Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Human Tasks für OpenTOSCA zum Aufsetzen von IoT-Anwendungen

Isabella Kutger

Studiengang:	Medieninformatik
Prüfer/in:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Betreuer/in:	Ana Cristina Franco da Silva, M.Sc.
Beginn am:	16. April 2018
Beendet am:	16. Oktober 2018

Kurzfassung

Internet of Things (IoT) ist ein kontinuierlich wachsendes Feld, wodurch immer mehr IoT-Umgebungen entstehen. Diese Umgebungen sind komplex und das manuelle Aufsetzten von Anwendungen auf solchen Umgebungen ist sehr aufwendig. Daher wird ein System benötigt, das diesen Vorgang optimal unterstützt. Ein solches System ist der OpenTOSCA-Container. Mit Hilfe von OpenTOSCA kann eine IoT-Anwendung fast automatisch aufgesetzt werden. Jedoch gibt es immer noch Aufgaben, die nicht von OpenTOSCA erledigt werden können. Diese Tätigkeiten müssen von einem Menschen ausgeführt werden, wodurch sie die Bezeichnung *Human Tasks* erhalten. Um Human Tasks optimal verwalten zu können, wird im Rahmen dieser Bachelorarbeit ein Konzept entwickelt, das Human Tasks beim Aufsetzen einer IoT-Anwendung mit OpenTOSCA unterstützt. Dem Konzept liegt die *Web Services - Human Task Specification* der Organisation OASIS zugrunde. Hierfür wird ein eigenständiger Task Manager, der über eine API die Human Tasks von OpenTOSCA erhält, entwickelt. Dieser Task Manager ist für die Verwaltung der Human Tasks zuständig. Weiterhin wird im Rahmen dieser Arbeit eine Smartphone-App entworfen, um Human Tasks für die Nutzer darzustellen. Basierend auf diesem Konzept wird ein Prototyp implementiert, der zur Verwaltung von Human Tasks beim Aufsetzen von IoT-Anwendungen mit OpenTOSCA genutzt werden kann. Dabei wird eine vorhandene Implementierung des Task Managers auf das Konzept zugeschnitten und eine Android-App für die Darstellung der Human Tasks realisiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

API Application Programming Interface. 16

CSAR Cloud Service Archive. 19

DI Dependency Injection. 56

HTTP Hypertext Transfer Protocol. 56

IoC Inversion of Control. 56

IoT Internet of Things. 15

JSON JavaScript Object Notation. 31

LPG Logical People Group. 26

MVC Model View Controller. 67

POJO Plain Old Java Objects. 56

REST Representational State Transfer. 31

SQL Structured Query Language. 53

SSH Secure Shell. 16

TOSCA Topology and Orchestration Specification for Cloud Applications. 19

UI User Interface. 23

URI Uniform Resource Identifier. 56

UTC Coordinated Universal Time. 23

WAR Web Application Archive. 53

WS Web Services. 17

WSDL Web Service Description Language. 26

XML Extensible Markup Language. 21

1. Einleitung

Im Zeitalter der Digitalisierung ist *Internet of Things (IoT)* [Atz+10] ein kontinuierlich wachsendes Feld. Gerade die Vernetzung des Alltags nimmt immer weiter zu, wodurch ganze IoT-Umgebungen entstehen. Das wohl bekannteste Beispiel für ein solches System ist das *Smart Home*, in dem Geräte in einem Haus miteinander vernetzt sind und automatisch auf verschiedene Begebenheiten reagieren. Das Paradigma hinter IoT beschäftigt sich mit der Vernetzung und dem Zusammenspiel von *Smart Devices*, die mit Sensoren und Aktuatoren ausgestattet sind. Smart Devices können somit die Umwelt durch ihre Sensoren wahrnehmen und mit ihren Aktuatoren auf Veränderungen reagieren. Eine Anwendung, die eine Vernetzung von mehreren solcher Geräte beinhaltet und ihre Daten verarbeitet, wird eine IoT-Anwendung genannt. Bei Smart Devices, Sensoren und Aktuatoren spricht man von einer IoT-Umgebung [Atz+10]. Diese Umgebungen sind komplex und das rein manuelle Aufsetzen von Anwendungen auf solchen Umgebungen ist sehr aufwendig. Aus diesem Grund wird ein System benötigt, welches das hybride (teils manuelle, teils automatische) Aufsetzen optimal unterstützt.

Ein solches System ist OpenTOSCA-Container [Bin+13], welcher den TOSCA-Standard implementiert. Mit Hilfe von OpenTOSCA kann eine IoT-Anwendung fast automatisch aufgesetzt werden. Jedoch gibt es immer noch Aufgaben, die nicht von OpenTOSCA erledigt werden können. Darunter fallen zum Beispiel das Anschließen eines Smart Devices oder das Bestücken von Microcontrollern (wie Raspberry Pis) mit Sensoren [Sil+16; Sil+17]. Diese Tätigkeiten muss daher der Mensch übernehmen, weshalb sie *Human Tasks* genannt werden [OAS A].

In dieser Arbeit wird ein Konzept entwickelt, das Human Tasks(kurz: Tasks) beim Aufsetzen einer IoT-Anwendung mit OpenTOSCA unterstützt. Mit ihm soll es möglich sein, Tasks zu verwalten und potenziellen Nutzern anzuzeigen. Hierfür wird eine eigenständige Komponente, die von OpenTOSCA die Tasks über eine API erhält, in Form eines Task Managers bereitgestellt. Dieser Task Manager ist für die Verwaltung der Tasks zuständig. Weiterhin wird in dieser Arbeit ein Konzept entwickelt, um Human Tasks für Nutzer darzustellen. Dies wird mit Hilfe einer Smartphone-App realisiert. Eine App eignet sich in diesem Fall, da das Smartphone heutzutage ein ständiger Begleiter ist. So können Nutzer jeder Zeit über neue Tasks benachrichtigt werden. Dadurch wird außerdem ein direkteres Umsetzen dieser Aufgaben ermöglicht, da der Nutzer zeitnah auf Human Tasks reagieren kann.

1.1. Ziele der Arbeit und Anwendungsfall

In diesem Abschnitt wird der zugrunde liegende Anwendungsfall und die Ziele dieser Bachelorarbeit vorgestellt.

Diese Arbeit soll ermöglichen, Human Tasks, welche beispielsweise beim Bereitstellen eines Raspberry Pis anfallen, zu bearbeiten. Diese Aufgaben umfassen unter anderem das Anschließen der nötigen Sensoren und eine grundlegende Konfiguration des Raspberry Pis. Die Ergebnisse von einem bearbeiteten Task sollen abgerufen werden können, sodass diese Informationen für das automatische Aufsetzen von IoT-Anwendungen mittels OpenTOSCA benutzt werden können. Hierzu wurden Beispieltasks erstellt, die schließlich mit dem Prototyp bearbeitet werden können.

Um Human Tasks beim Aufsetzen einer IoT-Anwendung entsprechend bearbeiten zu können, sollte das System die folgenden Anforderungen umsetzen. Zur Verwaltung der Tasks wird ein Task Manager benötigt, der eine *Application Programming Interface (API)* bereitstellt. Mit der API soll es möglich sein, zum einen Tasks zu bearbeiten und zum anderen Nutzer zu verwalten. Nutzer besitzen hierbei eine oder mehrere Rollen, denen jeweils bestimmte Tasks zugewiesen werden. Tasks erhalten Tasktypen, durch die sie in Kategorien zusammengefasst werden. Nur Nutzer mit der zugewiesenen Gruppe können den Task ausführen.

Außerdem benötigt die Bearbeitung von Human Tasks eine geeignete Benutzeroberfläche. Dafür wird eine Smartphone-App realisiert. Sie soll dem Nutzer alle Funktionen zur Verfügung stellen, die für die Bearbeitung von Human Tasks notwendig sind, und kommuniziert ebenfalls durch die API mit dem Task Manager. Damit der Nutzer auf neue Tasks aufmerksam wird, soll er eine Benachrichtigung erhalten.

Im Folgenden wird auf die Funktionen, die unterstützt werden sollen, eingegangen. Im Bereich der Nutzerverwaltung müssen Methoden zum Erstellen von Nutzern und Rollen bzw. Gruppen bereitgestellt werden sowie die Möglichkeit, diese abzufragen. Für die Taskverwaltung werden Operationen zum Erstellen von Tasks benötigt. Außerdem sollen Methoden zum Manipulieren des Lebenszyklus eines Tasks vorhanden sein. Das heißt, es muss unter anderem die Möglichkeit geben, einen Task zu starten oder ihn zu beanspruchen. Weiterhin muss ein Task als „Abgeschlossen“ markiert werden können. Hierbei ist wichtig, dass auch Funktionen für Output-Parameter unterstützt werden. Sie geben dem Nutzer die Möglichkeit, Informationen, welche nach der Bearbeitung in OpenTOSCA benötigt werden, an den Task anzuhängen. Damit die Smartphone-App den Task anzeigen und OpenTOSCA die Informationen von abgeschlossenen Tasks anfragen kann, muss es eine Methode geben, die alle benötigten Informationen über einen Task zurückgibt.

Der Ablauf eines Beispielszenarios sollte wie folgt aussehen. Zuerst registriert sich der Nutzer mit Hilfe der Smartphone-App beim Task Manager. Um eine IoT-Anwendung auf einem Raspberry Pi zu installieren, wird unter anderem ein Secure Shell (SSH)-Zugang benötigt. Dafür informiert OpenTOSCA den Task Manager über die API und ein Task wird erstellt. Die Smartphone-App fragt in regelmäßigen Abständen die neuen Tasks beim

Task Manager an und benachrichtigt die entsprechenden Rollen über die neu verfügbaren Tasks. Schließlich beansprucht einer der Nutzer innerhalb der betroffenen Rolle den Task und bearbeitet diesen. Nach Beendigung der Ausführung schickt die Smartphone-App die benötigten Daten an den Task Manager zurück. OpenTOSCA kann dabei jeder Zeit den aktuellen Stand des Tasks über die API abrufen.

Für die Umsetzung des Anwendungsfalls wird folgendermaßen vorgegangen:

- Erarbeitung eines Konzepts für Human Tasks zum Aufsetzen von IoT-Anwendungen basierend auf *Web Services (WS) Human Tasks*
- Erweiterung einer vorhandenen Softwarekomponente [Wag10] zum Verwalten von Human Tasks
- Implementierung einer Smartphone-App zum Bearbeiten von Human Tasks durch Nutzer

In Kapitel 2 wird auf die Grundlagen für diese Bachelorarbeit eingegangen. Anschließend wird in Kapitel 3 das dem Anwendungsfall angepasste Konzept für Human Tasks präsentiert. Danach gibt Kapitel 4 einen Überblick über die Implementierung des Task Managers und dem Task Manager Client in Form einer Smartphone-App. Kapitel 5 verweist auf verwandte Arbeiten und Kapitel 6 fasst die Arbeit kurz zusammen und gibt einen Ausblick.

2. Grundlagen

Im Folgenden wird ein Überblick über die nötigen Grundlagen dieser Bachelorarbeit gegeben. Hierfür notwendig ist zunächst ein Überblick über *Topology and Orchestration Specification for Cloud Applications (TOSCA)* und dessen Laufzeitumgebung OpenTOSCA. Außerdem wird eine Einführung in die Spezifikation der *WS Human Tasks* gegeben, welche die Grundlage zur Definition der Tasks und deren Behandlung liefert.

2.1. TOSCA und OpenTOSCA

Das Akronym TOSCA steht für *Topology and Orchestration Specification for Cloud Applications*. Dahinter verbirgt sich eine Spezifikation der Organisation OASIS für die Standardisierung von portierbaren Cloud-basierten Anwendungen, dabei wird vor allem auf Struktur und Managementaspekte eingegangen. Die aktuelle Version der Spezifikation ist Version 1.0 aus dem Jahr 2013 [OAS N].

TOSCA verwendet Topologien, um den Aufbau von Anwendungen zu definieren, welche durch *Build Plans* bereitgestellt werden. Diese Topologien bestehen aus Knoten (*node templates*), die die Komponenten der Anwendung repräsentieren, und Kanten (*relationship templates*), welche Abhängigkeiten und Beziehungen zwischen Komponenten darstellen. Da die TOSCA-Spezifikation generisch ist, ist es möglich beliebige Typen von Kanten und Knoten zu definieren, welche *node types* und *relationship types* entsprechen. Diese Typen spezifizieren bestimmte Eigenschaften und Management Operationen. Eigenschaften können z.B. `version` oder `port` sein. Management Operationen werden unter anderem zum Bereitstellen oder Start einer Komponente der Anwendung genutzt. In diesem Fall verwirklichen *Implementation Artifacts* diese Operationen. Die Implementierung einer Komponente ist in sogenannten *Deployment Artifacts* enthalten, z.B. Betriebssysteme oder Archive mit Quellcode. Die fertige TOSCA-Topologie und die dazugehörigen Artefakte werden im TOSCA Paketformat *Cloud Service Archive (CSAR)* gebündelt.

