da jedoch die kontextbezogenen Aktivitäten in der Realität von mehreren Faktoren abhängig sind und die Aktivitäten in den mobilen Arbeitslisten der Endanwender gesteuert werden, sollte diese Arbeit als Einstieg in die Realisierung der funktionalen Anforderungen des mobilen Kontext und der mobilen Aktivität dienen. Des Weiteren wurde in der Arbeit angenommen, dass die angelegten Aktivitäten in diesem Format bereitgestellt werden, sodass man dieses Gesamtprojekt als Referenz für künftige Implementierung in diesem Themengebiet behandeln sollte. Betrachtet man die prototypische Implementierung von Fabian Maier [7, S. 25ff], so erkennt man zudem, dass einige Implementierungsaspekte in der Android-Applikation bereits veraltet sind. Ebenfalls werden Web Services in SOAP mit XML durch REST und dem Datenformat JSON ersetzt. Somit sollte mit der Implementierung eines 'Prozessmanagementsystems' als REST-Schnittstelle und einer Android-Anwendung, auch der aktuelle Stand der Implementierungskonzepte und Strukturen festgehalten werden. Trotz des Schwerpunkts der Arbeit auf die Entwicklung der Android-Anwendung dürfen auch die Vorzüge und die Herausforderungen von mobilen Endgeräten wie Smartphones, Tablet-PC, aber auch Smartwatches in den kommenden Jahren nicht ignoriert werden.

Literaturverzeichnis

[1] *Android - Bluetooth Low Energy*. https://developer.android.com/guide/topics/connectivity/bluetooth-le.html, . – Letzter Zugriff: 08-03-2017

[2] *Apache Tomcat 8 - Documentation Index*. http://tomcat.apache.org/tomcat-8.0-doc/index.html, . – Letzter Zugriff: 08-03-2017

[3] *JAX-RS 2.0 API Specification*. https://jax-rs-spec.java.net/nonav/2.0-rev-a/apidocs/index.html, . – Letzter Zugriff: 20-01-2017

[4] *Jersey API Documentation*. https://jersey.java.net/apidocs/latest/jersey/index.html, . – Letzter Zugriff: 08-03-2017

[5] GUPT, M. : *Android Bluetooth Low Energy (BLE) Example*. http://www.truiton.com/2015/04/android-bluetooth-low-energy-ble-example/, . – Letzter Zugriff: 08-03-2017

[6] KOTHAGAL, K. : *Developing REST APIs with JAX-RS*. https://javabrains.io/courses/javaee_jaxrs, . – Letzter Zugriff: 20-01-2017

[7] MAIER, F. : *Entwicklung eines mobilen und Service getriebenen Workflow-Clients zur Unterstützung von evaluierten Studien der klinischen Psychologie am Beispiel der AristaFlow BPM Suite und Android*. 2012

[8] MUSIOL, S. : *A Process Engine Independent Architecture Enabling Robust Execution of Mobile Tasks in Business Processes*. 2014

[9] PRYSS, R. : *Robuste und kontextbezogene Ausführung mobiler Aktivitäten in Prozessumgebungen*. 2015

[10] PRYSS, R. ; LANGER, D. ; REICHERT, M. ; HALLERBACH, A. : Mobile Task Management for Medical Ward Rounds - The MEDo Approach. In: *1st Int'l Workshop on Adaptive Case Management (ACM'12), BPM'12 Workshops* (2012), S. 43–54

[11] PRYSS, R. ; MUSIOL, S. ; REICHERT, M. : Integrating Mobile Tasks with Business Processes: A Self-Healing Approach. In: *Handbook of Research on Architectural Trends in Service-Driven Computing*. 2014, S. 103–135

[12] PRYSS, R. ; MUSIOL, S. ; REICHERT, M. : Extending Business Processes with Mobile Task Support: A Self-Healing Solution Architecture. In: *Leadership and Personnel Management: Concepts, Methodologies, Tools, and Applications: Concepts, Methodologies, Tools, and Applications* (2016), S. 273

[13] PRYSS, R. ; REICHERT, M. : Robust Execution of Mobile Activities in Process-Aware Information Systems. In: *International Journal of Information System Modeling and Design (IJISMD)*. 2016, S. 33

[14] PRYSS, R. ; REICHERT, M. ; BACHMEIER, A. ; ALBACH, J. : BPM to Go: Supporting Business Processes in a Mobile and Sensing World. In: *BPM Everywhere*. 2015, S. 167–182

[15] PRYSS, R. ; REICHERT, M. ; SCHICKLER, M. ; BAUER, T. : Context-Based Assignment and Execution of Human-Centric Mobile Services. In: *5th IEEE International Conference on Mobile Services (MS 2016)*. IEEE Computer Society Press, 2016, S. 119–126

[16] SPEDA, E. : *Usability-Konzept für die Fehlerbehandlung von mobilen Prozess-Aktivitäten*. 2015

Name: Tran Bao Dat Nguyen Matrikelnummer: 787594

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Tran Bao Dat Nguyen