Für die Ausführung einer TOSCA-Topologie existieren zwei verschiedene Ansätze. Zum einen das *Imperative Processing* und zum anderen das *Declarative Processing*. Das Declarative Processing schiebt die Logik für Bereitstellung und Management vom Build Plan zur Laufzeitumgebung. Daher muss bei diesem Ansatz die Laufzeitumgebung die nötigen Operationen kennen und sie in der richtigen Reihenfolge durchführen, weshalb diese Ausführungsart nur für einfache Anwendungen geeignet ist. Dagegen benutzt das

Imperative Processing die Implementierung der Pläne, welche die gewünschte Aufgabe erfüllen. Der imperative Ansatz nutzt also die bereitgestellten Management Operationen der Implementation Artifacts [Bin+13; Sil+16; OAS N].

Wie schon zuvor erwähnt benötigt TOSCA eine Laufzeitumgebung um seine Topologien zu verwirklichen. Eine solche wäre OpenTOSCA Container [Bin+13]. OpenTOSCA Container wurde vom Institut für Architektur von Anwendungssystemen der Universität Stuttgart entwickelt und implementiert den TOSCA Standard, wodurch es zur automatischen Bereitstellung von TOSCA-Modellen dient. Dazu steuert OpenTOSCA die Management Operationen, führt die Pläne aus und verwaltet den Status der neu aufgesetzten Anwendung. Dabei werden Anfragen an eine Kontrollkomponente getätigt, die die verschiedenen Komponenten des Build Plans der TOSCA-Anwendung organisiert und interpretiert. Gemeinsame Services, wie z.B. Datenmanagement, werden in OpenTOSCA von der Kernkomponente verwaltet. Es werden sowohl Imperative Processing als auch Declarative Processing unterstützt. Um die Management Operationen zu nutzen, müssen diese entweder von laufenden Services bereitgestellt werden oder in den Implementation Artifacts des CSARs enthalten sein. Bei Letzterem werden die Implementation Artifacts von zugehörigen Plugins der *Implementation Artifact Engine*, die die Verarbeitung der Artefakte kennen, durchgeführt. Diese führen die Artefakte aus und speichern die Endpunkte der bereitgestellten Management Operationen in einer hierfür angelegten Datenbank. Auch die im CSAR mitgelieferten Management Pläne haben eine eigene Engine, die entsprechende Plugins zur Verfügung stellt. Diese Plugins binden dann den jeweiligen Management Plan an die zuvor gespeicherten Endpunkte der dazugehörigen Operation. So ist es möglich, Management Pläne zwischen verschiedenen Umgebungen zu portieren [Bin+13; Bre+14].

Darüber hinaus eignet sich TOSCA für IoT-Umgebungen durch das vereinfachte Bereitstellen von Anwendungen bzw. ganzen Systemen mit Hilfe des Standards. Da eine manuelle Einrichtung einer IoT-Umgebung zu zeitaufwändig und fehleranfällig wäre, sollte diese automatisch erfolgen. Mit der Hilfe von TOSCA können die Hardwarekomponenten eingerichtet und diese an die benötigte IoT-Middleware gebunden werden [Sil+16; Sil+17].

Ein Beispiel, wie eine TOSCA-Topologie für eine IoT-Anwendung aussehen könnte, zeigt Abbildung 2.1. Dieses Beispiel stammt aus [Sil+16]. Das Ziel dieser Anwendung ist es, dass ein Raspberry Pi, mit einem Temperatursensor ausgestattet, die Temperaturwerte einliest und ein zweiter Raspberry Pi anhand dieser Werte einen Ventilator einschaltet bzw. ausschaltet. TOSCA kann die nötigen Konfigurationen automatisch vornehmen, z.B. das Aufspielen der benötigten Python-Scripts auf die Raspberry Pis oder das Herstellen der SSH Verbindungen. Jedoch fallen hier noch einige Vorbereitungen an, die vor der Bereitstellung durch OpenTOSCA durchgeführt werden müssen. Eine dieser Aufgaben wäre das Konfigurieren der beiden Raspberry Pis [Sil+16]. Solche Aufgaben können leider nicht völlig automatisch erledigt werden und bedürfen immer noch dem Eingreifen eines Menschen, der bestimmte Schritte beim Aufsetzen einer IoT-Anwendung manuell

durchführt. Diese Aufgaben betreffen meist die Hardwarekomponenten, welche per Hand angeschlossen bzw. zusammengesetzt werden müssen und eine grundlegende Konfiguration vor dem Einsatz benötigen.

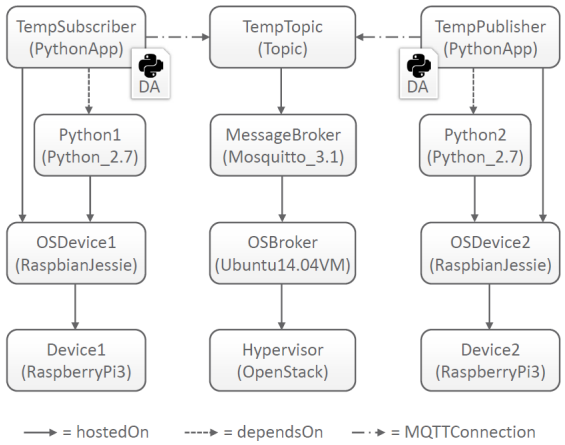


Abbildung 2.1.: Beispiel TOSCA-Topologie einer IoT-Anwendung [Sil+16]

Wie schon in Kapitel 1 erwähnt, werden Aufgaben, die von Menschen erledigt werden müssen, *Human Tasks* genannt.

Nach der vorangegangenen Vorstellung des TOSCA-Standards und seiner Laufzeitumgebung OpenTOSCA, wird im Folgenden auf die Human Tasks eingegangen und ein Überblick über die dazugehörige WS-Spezifikation gegeben.

2.2. WS Human Tasks

Die Definition von *Human Tasks* beruht auf der Spezifikation *Web Services – Human Task (WS-HumanTask) Specification* der Organisation OASIS [OAS A]. Die Spezifikation umfasst unter anderem eine Beschreibung der Human Tasks, Definition verschiedener generischer Nutzerrollen und Zuordnung von Nutzern zu Taskinstanzen (kurz: Tasks).

Human Tasks sind Aufgaben, die von Menschen übernommen werden müssen. Diese Aufgaben sind hauptsächlich Tätigkeiten, die mit einem Rechner nicht bzw. noch nicht realisiert werden können, oder dienen der Überwachung der Abläufe in einem *Workflow Management System*. Beispiele hierfür sind unter anderem das Anschließen von benötigter Hardware oder die Zustimmung zu bestimmten Abläufen, z.B. beim Kauf eines teuren Bauteils oder der Genehmigung eines Kredits. Das heißt, ein Task ist im Grunde eine Anfrage an eine Person, eine bestimmte Tätigkeit auszuführen.

Jeder Task wird einem Taskmodell zugeordnet. Diese Taskmodelle können von verschiedenen *Task Parents* stammen, das bedeutet verschiedene Systeme können einer Anwendung zur Verwaltung von Human Tasks Aufgaben zukommen lassen. Sowohl die Tasks selbst als auch die Taskmodelle werden durch Extensible Markup Language (XML) definiert. Die dazugehörigen Schemata finden sich in der WS-Human-Task-Spezifikation [OAS A].

Im Folgenden wird auf für diese Arbeit relevante Aspekte der WS-Human-Task-Spezifikation eingegangen. Außerdem werden in den Beispielen nur die für das Konzept relevanten Bestandteile der Definition von Tasks durch XML herangezogen. Genaueres zu jedem der vorgestellten Konzepte sowie die nicht hier behandelten Aspekte kann der Spezifikation [OAS A] entnommen werden.

2.2.1. Architektur einer Anwendung nach WS Human Tasks

Um die Tasks optimal verwalten zu können, definiert die Spezifikation eine Architektur für Anwendungen mit Human Tasks. Diese ist in Abbildung 2.2 enthalten. Dabei werden drei Hauptkomponenten unterschieden: der *Task Parent*, *Task Manager (Task Prozessor)* und der *Task Client*. Der Task Parent ist die Anwendung, bei der Human Tasks anfallen. Diese leitet er an den Task Manager weiter, da dieser für die Verwaltung der Human Task verantwortlich ist. Darunter fällt unter anderem das Zuweisen von Tasks an die richtigen Personen bzw. Personengruppen, das Verwalten von Statusänderungen oder das Ergebnis an den Task Parent zurückzuleiten. Für diese Aufgaben besitzt der Task Manager *Task Definitionen*, auch Taskmodelle genannt. Taskmodelle erstellen und initiieren die Tasks. Schließlich präsentiert der Task Client dem Nutzer eine graphische Oberfläche für Tasks und Funktionen, um diese zu bearbeiten.

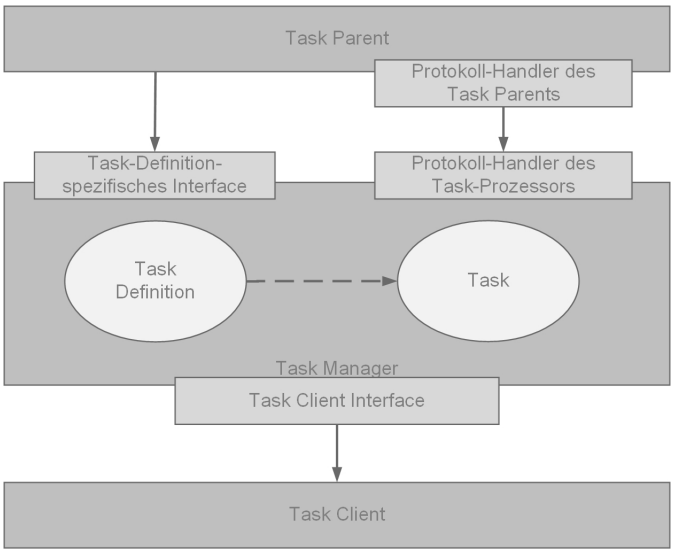


Abbildung 2.2.: Architektur einer Anwendung nach *WS Human Tasks* [OAS A]

Zuerst muss eine Task Definition zusammen mit einer für sie spezifischen Schnittstelle auf dem Task Prozessor bereitgestellt werden, damit Tasks erstellt werden können. Danach kann der Task Parent die Definition über ihre spezifische Schnittstelle ansprechen und die jeweiligen Taskdaten zukommen lassen. Der Task Parent wird somit zum Ersteller des Tasks. Daraufhin kann er über die Protokoll-Handler den Lebenszyklus des Tasks steuern. Damit ein Task schließlich bearbeitet werden kann, muss er dem Nutzer angezeigt werden. Dies geschieht über den Task Client. Für diesen Fall stellt der Task Manager ein *Task Client Interface* bereit, mit dessen Funktionen die Client-Anwendungen die nötigen Informationen abfragen und entsprechend manipulieren können. Dieses Interface ist in der Spezifikation genau standardisiert.

2.2.2. Taskattribute und Taskmodellattribute

In diesem Abschnitt wird auf die Attribute eines Tasks eingegangen. Außerdem werden in Folge dessen die wichtigsten Elemente zur Erstellung des Taskmodells vorgestellt.

Jeder Task besitzt eine Menge an Attribute, deren Werte in einem XML definiert werden. Ein Beispiel für ein solches XML ist in Abschnitt 3.3 zu finden. Bevor jedoch ein Task als XML definiert werden kann, muss zunächst ein Taskmodell erstellt werden, da jeder Task aus einem Taskmodell erzeugt wird. Dieses legt unter anderem fest, wie die Zuweisung von Personen zu bestimmten generischen Nutzerrollen aussieht. Die wichtigsten Elemente zur Erstellung eines Taskmodells sind in Tabelle 2.1 zu finden. Darauffolgend werden in Tabelle 2.2 die wichtigsten Attribute von Tasks referenziert.

Die Werte der aufgezeigten Attribute können sich im Laufe der Bearbeitung eines Tasks ändern. Dies geschieht meistens durch den Aufruf von Methoden des Task Client Interfaces. Anhand dieser Operationen können Tasks, z.B. gestartet oder beansprucht werden.

2.2.3. Darstellung von Tasks

Damit Menschen einen Task bearbeiten können, benötigen sie eine Benutzeroberfläche, welche diesen anzeigt. WS Human Task unterscheidet dafür zwei Arten von Nutzerschnittstellen. Einerseits wird für das Rendering der Meta-Informationen eines Tasks ein *Task List User Interface (UI)* definiert, andererseits wird zum Rendern des Tasks selbst eine *Task UI* bestimmt. Zweitere wird außerdem für die Ausführung des Tasks verwendet.

Die Task List UI beinhaltet eine zusammenfassende Liste bestehend aus den Meta-Informationen von ausstehenden und vollendeten Tasks. Diese Meta-Informationen umfassen z.B. Stichtage, die Priorität und eine Beschreibung über die Ausführung des Tasks. Mit anderen Worten zeigt die Task List UI eine Taskliste mit allen Informationen an, die notwendig für einen Überblick über den Task sind.

[1] siehe http://docs.oasis-open.org/bpel4people/ws-humantask-1.1-spec-cs-01.pdf

Bezeichnung	Beschreibung	Pflicht
name	Der Name muss einzigartig sein	Ja
priority	Die Priorität muss zwischen 0 und 10 liegen, wobei 0 die höchste Priorität hat.	Nein
peopleAssignments	Dieses Element wird für die Zuweisung von Personen bzw. Gruppen zu *Generic Human Roles* (vgl. 2.2.5) benutzt.	Nein
delegation	Hier werden Beschränkungen zur Weitergabe eines Tasks definiert, d.h. wem der Task weitergegeben werden kann.	Nein
presentationElements	Dieses Element enthält die Informationen zum Darstellen eines Tasks in der Benutzeroberfläche (vgl. 2.2.3)	Nein
outcome	Hier wird ein Ausdruck angegeben, der das Feld mit dem Ergebnis eines Tasks enthält.	Nein
searchBy	Zum Suchen von Tasks kann dieses benutzerdefinierte Suchkriterium genutzt werden	Nein
rendering	Dieses Element enthält die Methode, die zum Rendern eines Tasks benötigt wird.	Nein
deadlines	Hier werden verschiedene Strichtage für die Tasks des Modells definiert.	Nein
composition	Dieses Attribut spezifiziert Subtasks bei *Composite Tasks*[1]	Nein

Tabelle 2.1.: Auflistung aller wichtigen Attribute eines Taskmodells

Durch die Auswahl eines Task aus der Liste, gelangt der Nutzer in die Task UI. Diese Benutzeroberfläche stellt die gesamten Informationen des Tasks dar und gibt dem Nutzer die Möglichkeit, den Task entsprechend zu bearbeiten. Die Task UI hat also Zugang zu den Daten des Tasks und kann diese durch Aufrufen von Methoden des Task Client Interfaces ändern. Hier sollte der Nutzer zunächst die Gelegenheit haben, den Task genauer betrachten zu können, ohne sofort die Bearbeitung zu beanspruchen. Zur Auswahl für das Rendering wird eine Rendering-Methode der Task UI bei der Definition des Tasks spezifiziert. Dies geschieht durch eine einzigartige Kennung eines Attributs, die die Art des Renderings enthält. Eine Task UI kann mehrere Rendering-Methoden besitzen, z.B. für jedes Gerät, auf dem der Task angezeigt werden soll. Jedoch sollte beachtet werden, dass das eigentliche Rendering der Task UI nicht Teil des Human-Task-Standards ist.

Um die Benutzeroberflächen mit den Informationen über einen Task zu versorgen, werden in der Definition des Tasks die sogenannten *Presentation Elements* mit dem XML-Tag `<htd:presentationElements>` angegeben. Diese enthalten für den jeweiligen Task, einen Titel, eine Kurzbeschreibung und eine ausführliche Beschreibung in für Menschen lesbarer Form. Der Titel des Task wird in `<htd:name>` definiert. Dieser sollte eher kurz und prägnant gehalten werden. Ein etwas längerer Text zur Beschreibung des Tasks wird in `<htd:subject>`

hinterlegt. Schließlich folgt in `<htd:description>` die ausführliche Beschreibung des Tasks. Alle drei Elemente können in mehreren Sprachen definiert werden. Außerdem besteht für die *Description* und das *Subject* die Möglichkeit, mit `<htd:presentationParameter>` Parameter zu definieren, die zur Laufzeit mit den entsprechenden Werten ersetzt werden.

2.2.4. Generic Human Roles

Wie in Abschnitt 2.2.2 erwähnt, können unterschiedliche Aktionen auf Human Tasks ausgeführt werden. Einige dieser Operationen umfassen Änderungen im Status oder Attributen eines Tasks. Aus diesem Grund ist es sicherzustellen, dass bestimmte Aktionen nur von Personen bzw. Personengruppen ausgeführt werden, die für die jeweilige Aktion autorisiert sind. Dafür führt die WS Human Tasks sogenannte *Generic Human Roles* ein. Die folgenden generischen Nutzerrollen sind spezifiziert:

- Der *Task Initiator* ist die Person, die den Task erstellt hat.
- Ein *Task Stakeholder* ist verantwortlich für das Ergebnis des Task und behält die Übersicht über den Lebenszyklus des Tasks. Deshalb ist es ihm erlaubt, den Prozess des Tasks zu beeinflussen und administrative Aktionen auszuführen, z.B. eine Benachrichtigung bei einem verpassten Stichtag zu senden. Außerdem besteht für ihn die Möglichkeit, den Task zu beanspruchen.
- *Potential Owners* sind die Personen, die den Task für sich beanspruchen können. Bevor dies geschieht, können sie den Prozess des Tasks beeinflussen, beispielsweise durch die Änderung der Priorität. Wenn ein Potential Owner den Task für sich beansprucht, wird er zum *Actual Owner*.
- Der *Actual Owner* ist die Person, die den Task für sich beansprucht hat und ihn somit auch bearbeiten wird. Er kann verschiedene Aktionen, wie den Task an eine andere Person weitergeben oder den Task suspendieren, ausführen. Ein beanspruchter Task kann immer nur genau einen Actual Owner besitzen.
- Zu den *Excluded Owners* gehören alle Personen, die nicht Actual Owner oder Potential Owner des Tasks sind.
- *Notification Recipients* sind jene Personen, die Benachrichtigungen betreffend zum Task erhalten.
- *Business Administrators* sind genau wie die Task Stakeholder für den Task verantwortlich, jedoch gilt dies für alle Tasks des Taskmodells. Sie können die gleichen die gleichen Operationen ausführen, wie die Task Stakeholder.

2.2.5. Nutzerzuweisungen

Nachdem die verschiedenen generischen Rollen zum Ausführen bestimmter Operationen auf Tasks aufgezeigt wurden, wird in diesem Abschnitt auf die Zuweisung von Nutzern zu generischen Rollen eingegangen. WS Human Task stellt hierzu drei verschiedene Wege zur Verfügung, die im Nachfolgenden kurz erläutert werden.

Logical People Groups

Die erste Möglichkeit ist eine *Logical People Group (LPG)*, die entweder eine Person, eine Menge von Personen oder eine Gruppe repräsentiert. Dabei wird bei einer Menge von Personen jeweils der Name der beteiligten Personen und bei einer Gruppe ein Gruppenname hinterlegt. Zur Initialisierungszeit wird die LPG an eine oder mehrere Anfrage(n) gegen ein Personenverzeichnis gebunden. Dadurch kann eine Anfrage zur Laufzeit ausgewertet werden und die Personen, für die der Task relevant ist, werden zurückgegeben. Da LPGs Parameter in der Anfrage unterstützen, welche Referenzen zu einem bestimmten Task enthalten können, werden die eigentlichen Werte erst zur Laufzeit an die Anfrage übergeben.

Ein Beispiel für eine LPG zum Zuweisen von Personen könnte in etwa wie in Listing 2.1 aussehen.

Listing 2.1 Beispiel einer Personenzuweisung über *Logical People Groups*

```
1  <htd:logicalPeopleGroups>
2    <htd:logicalPeopleGroup name="projectManager">
3      <htd:parameter name="project" type="xsd:string" />
4    </htd:logicalPeopleGroup>
5  </htd:logicalPeopleGroups>
6
7  <htd:task name="ApproveBuyRequest">
8  ....
9    <htd:potentialOwners>
10     <htd:from logicalPeopleGroup="projectManager">
11           <htd:argument name="project">
12             htd:getInput("buyRequest")/projectName
13           </htd:argument>
14     </htd:from>
15   </htd:potentialOwners>
16 ....
17 <htd:task>
```

Im vorliegenden Beispiel sollen die jeweiligen Projektmanager als Potential Owners für einen Task zugewiesen werden, der ihre Zustimmung zum Kauf eines Bauteils fordert. Dafür wird am Beginn des XML-Ausschnitts eine LPG `projectManager`, die alle Projektmanager umfasst, definiert. Außerdem wird ein Parameter `project` des Typs String definiert. Nun kann die unterliegende Infrastruktur die Zuordnung der Personen vornehmen. Als

nächstes werden im Task, mit dem Tag `<htd:from>` und dem Attribut `logicalPeopleGroup`, die Projektmanager als `potentialOwners` festgelegt. Der Parameter `project` wird mit `getInput("buyRequest")` aus der *Web Service Description Language (WSDL)*[2] des Task mit einem *XPath*[3] Ausdruck ausgelesen.

Literale

Ein weiteres Vorgehen, um Personen einer Generic Human Role zuzuweisen sind *Literale*. Hierfür wird explizit die Nutzerkennung oder der Name einer Personengruppe benutzt, um Personen bzw. Personengruppen einer Rolle zuzuordnen. Dadurch ist dieser Ansatz etwas unflexibler als LPG, da schon zur Entwurfszeit Nutzerkennungen und Namen der Personengruppen spezifiziert sein müssen.

In Listing 2.2 ist das Beispiel der LPGs auf die Zuweisung durch Literale übertragen.

Listing 2.2 Beispiel einer Personenzuweisung über *Literale*

```
<htd:task name="ApproveBuyRequest">
....
  <htd:potentialOwners>
    <htd:from>
      <htd:literal>
        <htd:organizationalEntity>
          <htd:users>
            <htd:user>Sam</htd:user>
            <htd:user>Dean</htd:user>
            <htd:user>Castiel</htd:user>
          </htd:users>
        </htd:organizationalEntity>
      </htd:literal>
    </htd:from>
  </htd:potentialOwners>
....
<htd:task>
```

Hier werden nun die Projektmanager direkt mit ihrem Namen im XML-Dokument als Potential Owners angegeben, in diesem Fall Sam, Dean und Castiel. Dafür wird wie schon zuvor `<htd:from>` benutzt, jedoch muss nun eine `organizationalEntity` verwendet werden. Diese `organizationalEntity` besitzt entweder das Element `<htd:users>`, das eine Liste von Usernamen enthält, oder das Element `<htd:groups>`, das entsprechend eine Liste mit Gruppennamen enthält.

[2] siehe https://www.w3.org/TR/wsdl20/

[3] XML Path Language - dient zum Adressieren von Knoten in XML-Dokumenten (https://www.w3.org/TR/xpath/)

Ausdrücke

Der letzte Weg der Zuweisung sind Ausdrücke, die zur Laufzeit auf Daten von Tasks referenzieren. Daher geben sie entweder eine User- oder Gruppenliste zurück.

Das auf Ausdrücke übertragene Beispiel ist in Listing 2.3 gegeben.

Listing 2.3 Beispiel einer Personenzuweisung über *Ausdrücke*

```
1 <htd:task name="ApproveBuyRequest">
2 ....
3   <htd:potentialOwners>
4     <htd:from>
5       htd:getInput("buyRequest")/projectManager
6     </htd:from>
7   </htd:htd:potentialOwners>
8 ....
9 <htd:task>
```

In diesem XML-Ausschnitt wird `buyRequest` vom Input des Tasks ausgewählt und mit `projectManager` die Namen der jeweiligen Projektmanager zurückgegeben. Hier geschieht das Ganze über einen XPath-Ausdruck. Der Wert, der durch den Ausdruck erhalten wird, wird explizit als Name der Projektmanager benutzt. Also wird keine Anfrage wie bei LPGs durchgeführt.

Bezeichnung	Beschreibung
id	Mit diesem Attribut wird der Task identifiziert. Der Wert muss einzigartig sein.
name	Hier wird der Name des Taskmodells angegeben, von dem der Task abstammt.
status	Dieses Attribut enthält den aktuellen Status des Tasks.
priority	Dieses Element definiert die Priorität des Tasks basierend auf der Angabe im Taskmodell.
task initiator	Die Zuordnung von Personen zum *Task Initiator* wird hier definiert. Dies geschieht über UserIDs.
task stakeholders	Die Zuordnung von Personen zu *Task Stakeholders* wird hier definiert. Dies geschieht über UserIDs.
potential owners	Die Zuordnung von Personen zu *Potential Owners* wird hier definiert. Dies geschieht über UserIDs.
business administrator	Die Zuordnung von Personen zu *Business Administrator* wird hier definiert. Dies geschieht über UserIDs.
actual owner	Wenn ein Task beansprucht wurde, dann wird hier die UserID des *Actual Owners* angegeben.
created time	Hier werden das Datum und die Uhrzeit des Task definiert, zu der der Task in den Status *created* übergegangen ist. Der Zeitpunkt wird in Coordinated Universal Time (UTC) angegeben.
activation time	Hier werden das Datum und die Uhrzeit des Task definiert, zu der der Task in den Status *ready* übergegangen ist. Der Zeitpunkt wird in UTC angegeben.
expiration time	Hier werden das Datum und die Uhrzeit des Task definiert, zu der der Task in den Status *obsolete* übergehen wird. Der Zeitpunkt wird in UTC angegeben.
is skippable	Dieses Element zeigt an, ob ein Task übersprungen werden kann, wenn die Ausführung nicht mehr nötig ist.
input data	Hier sind Informationen angegeben, die zum Bearbeiten des Tasks benötigt werden.
output data	Dieses Attribut enthält die Daten, die nach Ausführen des Tasks als Resultat entstehen.
presentation information	Dieses Element enthält die Informationen über den Task, die auf der Benutzeroberfläche angezeigt werden sollen. Diese umfassen einen Titel, einen Betreff und eine Beschreibung.

Tabelle 2.2.: Auflistung aller wichtigen Attribute eines Tasks

3. Human Tasks für OpenTOSCA zum Aufsetzen von IoT-Anwendungen

In diesem Kapitel wird das erarbeitete Konzept für die Ausführung von Human Tasks, die beim Aufsetzen von IoT-Anwendungen mit OpenTOSCA anfallen, vorgestellt. Dieses Konzept basiert auf der Spezifikation WS Human Tasks [OAS A]. Jedoch wurden einige Anpassungen und Erweiterungen für den Anwendungsfall aus Abschnitt 1.1 vorgenommen, auf die in den nächsten Abschnitten eingegangen wird. Als Erstes wird eine Einführung in das vorliegende Konzept gegeben. Des Weiteren wird der Aufbau des Task Managers, der mit einer Representational State Transfer (REST)-API erweitert wurde, vorgestellt. Um die REST-API optimal anzusprechen, werden die nötigen Task- und Userdaten von XML zu JavaScript Object Notation (JSON) konvertiert. Danach werden die Funktionen der REST-API definiert und auf die Darstellung der Tasks in einer Smartphone-App eingegangen. Zum Schluss werden einige Abgrenzungen und Entscheidungen erläutert.

3.1. Überblick über das Konzept

In Abbildung 3.1 werden die Komponenten des Konzepts und ihre Beziehungen untereinander dargestellt. Im Mittelpunkt steht der Task Manager, der die Tasks, Nutzer und Rollen verwaltet und in einer Datenbank speichert. Er ist über eine REST-API ansprechbar. Diese wird vorrangig von OpenTOSCA und einer Smartphone-App genutzt, aber auch externe Systeme können sie ansprechen. OpenTOSCA erzeugt dabei die Tasks, die die Smartphone-App dem Nutzer zur Bearbeitung präsentiert.

Die Architektur wird größtenteils übernommen, wie sie in der WS-Human-Task-Spezifikation beschrieben ist. So entspricht OpenTOSCA dem Task Parent, der Task Manager dem Task Prozessor und die Smartphone-App dem Task Client. Während des Aufsetzens einer IoT-Anwendung mit OpenTOSCA fallen die Tasks an, welche über die REST-API an den Task Manager weitergeleitet werden (vgl. Abschnitt 3.5). Dieser weist den Task den jeweiligen Rollen des übergebenen *Task Types* zu und speichert alle Informationen zum Task. Die Smartphone-App kann über die REST-API die für den Nutzer relevanten Tasks abfragen und diese anzeigen. Nach der Bearbeitung schickt die Smartphone-App die entsprechenden Daten über die REST-API zurück an den Task Manager. OpenTOSCA holt sich die Ergebnisse des vollständigen Tasks ebenfalls über die REST-API und kann das Aufsetzen der IoT-Anwendung fortsetzen.

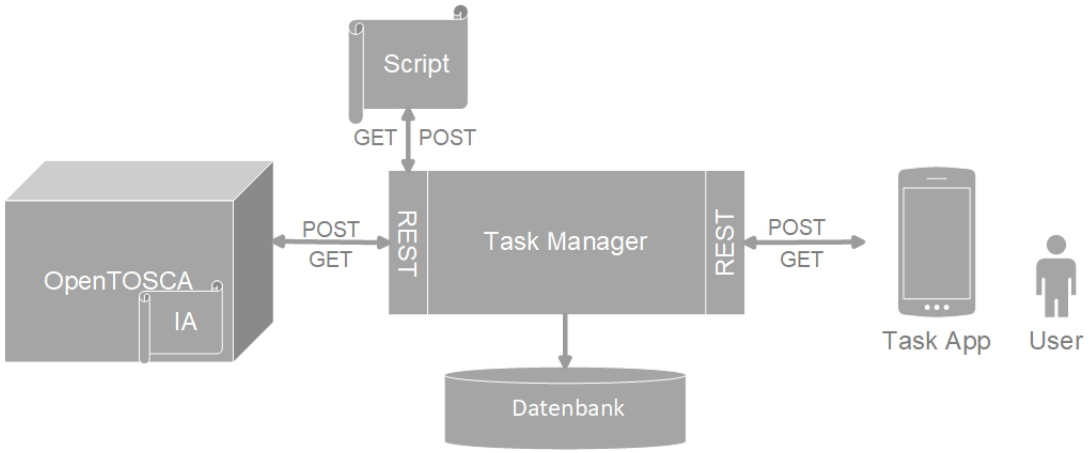


Abbildung 3.1.: Architektur des gesamten Systems

Es gibt zwei Unterschiede zu der in Abschnitt 2.2.1 vorgestellten Architektur. Erstens leitet der Task Manager nicht automatisch die Ergebnisse eines vollständigen Tasks an OpenTOSCA weiter. Stattdessen holt sich OpenTOSCA mit der entsprechenden API-Funktion die Informationen beim Task Manager. Ein Vorteil hierbei ist, dass OpenTOSCA jederzeit den aktuellen Stand des Task abrufen und somit die Bearbeitung besser überwachen kann. Zweitens gibt es im vorliegenden System einen Task Parent, wodurch der Task direkt im Task Manager abgespeichert werden kann, ohne eine Zuordnung zu einem elternspezifischen Taskmodell durchführen zu müssen. Aus diesem Grund ist es nicht zwingend notwendig, den Task Parent bei der weiteren Bearbeitung eines Tasks zu berücksichtigen.

Für das vorliegende Konzept werden nicht alle der in WS Human Task geforderten Funktionen des Task Client Interfaces umgesetzt. Es wird sich nur auf die für den Anwendungsfall nötigen Funktionen, z.B. Funktionen zum Abrufen der *Presentation Details* und zum Beanspruchen des Tasks, konzentriert.

Im Folgenden werden Begrifflichkeiten und Aspekte vorgestellt, die für das Konzept wichtig sind. Darunter fallen die erwähnten Task Types, aber auch die Einführung von *Input-* und *Output-Parametern* eines Tasks.

User-Management

Für die Nutzer wird eine User-Datenbank im Task Manager bereitgestellt. Diese speichert die wichtigsten Informationen über die Nutzer, sowie eine einzigartige UserID. Anhand dieser UserID kann der Nutzer vom Task Manager identifiziert werden. Neben Nutzern werden auch Rollen hinterlegt, denen Nutzer zugeteilt werden. Bei der Erstellung werden diese Rollen schließlich den Generic Human Roles zugeordnet.

Task Types

Task Types dienen dazu, Tasks in Kategorien einzuordnen. Bei der Erstellung werden sie als Attribut mitgegeben und dazu genutzt, den Nutzern Tasks für die Bearbeitung zuzuweisen. Damit dies funktioniert, werden jedem Task Type Rollen zugeordnet. Der Task wird den zugehörigen Nutzern der jeweiligen Rollen zur Bearbeitung angezeigt. Einer dieser Nutzer kann den Task für sich beanspruchen und ihn bearbeiten. Task Types und ihre Rollen werden ebenfalls in der Datenbank des Task Managers hinterlegt.

Presentation Details

Presentation Details sind Informationen über die Darstellung eines Tasks und werden in der Task-Datenbank gespeichert. Diese umfassen den Titel, eine kurze Beschreibung und eine ausführliche Aufgabenstellung des Tasks. Die Kurzbeschreibung soll aufzeigen, um welche Aufgabe es sich bei dem Task handelt. Bei der Darstellung werden die Elemente in der Reihenfolge Titel, kurze Beschreibung und ausführliche Beschreibung angezeigt (vgl. Abschnitt 3.6). In der WS-Human-Task-Spezifikation werden außerdem *Presentation Parameter* definiert. Dabei handelt es sich um Werte, die Platzhalter innerhalb der Kurzbeschreibung und der Aufgabenstellung füllen. Dies wird in diesem Konzept nicht unterstützt. Stattdessen werden die Parameter als Input-Parameter unterhalb der Aufgabenstellung aufgelistet.

Input-Parameter

In der WS-Human-Task-Spezifikation werden Parameter, die für die Bearbeitung des Tasks notwendig sind, in einer *Input-Message* hinterlegt. Diese wird zur Laufzeit ausgelesen und in Platzhalter innerhalb der Presentation Elements eingefügt (vgl. Abschnitt 3.3). Für dieses Konzept werden diese Parameter als Input-Parameter bezeichnet und dienen z.B. dazu, die genaue Bezeichnung eines Sensors anzugeben. Ein Input-Parameter besitzt einen allgemeinen Bezeichner und einen spezifischen Wert. In der Smartphone-App werden sie unterhalb der Presentation Details in einer Auflistung angezeigt. Sie werden verwendet, um dem Nutzer zusätzliche Informationen zu liefern.

Output-Parameter

Analog zur Input-Message existiert eine *Output-Message* innerhalb der WS-Human-Task-Spezifikation. Diese dient dazu, dem Nutzer die Möglichkeit zu bieten, für die Bearbeitung des Tasks notwendige Eingaben zu hinterlegen. In diesem Kontext werden diese Parameter als Output-Parameter bezeichnet. Auch Output-Parameter besitzen einen Bezeichner, der die benötigte Information benennt, und einen Wert, welcher vom Nutzer bei der Bearbeitung angegeben wird (vgl. Abschnitt 3.3). Für die Darstellung werden Eingabefelder bereitgestellt,

die vom Nutzer ausgefüllt werden können und nach Beendigung des Tasks an den Task Manager zurückgeleitet werden. OpenTOSCA kann diese schließlich über die REST-API abfragen.

3.2. Aufbau auf des Task Managers

Für den Task Manager wird eine bereits existierende Implementierung benutzt, die jedoch in einigen Punkten erweitert und abgeändert wird. In diesem Abschnitt wird der benutzte Task Manager vorgestellt sowie auf die Erweiterungen und Änderungen bezüglich des vorliegenden Konzepts eingegangen.

Der Task Manager wurde im Rahmen der Diplomarbeit [Wag10] von Sebastian Wagner am Institut für Architektur von Anwendungssystemen der Universität Stuttgart entwickelt. Der Task Manager implementiert das in der Arbeit vorgestellte Konzept eines Menschen-orientierten Arbeitsflusses. Hierfür wurden nicht nur Human Tasks betrachtet, sondern auch, wie ein solcher Arbeitsfluss optimal gestaltet und in ein bestehendes *Workflow Management System* eingebunden werden kann.

Die Implementierung des Task Managers basiert auf der WS-Human-Task-Spezifikation. Aus diesem Grund ist er für diese Arbeit als Task Prozessor geeignet. Jedoch muss beachtet werden, dass einige Anpassungen vorgenommen wurden. Unter anderem wurde das Konzept von *Work Items* eingeführt. Ein Work Item ist eine Menge an Operationen, die Nutzer auf dem Task ausführen dürfen. Diese Operationen werden nach der Generic Human Role des Nutzers ausgewählt. Somit kann ein Task mehrere Work Items besitzen. Außerdem werden die Nutzer nicht dem Task, sondern dem Taskmodell zugewiesen. Jeder Task Parent definiert ein Taskmodell. Dadurch ist jeder Task eines Taskmodells mit den gleichen Personen verbunden [Wag10].

Im Rahmen dieser Bachelorarbeit werden Erweiterungen und Änderungen des Task Managers benötigt. Die Implementierung von [Wag10] ist kein eigenständiges Modul, sondern benötigt ein Workflow Management System, in das sie integriert wird. Dieses Konzept fordert jedoch ein eigenständiges Modul. Deshalb muss die Implementierung angepasst werden, sodass sie in einer eigenen Serverumgebung bereitgestellt werden kann (vgl. Abschnitt 4.2.3). Weiterhin muss dieser eigenständige Task Manager angesprochen werden können. Dafür wird er um eine REST-API erweitert. Diese besitzt die Funktionen, die zur Ausführung nötig sind. Weitere Informationen folgen in Abschnitt 3.5.

Durch die Einführung der Task Types und der daraus resultierenden Zuweisung eines Task zu den Rollen, die dem Task Type zugeordnet sind, wird nur ein Work Item pro Task erstellt. Dieses Work Item enthält die Informationen, ob der Task beansprucht wurde und die UserID des Nutzers, der den Task beansprucht hat.

3.3. Vergleich zwischen XML- und JSON-Repräsentation

In WS Human Tasks werden Tasks als XML modelliert. Diese Modellierung lässt sich auch auf andere Formate übertragen. Eines davon wäre JSON, welches von der REST-API benutzt wird. Dabei erhält die REST-API ihre benötigten Daten in Form eines JSON-Objekts und antwortet dementsprechend ebenfalls mit JSON-Objekten. Aus diesem Grund wird in diesem Abschnitt darauf eingegangen, wie die Definition eines Tasks von XML zu JSON konvertiert werden kann.

Sowohl XML als auch JSON sind von Menschen und Maschinen lesbar. Mit XML lassen sich durch die Definition von *Namespaces* auch komplexe Datenstrukturen darstellen und übertragen. Dadurch hat XML mehr Erweiterungsmöglichkeiten, lässt sich allerdings nicht kompakt darstellen und behandeln. Zum einen müssen Tags zum Öffnen und Schließen sowie Namespaces bei der Nutzung von XML vollständig angegeben werden und zum anderen verlangt dies zusätzlichen Quellcode, den der Programmierer bereitstellen muss. JSON dagegen besitzt nicht die Möglichkeit, Namespaces für die Struktur der darzustellenden Daten zu definieren, weswegen JSON nicht die Erweiterungsmöglichkeiten von XML bietet. Daten werden durch ein Schlüssel-Wert-Paar dargestellt. Dies sorgt dafür, dass JSON-Repräsentationen um einiges kompakter dargestellt werden können (vgl. Listing 3.1). JSON ist leichtgewichtiger als XML, aber diese Leichtgewichtigkeit genügt, um Funktionen über eine REST-API Parameter zu übergeben. Außerdem ist JSON durch diese Leichtgewichtigkeit performanter als XML. Dies wirkt sich auch auf die Smartphone-App aus. Das Auslesen von Daten aus JSON-Objekten ist ressourcenschonend, wodurch sich der Batterieverbrauch verringert, was sich wiederum positiv auf die Performanz der Smartphone-App auswirken kann. Aus den beschriebenen Gründen und der Tatsache, dass JSON sowohl für Menschen und Maschinen lesbar als auch leicht zu generieren ist, wurde sich für JSON entschieden [GT11; Nur+; ECMce].

Die XML-Repräsentation eines Tasks, basierend auf der WS-Human-Task-Spezifikation, ist in Listing 3.2 dargestellt. Die Repräsentation in JSON befindet sich inListing 3.1.

Einige Teile der XML-Repräsentation werden für die JSON-Repräsentation übernommen, andere werden abgeändert und neu hinzugefügt. Ein Unterschied ist, dass in der JSON-Repräsentation die Werte aller Schlüssel direkt angegeben werden müssen, während es in XML die Möglichkeit gibt, die Inhalte der Tags mit einem Ausdruck, z.B. XPath, zu referenzieren und diese zur Laufzeit dynamisch auszulesen. Da bei der Nutzung einer Methode der REST-API die Werte feststehen, können diese direkt dem jeweiligen Schlüssel zugeordnet werden.

Bei der Umwandlung zu JSON wurden die Tags `name` und `priority` direkt als JSON-Schlüssel übernommen. Für die Repräsentation der Presentation Details wird `presentationElements` und die darin enthaltenen Tags auf ein JSON-Objekt `presentationDetails` gemappt, wobei `name` zu `title` umbenannt wird. Die im XML enthaltenen `presentationParameters` werden je nach Art entweder zu Input-Parametern (`inputParameters`) oder Output-Parametern

(outputParameters). InputParameters enthält ein JSON-Array, welches für jeden Input-Parameter des Tasks ein JSON-Objekt mit den Schlüsseln label und value besitzt. Dabei steht label für den Bezeichner des Parameters und value für den Wert. OutputParameters werden analog angegeben, allerdings wird hier value mit “ “ initialisiert. Nach der Bearbeitung des Tasks wird der vom Nutzer eingegebene Wert eingefügt.

In dem vorliegenden Konzept wird die Weitergabe von Tasks nicht berücksichtigt, weswegen delegation nicht in die JSON-Repräsentation übernommen wird. peopleAssignments wird ebenfalls nicht übernommen, da die Zuweisung der User derzeit nur über Task Types stattfindet.

Die Task Types werden in Form eines JSON-Arrays mit dem Schlüssel taskTypes angegeben. Zusätzlich wird der Schlüssel claimedBy eingeführt. Dieser enthält die UserID des Nutzers, der den Task beansprucht hat. Sollte dies noch kein User getan haben, wird der Wert frei gelassen. Zuletzt existiert in der JSON-Repräsentation der Schlüssel status. Dieser enthält den aktuellen Status des Tasks. So kann erkannt werden, ob ein Task bereits fertig bearbeitet wurde.

Listing 3.1 Beispiel einer Task Darstellung in JSON

```
{
  "id" : "86",
  "name" : "enableSSH",
  "taskTypes" : ["hardware"],
  "status" : "inProgress",
  "claimedBy" : "Sam83",
  "outputParameters" : [
    { "label" : "SSH Key",
      "value" : " " },
    { "label" : "Name of device",
      "value" : " " }
  ],
  "presentationDetails" : {
    "title" : "SSH connection",
    "subject" : "Activate SSH connection",
    "description" : "Please enable the SSH connection for
    the Raspberry Pi and provide name of the device."
  }
}
```

Listing 3.2 Beispiel einer Task Darstellung in XML

```
<htd:task name="enableSSH">
  <htd:documentation xml:lang="en-US">
    This task is used to enable the SSH connection of the Raspberry Pi.
  </htd:documentation>
  <htd:interface portType="af:enableSSHPT"
    operation="done"
    responsePortType="af:enableSSHCallbackPT"
    responseOperation="enableSSHResponse"/>
  <htd:priority>
    htd:getInput("EnableSSHRequest")/prio
  </htd:priority>
  <htd:peopleAssignments>
    <htd:potentialOwners>
      <from>
        <htd:literal>
          <htt:organisationalEntity>
            <htt:group>hardwareConfigurator</htt:group>
          </htt:organisationalEntity>
        </htd:literal>
      </from>
    </htd:potentialOwners>
    <htd:businessAdministrators>
      <from>
        <htd:literal>
          <htt:organisationalEntity>
            <htt:group>admin</htt:group>
          </htt:organisationalEntity>
        </htd:literal>
      </from>
    </htd:businessAdministrators>
  </htd:peopleAssignments>
  <htd:delegation potentialDelegation="nobody"/>
  <htd:presentationElements>
    <htd:name xml:lang="en-US">SSH connection</htd:name>
    <htd:presentationParameters>
      <htd:presentationParameter name="sshKey"
        type="xsd:string">
        htd.getOutput("EnableSSHRequest")/key
      </htd:presentationParameter>
      <htd:presentationParameter name="nameOfDevice"
        type="xsd:string">
        htd.getOutput("EnableSSHRequest")/device
      </htd:presentationParameter>
    </htd:presentationParameters>
    <subject xml:lang="en-US">
      Activate SSH connection
    </subject>
    <htd:description xml:lang="en-US" contentType="text/html">
      <![CDATA[
        Please enable the SSH connection for the Raspberry Pi and provide name of the
            device.
      ]]>
    </htd:description>
  </htd:presentationElements>
</htd:task>
```

3.4. Statusänderungen eines Tasks

In WS Human Tasks werden einige Zustände für Tasks eingeführt, welche in Abbildung 3.2 gegeben sind. Ein Task durchläuft diesen Lebenszyklus ab seiner Erstellung bis zur Fertigstellung. Zustände können sich entweder direkt über das Ansprechen der REST-API oder im Zuge des Ausführens bestimmter Methoden innerhalb der Bearbeitung des Tasks ändern.

Da einerseits die Implementierung des Task Managers bereits Änderungen an diesem Lebenszyklus vorgenommen hat und andererseits Änderungen bezüglich des Anwendungsfalls vorgenommen werden, wird in diesem Abschnitt auf die geänderten Zustandsübergänge eingegangen. Zuerst werden jedoch die wichtigsten Zustandsänderungen anhand der in WS Human Task definierten Zustandsübergänge dargestellt. Danach wird der veränderte Ablauf vorgestellt.

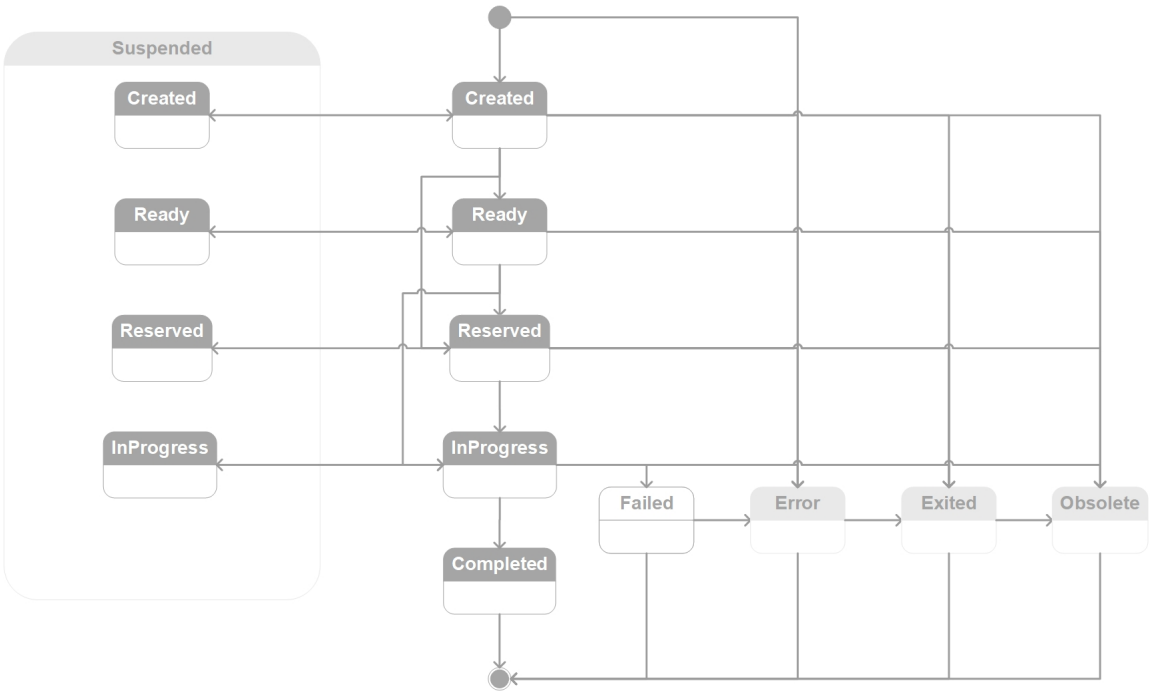


Abbildung 3.2.: Zustände nach [OAS A]

In Abbildung 3.2 sind alle Zustände der WS Human Task zu sehen. Wenn ein Task erstellt bzw. instanziiert und die Zuweisung der Personen für diesen Task vorbereitet wird, wird der Task in den Zustand `created` gesetzt. Danach werden dem Task seine *Potential Owners* zugewiesen. Dabei ändert sich der Zustand zu `ready`. Wenn ein Nutzer den Task für sich beansprucht, aber noch nicht mit der Bearbeitung begonnen hat, ist der Task `reserved`. Beginnt der Nutzer mit der Bearbeitung, wird der Task in den Zustand `inProgress` versetzt. Ist der Nutzer mit der Bearbeitung fertig, geht der Task in `completed` über. Sollte es allerdings zu einem Fehler während der Ausführung kommen, erhält der Task den Zustand `failed`. Falls der Nutzer den Task während der Bearbeitung freigibt, geht der Task wieder

in ready über, bis ein anderer Nutzer ihn beansprucht. In den Zuständen created, ready, reserved und inProgress kann ein Task suspendiert werden und wird in den entsprechenden suspended-Status gesetzt. Sollte der Task einen Stichtag besitzen, der vor Beendigung der Ausführung eintritt, landet der Task in obsolete [OAS A].

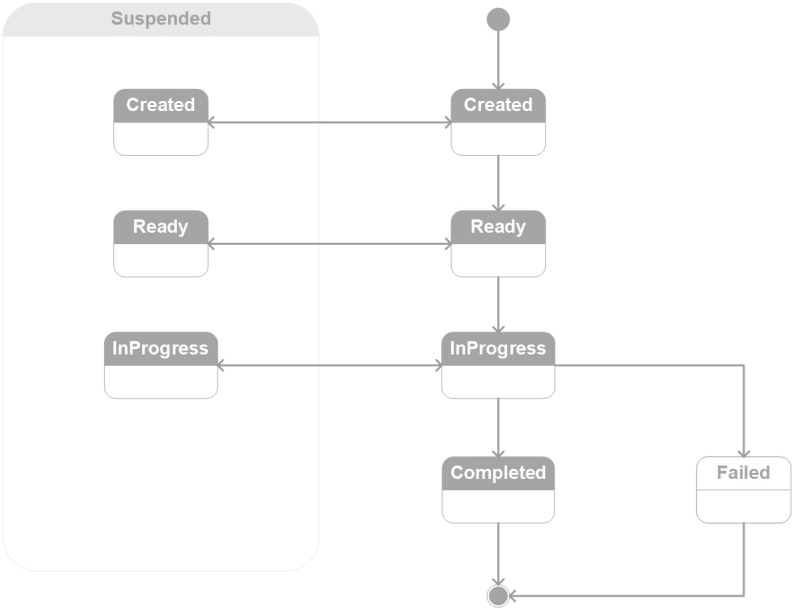


Abbildung 3.3.: Zustände des Task Manager

Aufgrund des Anwendungsfalls und den bereits bestehenden Änderungen durch die verwendete Implementierung des Task Managers wird hier ein abgewandelter Lebenszyklus für Tasks verwendet. Dieser ist in Abbildung 3.3 gegeben. Im Folgenden wird auf die finalen Zustandsänderungen eingegangen. Informationen zu den vorangegangenen Änderungen durch die Implementierung des Task Managers sind in [Wag10] zu finden.

Der Initialzustand eines Tasks ist immer noch created. Jedoch wird, direkt mit der Erstellung, der Task über die Task Types den Rollen zur potenziellen Bearbeitung zugewiesen (vgl. Abschnitt 3.7). Deshalb geht der Task erst mit der Beanspruchung durch einen Nutzer in ready über. Wenn der Task gestartet wurde, wird dieser in inProgress gesetzt. Im Gegensatz zur WS-Human-Task-Spezifikation ist es hier nicht möglich, dass ein Nutzer den Task wieder freigeben kann. Also kann der Task nicht in den ready-Zustand zurückkehren. Nach der Bearbeitung landet der Task im completed-Status. Falls ein Fehler auftritt, ist failed der finale Zustand. Auch hier gibt es wieder suspended-Zustände.

3.5. Funktionen der REST API

Wie zuvor erwähnt, erhält der Task Manager eine REST-API. Diese dient dazu, die Funktionen des Task Managers für OpenTOSCA und die Smartphone-App zugänglich zu machen. REST hat den Vorteil, leichtgewichtiger zu sein als vergleichbare Architekturen.

Außerdem ist REST universell einsetzbar, da es mittlerweile eine Vielzahl von Bibliotheken gibt, die REST in Form von HTTP implementieren. Dadurch können die Komponenten plattformunabhängig kommunizieren [FT02; Fen+09].

Der Task Manager muss für die Bearbeitung der Tasks Funktionen zur Verfügung stellen, die es OpenTOSCA und der Smartphone-App erlauben, Tasks zu erstellen, abzuändern, abzurufen und zu löschen. Darüber hinaus sollen Funktionen zur Nutzerverwaltung unterstützt werden. Darunter fallen das Erstellen von Nutzern und Rollen sowie die Möglichkeit, Nutzer zu Rollen zuzuweisen und abzufragen.

In den nächsten Abschnitten werden jeweils die nötigen API-Funktionen für das User- und das Task-Management definiert. Diese umfassen PUT-, GET-, POST- und DELETE-Methoden für Rollen, Nutzer, Task Types und Tasks. Parameter werden in allen Funktionen entweder als Pfad-Parameter oder als JSON-Objekt übergeben.

3.5.1. User-Management

In diesem Abschnitt werden die Funktionen vorgestellt, die für die Nutzerverwaltung benötigt werden. Nutzer und deren Rollen können über die REST-API angelegt und verwaltet werden. Das User-Management wird in zwei APIs unterteilt. Zum einen die *roles-API*, welche Funktionen zu den Rollen enthält, und zum anderen die *users-API*, die entsprechende Funktionen für die Nutzerverwaltung besitzt. Im Folgenden werden beide APIs und ihre Funktionen vorgestellt.

Rollen

Die roles-API unterstützt Funktionen zum Erstellen, Updaten und Löschen von Rollen. Außerdem besteht die Möglichkeit, die zu einer Rolle zugeordneten Task Types und Nutzer abzufragen. Funktionen, die Parameter benötigen, erhalten diese über ein JSON-Objekt, das folgende Informationen enthalten kann:

- `id`: Bezeichner der Rolle, der bei der Erstellung vom Task Manager generiert wird
- `rolename`: Einzigartige Bezeichnung der Rolle
- `genericHumanRoles`: Generic Human Roles, die auf die Rolle gemappt werden, möglich sind:
  - `potential_owner`
  - `actual_owner`
  - `excluded_owner`
  - `task_initiator`
  - `task_stakeholder`
  - `business_administrator`
  - `anybody`

Listing 3.3 Beispiel eines JSON-Objekts einer Rolle

```
1 {
2   "id" : "389479",
3   "rolename" : "hardwareConfigurator",
4   "genericHumanRoles" : ["potential_owner"]
5 }
```

Ein Beispiel einer Rolle als JSON-Objekt ist in Listing 3.3 zu finden.

Die roles-API unterstützt die nachfolgenden Funktionen:

- *PUT /roles*
  Parameter: JSON-Objekt mit `rolename` und `genericHumanRoles`
  Diese Methode erstellt eine neue Rolle mit den mitgelieferten Attributen. Nach Ausführung wird die generierte ID der neu erstellten Rolle zurückgegeben.
- *GET /roles/*{*role*}
  Pfad-Parameter: {role}: Name der Rolle, deren Werte abgefragt werden sollen
  Diese Methode gibt die Werte zur angefragten Rolle als JSON-Objekt zurück.
- *GET /roles*
  Es werden alle vorhandenen Rollen in einem JSON-Array zurückgegeben.
- *GET /roles/*{*role*}*/user*
  Pfad-Parameter: {role}: Name der Rolle, deren User zurückgegeben werden sollen
  Diese Methode gibt alle Nutzer der angefragten Rolle als JSON-Objekt zurück.
- *GET /roles/*{*role*}*/taskType*
  Pfad-Parameter: {role}: Name der Rolle, deren Task Types zurückgegeben werden sollen
  Diese Methode gibt alle Task Types der angefragten Rolle als JSON-Objekt zurück.
- *POST /roles/*{*role*}
  Pfad-Parameter: {role}: Name der Rolle, die geupdatet werden soll
  Parameter: JSON-Objekt mit `rolename` und `genericHumanRoles`
  Diese Methode führt ein Update der Rolle mit dem angegebenen Namen durch. Hierfür werden die im JSON-Objekt mitgegebenen Daten verwendet.
- *DELETE /roles/*{*role*}
  Pfad-Parameter: {role}: Name der Rolle, die gelöscht werden soll
  Diese Methode löscht die Rolle mit dem angegebenen Namen.

User

Entsprechend zur roles-API enthält die users-API Funktionen zum Erstellen, Updaten und Löschen von Nutzern. Für das JSON-Objekt werden dabei die folgenden Schlüssel unterstützt:

- `id`: Bezeichner des Nutzers, welcher vom Task Manager bei der Erstellung generiert wird
- `firstname`: Vorname des Nutzers
- `lastname`: Nachname des Nutzers
- `userId`: Einzigartiger Nutzername des Nutzers
- `roles`: Rollen des Nutzers

Ein Beispiel für den Einsatz in einem JSON-Objekt ist in Listing 3.4 zu finden.

Listing 3.4 Beispiel eines JSON-Objekts eines Users

```
1 {
2   "id" : "49759",
3   "firstname": "Dean",
4   "lastname" : "Winchester",
5   "userId" : "Dean79",
6   "roles" : ["admin", "hardwareConfigurator"]
7 }
```

Im Einzelnen sind in der API folgende Methoden enthalten:

- *PUT /users*
  Parameter: JSON-Objekt mit `firstname`, `lastname`, `userId` und `roles`
  Diese Methode erstellt einen neuen Nutzer mit den mitgelieferten Attributen und weist ihm die Rollen aus `roles` zu. Nach Ausführung wird die generierte ID des neu erstellten Nutzers zurückgegeben.
- *GET /users/{user}*
  Pfad-Parameter: {user}: UserID des Nutzers, dessen Werte abgefragt werden sollen
  Diese Methode gibt die Werte zum angefragten Nutzer als JSON-Objekt zurück.
- *GET /users*
  Es werden alle vorhandenen Nutzer in einem JSON-Array zurückgegeben.
- *POST /users/{user}*
  Pfad-Parameter: {user}: UserID des Nutzers, der geupdatet werden soll
  Parameter: JSON-Objekt mit `firstname`, `lastname`, `userId` und `roles`
  Diese Methode führt ein Update des Nutzers mit der angegebenen UserID durch und weist ihm die Rollen aus `roles` zu. Hierfür werden die im JSON-Objekt mitgegebenen Daten verwendet.

- *DELETE /users/{user}*
  Pfad-Parameter: {user}: UserID des Nutzers, der gelöscht werden soll
  Diese Methode löscht den Nutzer mit der angegebenen UserID.

3.5.2. Task-Management

Im Folgenden wird auf die Funktionen für das Task-Management eingegangen. Die bereitgestellten Methoden dienen hauptsächlich dem Erstellen und dem Updaten der Tasks. Außerdem können Task Types verwaltet werden. Hier werden die Funktionen in die *taskTypes-API*, für Task Types betreffende Methoden, und in die *tasks-API*, für Tasks betreffende Methoden, aufgeteilt.

Task Types

Die taskTypes-API besitzt Methoden, die Task Types erstellen, updaten und löschen können. Analog zu den bisher vorgestellten Funktionen der anderen APIs werden Parameter als JSON-Objekte übergeben. Ein Beispiel für die Definition eines Task Types in JSON ist in Listing 3.5 gegeben. Hierfür werden die folgenden JSON-Schlüssel unterstützt:

- `id`: Bezeichner des Task Types, welcher vom Task Manager bei der Erstellung generiert wird
- `typename`: Einzigartige Bezeichnung des Task Types
- `associatedRoles`: Rollen, die auf den Task Type gemappt werden

Listing 3.5 Beispiel eines JSON-Objekts eines Task Types

```
1 {
2   "id" : "98598",
3   "typename" : "hardware",
4   "associatedRoles" : ["admin", "hardwareConfigurator"]
5 }
```

Nachfolgende Methoden werden von der taskTypes-API unterstützt:

- *PUT /taskTypes*
  Parameter: JSON-Objekt mit `typename` und `associatedRoles`
  Diese Methode erstellt einen neuen Task Type mit den mitgelieferten Attributen und weist ihm die Rollen aus `associatedRoles` zu. Nach Ausführung wird die generierte ID des neu erstellten Task Types zurückgegeben.
- *GET /taskTypes/{taskType}*
  Pfad-Parameter: {taskType}: Name des Task Types, dessen Werte abgefragt werden sollen
  Diese Methode gibt die Werte zum angefragten Task Type als JSON-Objekt zurück.

- *GET /taskTypes*
  Es werden alle vorhandenen Task Types in einem JSON-Array zurückgegeben.
- *POST /taskTypes/{taskType}*
  Pfad-Parameter: {taskType}: Name des Task Types, der geupdatet werden soll
  Parameter: JSON-Objekt mit `typename` und `associatedRoles`
  Diese Methode führt ein Update des Task Types mit dem angegebenen Namen durch und weist ihm die Rollen aus `associatedRoles` zu. Hierfür werden die im JSON-Objekt mitgegebenen Daten verwendet.
- *DELETE /taskTypes/{taskType}*
  Pfad-Parameter: {taskType}: Name des Task Types, der gelöscht werden soll
  Diese Methode löscht den Task Type mit dem angegebenen Namen.

Tasks

Die tasks-API stellt entsprechende CRUD-Methoden für Tasks bereit. Zusätzlich werden Funktionen definiert, die nur einzelne Bestandteile von Tasks ändern oder zurückgeben. Zu diesen zählt unter anderem das Abrufen der Presentation Details oder der Output-Parameter sowie die Möglichkeit, einen Task zu beanspruchen oder direkt den Status zu ändern. Auch hier werden die Parameter in einem JSON-Objekt, mit den entsprechenden Schlüsseln, übergeben:

- `id`: Bezeichner des Tasks, der bei der Erstellung vom Task Manager generiert wird
- `name`: Einzigartige Bezeichnung des Tasks
- `priority`: Priorität des Tasks
- `taskTypes`: Task Types des Tasks
- `status`: Status des Tasks, welcher bei der Generierung auf `created` gesetzt wird
- `claimedBy`: Nutzer, der den Task für sich beansprucht hat. Ist bei der Generierung des Tasks `null`
- `inputParameters`: JSON-Objekte, die die Parameter, die zur Bearbeitung des Tasks benötigt werden, enthalten
  - `label`: Bezeichner des Parameters
  - `value`: Wert des Parameters
- `outputParameters`: JSON-Objekte, die die Parameter, die nach der Bearbeitung des Tasks vom Nutzer zurückgesendet werden, enthalten
  - `label`: Bezeichnung, die dem Nutzer angezeigt wird
  - `value`: Wert, den der Nutzer in der Benutzeroberfläche eingetragen hat
- `presentationDetails`: JSON-Objekt, das die Informationen für die graphische Repräsentation in der Benutzeroberfläche enthält
  - `title`: Titel des Tasks
  - `subject`: Kurze Beschreibung des Tasks
  - `description`: Ausführliche Beschreibung des Tasks

Listing 3.6 Beispiel eines JSON-Objekts eines Tasks

```
{
  "id" : "89",
  "name" : "buyingHardware",
  "taskTypes" : ["hardware"],
  "status" : "ready",
  "claimedBy" : "Dean79",
  "priority" : "3",
  "inputParameters" : [
    { "label" : "raspberry",
      "value" : "Raspberry Pi 3"},
    { "label" : "shield",
      "value" : "Extension board LK-RB-Shield"},
    { "key" : "tempSensor",
      "label" : "LK temperature sensor"}
  ],
  "presentationDetails" : {
    "title" : "Buying Hardware",
    "subject" : "Buy the required hardware for Raspberry Pi with temperature sensor",
    "description" : "Please buy the required hardware at a suitable shop. The following
    items are needed:"
  }
}
```

Eine Definition eines Tasks in JSON könnte wie in Listing 3.6 aussehen.

Die tasks-API enthält folgende Methoden:

- *PUT /tasks*
  Parameter: JSON-Objekt mit `name`, `priority`, `taskType`, `presentationDetails`, `inputParameters` (optional) und `outputParameters` (optional)
  Diese Methode erstellt einen neuen Task mit den mitgelieferten Attributen. Nach Ausführung wird die generierte ID des neu erstellten Task zurückgegeben.
- *GET /tasks/*{*task*}
  Pfad-Parameter: {task}: ID des Tasks, dessen Werte abgefragt werden sollen
  Diese Methode gibt die Werte zum angefragten Task als JSON-Objekt zurück.
- *GET /tasks*
  Es werden alle vorhandenen Tasks in einem JSON-Array zurückgegeben.
- *GET /tasks/user/*{*user*}
  Pfad-Parameter: {user}: UserID des Nutzers, dessen Tasks aufgelistet werden sollen
  Diese Methode gibt alle Tasks, die von einem bestimmten Nutzer beansprucht wurden, in einem JSON-Array zurück.

- *GET /tasks/taskType/{taskType}*
  Pfad-Parameter: {taskType}: Typname des Task Types, dessen Tasks aufgelistet werden sollen
  Diese Methode gibt alle Tasks, die zu einem bestimmten Task Type gehören, in einem JSON-Array zurück.
- *GET /tasks/{task}/input*
  Pfad-Parameter: {task}: ID des Tasks, dessen Input-Parameter zurückgegeben werden sollen
  Diese Methode gibt die Input-Parameter eines bestimmten Tasks in einem JSON-Array zurück.
- *GET /tasks/{task}/output*
  Pfad-Parameter: {task}: ID des Tasks, dessen Output-Parameter zurückgegeben werden sollen
  Diese Methode gibt die Output-Parameter eines bestimmten Tasks in einem JSON-Array zurück.
- *GET /tasks/{task}/presentation*
  Pfad-Parameter: {task}: ID des Tasks, dessen Presentation Details zurückgegeben werden sollen
  Diese Methode gibt die Presentation Details eines bestimmten Tasks in einem JSON-Objekt zurück.
- *POST /tasks/{task}*
  Pfad-Parameter: {task}: ID des Tasks, der geupdatet werden soll
  Parameter: JSON-Objekt mit `name`, `priority`, `status`, `taskType`, `presentationDetails`, `inputParameter` (optional) und `outputParameter` (optional)
  Diese Methode führt ein Update des Tasks mit der angegebenen ID durch. Hierfür werden die im JSON-Objekt mitgegebenen Daten verwendet.
- *POST /tasks/{task}/output*
  Pfad-Parameter: {task}: ID des Tasks, dessen Output-Parameter geupdatet werden sollen
  Parameter: JSON-Array aus JSON-Objekten mit `label` und `value`
  Diese Methode führt ein Update des Tasks mit der angegebenen ID durch. Hierfür werden die im JSON-Objekt mitgegebenen Daten verwendet.
- *POST /tasks/{task}/claim*
  Pfad-Parameter: {task}: ID des Tasks, der beansprucht werden soll
  Parameter: JSON-Objekt mit `userId`
  Diese Methode beansprucht den Task für den Nutzer mit der im JSON angegebenen UserID.

- *POST /tasks/{task}/{status}*
  Pfad-Parameter: {task}: ID des Tasks, dessen Status geändert werden soll
  {status}: neuer Status des Tasks
  Diese Methode setzt einen neuen Status bei dem Task mit der angegebenen ID.
- *DELETE /tasks/{task}*
  Pfad-Parameter: {task}: ID des Tasks, der gelöscht werden soll
  Diese Methode löscht den Task mit der angegebenen ID.

3.6. Darstellung der Tasks in einer Smartphone-App

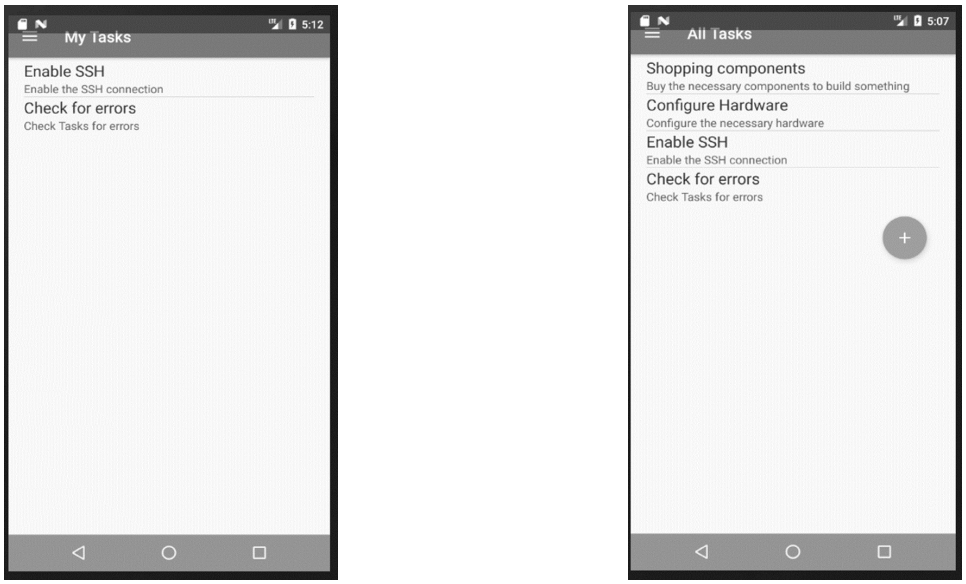
Um die Tasks schließlich dem Nutzer zur Bearbeitung zur Verfügung zu stellen, wird eine Smartphone-App implementiert. Die App unterstützt dafür Funktionen zur Auswahl und zur Beanspruchung von Tasks. Außerdem kann der Nutzer sich über die Smartphone-App registrieren und einige Verwaltungsoperationen durchführen. Darüber hinaus kann er Tasks erstellen und bearbeiten. Dadurch ist es den Nutzern der Smartphone-App möglich, in einer Anwendung alle für die Ausführung wichtigen Funktionen der REST-API sowie die Funktionen der Nutzerverwaltung optimal zu nutzen.

Eine Smartphone-App zur Darstellung der Tasks eignet sich besonders wegen der einfachen Bereitstellung und der Erreichbarkeit der Nutzer. So kann sie ohne großen Aufwand auf Smartphones oder Tablets installiert werden, die die meisten Nutzer ohnehin schon besitzen. Gerade diese Geräte sind in der heutigen Zeit unsere ständigen Begleiter, wodurch ein Nutzer jederzeit über neue Tasks benachrichtigt werden kann. Dadurch kann auf einen Task schneller reagiert und dieser zeitnah bearbeitet werden.

In der Smartphone-App werden die *Task List UI* sowie die *Task UI* umgesetzt. Um die Übersichtlichkeit in der Task List UI beizubehalten, wird in dieser nur der Titel und die Kurzbeschreibung der Tasks angezeigt. Die Smartphone-App verfügt über zwei Task List UIs. In einer werden alle für den Nutzer relevanten Tasks und in der anderen die vom Nutzer beanspruchten Tasks angezeigt. Beispiele für diese Ansichten sind in Abbildung 3.4 zu finden.

In der Task UI werden der Titel des Tasks sowie die Kurzbeschreibung und die ausführliche Beschreibung dargestellt. Zusätzlich werden die Input-Parameter unter die ausführliche Beschreibung gesetzt. Danach folgen die Output-Parameter. *Subject* und *Description* besitzen daher keine *Presentation Parameter*, sondern es werden stattdessen die Input-Parameter in einer Art Tabelle unter die Beschreibung gesetzt. Damit wird ein zusammenfassender Überblick über die wichtigsten Eingabeparameter für einen Task gegeben. Falls ein Task Eingaben vom Nutzer verlangt, werden Eingabemöglichkeiten für diese Output-Parameter bereitgestellt. Wenn die Bearbeitung eines Tasks abgeschlossen ist, wird dies dem Task Manager durch das Betätigen eines „Done“-Knopfes über die REST-API mitgeteilt und gegebenenfalls die Output-Parameter des Tasks upgedatet. Wichtig hierbei ist, dass der

Nutzer die Möglichkeit hat, Tasks anzuschauen, bevor er sie für sich beansprucht. Deshalb muss der Nutzer innerhalb der Task-Ansicht explizit einen Knopf drücken, damit der Task für ihn beansprucht wird (vgl. Abbildung 3.5b).



(a) Liste der vom Nutzer beanspurchten Tasks **(b)** Liste der für den Nutzer relevanten Tasks

Abbildung 3.4.: Listen-Ansichten der Smartphone-App

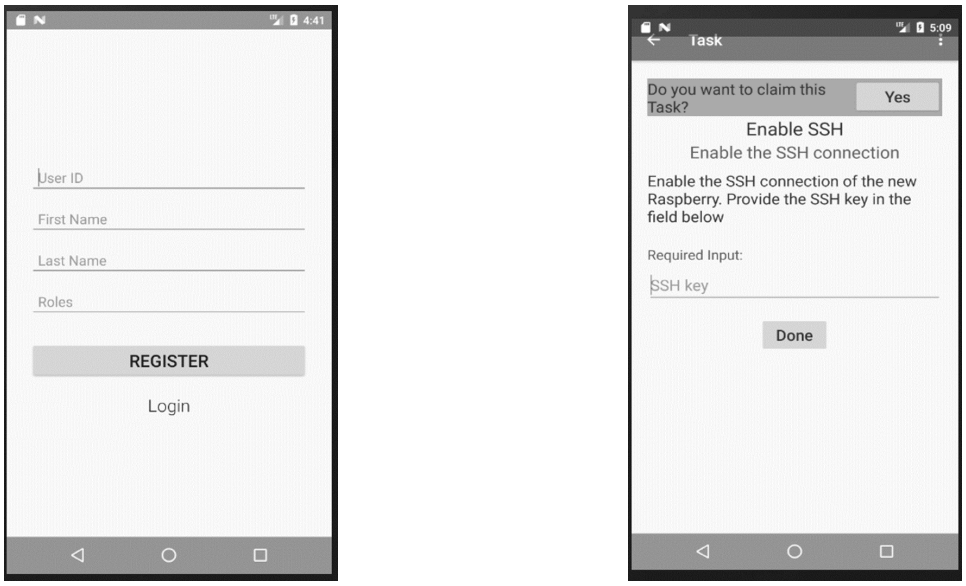
Alle Ansichten der Smartphone-App finden sich mit Beispielen in Anhang A. Im Folgenden wird auf die wichtigsten dieser Ansichten eingegangen:

- In den *Register*- und *Login*-Ansichten kann der Nutzer sich registrieren bzw. anmelden. Hierfür ist eine einzigartige UserID nötig (vgl. Abbildung 3.5a).
- *My Tasks* zeigt eine Liste, die die vom Nutzer beanspruchten Tasks enthält (vgl. Abbildung 3.4a). Auf diese Liste wird der Nutzer nach dem Login geleitet und hat einen Überblick über seine noch zu bearbeiteten Tasks. Ist ein Task abgeschlossen, wird er nicht mehr angezeigt.
- *All Tasks* beinhaltet eine Liste mit allen für den Nutzer relevanten Tasks (vgl. Abbildung 3.4b), d.h. mit allen noch nicht beanspruchten Tasks, deren Task Types seinen Rollen zugeordnet sind, sowie die von ihm beanspruchten Tasks. Wird ein Task von einem anderen Nutzer beansprucht, wird dieser Task nicht mehr angezeigt.
- *Task* zeigt einen ausgewählten Task an. Die Ansicht beinhaltet den Titel des Tasks, die Kurzbeschreibung und die ausführliche Beschreibung. Darunter werden die Input-Parameter als *additional Information* und die Output-Parameter als *required Input* angezeigt. Am Ende der Ansicht befindet sich ein „Done“-Knopf mit dem ein Task in den Zustand `completed` gesetzt wird. Falls der Task noch nicht beansprucht wurde, wird eine Meldung eingeblendet, die dem Nutzer die Möglichkeit gibt, dies zu tun (vgl. Abbildung 3.5b). Nach dem Beanspruchen erscheint an ihrer Stelle ein

Knopf, durch den mit der Bearbeitung des Tasks anfangen werden kann. Erst nach dem Betätigen dieses Knopfes ist es möglich den „Done“-Knopf zu drücken, da der Task in `inProgress` gesetzt wurde.

- In *Update Task* gelangt der Nutzer über das Menü innerhalb der Task-Ansicht. Hier kann ein Task noch nachträglich verändert werden, z.B. wenn ein zusätzlicher Output-Parameter hinzugefügt werden soll oder ein Fehler in der Beschreibung vorliegt.
- Über den „+“-Knopf in All Tasks besteht die Möglichkeit, einen Task hinzuzufügen. Der Nutzer wird dafür in *Add Task* navigiert. An dieser Stelle wird ein Formular mit allen notwendigen Attributen eines Tasks zum Ausfüllen bereitgestellt.
- In den *Settings* bzw. Einstellungen kann der Nutzer einige administrative Operationen durchführen. Er kann neue Rollen und Task Types anlegen, Rollen beitreten und seine Accountdaten bearbeiten.
- Wenn neue Tasks anfallen, die für den Nutzer relevant sind, erhält dieser eine *Benachrichtigung*. Diese erscheint in der Benachrichtigungsleiste des Smartphones und enthält die Anzahl der neuen, relevanten Tasks (vgl. Abbildung 3.6).

Die einzige Rendering-Methode, die vom System unterstützt wird, ist derzeit die Smartphone-App. Jedoch können diese bei Belieben erweitert werden.



(a) Register-Ansicht **(b)** Ansicht eines unbeanspruchten Tasks

Abbildung 3.5.: Ansichten der Smartphone-App

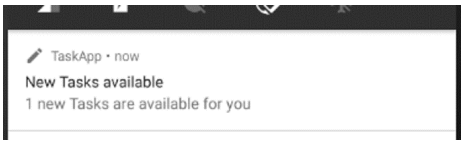


Abbildung 3.6.: Beispiel der Benachrichtigung der Smartphone-App

3.7. Abgrenzungen des Anwendungsfalls

Das vorgestellte Konzept basiert auf der WS-Human-Task-Spezifikation. Jedoch werden für den in Abschnitt 1.1 vorgestellten Ablauf nur grundlegende Teile dieser Spezifikation benötigt. Deshalb werden einige Abgrenzungen und Entscheidungen für den Anwendungsfall vorgenommen. Zum einen werden einige Aspekte nicht umgesetzt, da sie über den Anwendungsfall hinaus gehen. Diese umfassen beispielsweise, Stichtage für die Bearbeitung eines Tasks festzulegen, Anhänge einem Task anzufügen und den Task nach dem Beanspruchen an andere Nutzer weiterzureichen. Außerdem müssen Aspekte der Spezifikation an den Anwendungsfall angepasst werden. Hier sind unter anderem die Zuweisung der Nutzer und Benachrichtigungen betroffen. Auf diese zwei wird im Folgenden eingegangen. Weitere sind in den Abschnitten 3.3 und 3.4 enthalten und wurden in diesen Abschnitten vorgestellt.

3.7.1. Zuweisung der Nutzer

Nutzer werden zu Rollen zugewiesen, welche wiederum über Literale zu Generic Human Roles zugewiesen werden. Dies wird bereits von der vorherigen Implementierung des Task Managers unterstützt [Wag10]. Die Zuweisung über Literale eignet sich allerdings besonders, da die Daten für die Zuweisung von Rollen zu Generic Human Roles über die REST-API an den Task Manager übertragen werden. Das heißt, Rollen bekommen bei ihrer Erstellung mögliche Generic Human Roles zugeordnet. Da die REST-API Daten in JSON-Format übergeben werden, ist die Struktur, welche nur die Angabe des Rollennamens fordert, in diesem Fall weniger kompliziert als bei Logical People Groups (vgl. Abschnitt 2.2.5). Ein weiterer Vorteil ist die Unabhängigkeit von Änderungen im Nutzerverzeichnis, wodurch gegebenenfalls keine Neuzuweisungen benötigt werden.

Zwar besteht die Möglichkeit, Rollen zu Generic Human Roles hinzuzufügen, jedoch werden Generic Human Roles derzeit nicht weiter verwendet. Stattdessen werden *Task Types*, die ein zusätzliches Attribut eines Tasks sind, eingeführt. Den Task Types werden Rollen direkt zugeordnet. Deren Mitgliedern steht schließlich der Task für die Bearbeitung zur Verfügung. Da das fertige System zusätzlich zur Möglichkeit der Bearbeitung eines Tasks keine administrativen Operationen, wie z.B. das Setzen von Stichtagen oder die Weitergabe von Tasks, unterstützt, ist das Mapping von Task Types auf Rollen ausreichend. Beim Beanspruchen eines Tasks durch einen Nutzer wird seine UserID hinterlegt und er ist der einzige, der den jeweiligen Task updaten kann.

3.7.2. Benachrichtigungen

Benachrichtigungen werden für Tasks anders behandelt als in der Spezifikation angegeben. In der WS-Human-Task-Spezifikation werden unter anderem auch Benachrichtigungen für verpasste Stichtage, neue Anhänge oder Kommentare verschickt [OAS A]. Diese Aspekte werden in diesem Konzept nicht beachtet, da sie über den Anwendungsfall hinaus gehen. Aus diesem Grund sind die einzigen Benachrichtigungen, die ein Nutzer erhält, die über neue Tasks. Das heißt, wird ein neuer Task hinzugefügt, werden die Nutzer der relevanten Rollen darüber benachrichtigt. Diese Benachrichtigung geht allerdings nicht direkt vom Task Manager aus, sondern wird von der Smartphone-App durchgeführt. Das liegt vor allem an der Tatsache, dass in Android bereits Benachrichtigungen unterstützt werden und diese für eine solche Benachrichtigung geeignet sind [Goob].

4. Implementierung

In diesem Kapitel wird auf die prototypische Implementierung des Konzepts dieser Bachelorarbeit eingegangen. Dazu wurde die Implementierung des Task Managers aus [Wag10], um eine REST-API und dem Konzept der Task Types erweitert. Außerdem wurde der Task Manager soweit angepasst, dass er eigenständig auf einem Java-Webserver genutzt werden kann. Weiterhin wurde eine Smartphone-App, mit der der Nutzer die Tasks bearbeiten kann, für Android entwickelt.

4.1. Überblick über die verwendeten Technologien

Im Folgenden wird kurz auf die Implementierung der wichtigsten Komponenten eingegangen und die verwendeten Technologien genannt. Dabei liegt die in Abschnitt 3.1 vorgestellte Architektur (vgl. Abbildung 3.1) zugrunde.

Der Task Manager wird in Java 8.0[1] implementiert und schließlich auf einem Webserver bereitgestellt. Als Webserver wird eine Instanz des Apache Tomcat[2] 9.0.7 Webservers verwendet. Um den Task Manager auf Tomcat bereitzustellen, wird dieser in einem *Web Application Archive (WAR)* verpackt. Für die Implementierung wird IntelliJ IDEA[3] 2018.1.1 von JetBrains in der Ultimate-Version benutzt.

Für die Speicherung der Daten wird die Datenbank MySQL[4] verwendet. Um diese mit dem Task Manager ansprechen zu können, wird eine lokale Instanz des MySQL Community Servers[5] 8.0.11 bereitgestellt. Außerdem ist die Einbindung des MySQL Connector/J[6] 8.0 nötig, damit der Task Manager eine Verbindung zur Datenbank herstellen und diese mit *Structured Query Language (SQL)* abfragen kann.

[1] https://www.java.com/
[2] http://tomcat.apache.org/
[3] https://www.jetbrains.com/idea/
[4] https://www.mysql.com/
[5] https://dev.mysql.com/doc/refman/8.0/
[6] https://dev.mysql.com/doc/connector-j/8.0/

Die Smartphone-App wird in Android[7] implementiert und von Smartphones ab Android KitKat (4.4) unterstützt. Für die Ausführung der App wird ein lokaler Emulator[8] der Konfiguration Nexus 5X API 24 benutzt. Für die Entwicklung wird Android Studio[9] 3.1.1 verwendet.

4.2. Task Manager

Die nächsten Abschnitte beschäftigen sich mit der Implementierung des Task Managers. Hierfür wird die in [Wag10] vorgestellte Implementierung als Grundlage genutzt und erweitert. Dafür erhält der Task Manager eine REST-API, wird in ein WAR verpackt und in einem Webservice umgewandelt, um ihn auf Tomcat bereitstellen zu können. Weiterhin werden das Konzept der Task Types eingeführt sowie die Implementierung der Input- und Outputdaten angepasst. Im Zuge dessen sind Änderungen an der Datenbank notwendig. Auf diese Erweiterungen und Änderungen wird im Folgenden eingegangen.

4.2.1. Architektur

Der Aufbau des Task Managers ist in Abbildung 4.1 zu sehen. Dieser Aufbau kann aufgrund seiner Schichten als Layer-Architektur bezeichnet werden. Die unterstrichenen Bestandteile in Abbildung 4.1 werden im Rahmen dieser Arbeit hinzugefügt. Auf diese wird im Folgenden konkreter eingegangen und die wichtigsten der vorhandenen Bestandteile werden vorgestellt. Informationen zu den nicht vorgestellten Modulen sind in [Wag10] zu finden.

Der Task Manager wird als WAR auf einem Tomcat Webserver bereitgestellt. Dadurch stellt er eine eigenständige Komponente innerhalb des Gesamtsystems dar. Der Vorteil einer eigenständigen Komponente ist das leichte Verändern bzw. Updaten des Task Manager, ohne die restliche Implementierung des Systems zu beeinflussen. Dadurch entsteht eine minimale Abhängigkeit zwischen den einzelnen Komponenten im System. Außerdem kann der Task Manager von weiteren Task Parents benutzt werden.

Damit OpenTOSCA den Task Manager von außen ansprechen kann, wird ein Kommunikationskanal benötigt. Hierzu wird eine REST-API bereitgestellt. Diese besteht aus der *Task API*, der *Task Type API*, der *Role API* und der *User API*. Jede dieser APIs implementiert die entsprechenden Funktionen aus Abschnitt 3.5. Für die Implementierung werden für jede API Endpunkte und ein Service definiert (vgl. Abschnitt 4.2.4). Die Endpunkte nehmen die REST-Anfragen an und lesen die Daten aus den JSON-Objekten aus. Die Services greifen auf die entsprechenden Methoden der Interface-Schicht zu.

[7] https://developer.android.com/
[8] https://developer.android.com/studio/run/emulator
[9] https://developer.android.com/studio/

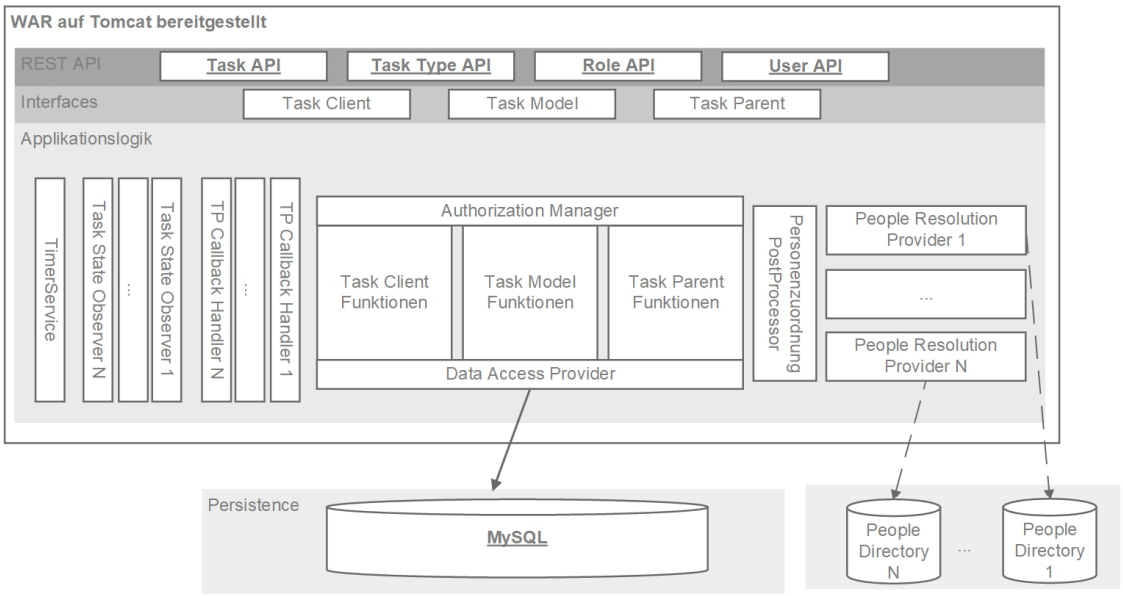


Abbildung 4.1.: Architektur des Task Manager nach [Wag10]

In der Interface-Schicht können drei verschiedene Interfaces implementiert werden. Diese sind das *Task Client Interface*, das *Task Model Interface* und das *Task Parent Interface*. Mit der Hilfe dieser Interfaces kann auf die Implementierung der Applikationslogik zugegriffen werden. Für den Prototyp wird eine auf das Konzept angepasste Implementierung der Interfaces angefertigt (vgl. Abschnitt 4.2.3).

Jedes der drei Interfaces greift auf die entsprechenden Funktionen der Applikationslogik zu. Konkret bedeutet das, das Task Client Interface auf *Task Client Funktionen*, das Task Model Interface auf *Task Model Funktionen* und das Task Parent Interface auf *Task Parent Funktionen*. Task Model Funktionen sind Methoden, die zur Entwurfszeit des Taskmodells benötigt werden. Die Task Parent Funktionen implementieren darauf aufbauend Methoden, um einen Task zu erstellen und diesen Personen zuzuweisen. Schließlich enthalten die Task Client Funktionen alle Aktionen, die den Lebenszyklus des Tasks und seine Attribute beeinflussen. Genauere Informationen zu den Funktionen der Applikationslogik folgen in Abschnitt 4.2.3 bzw. ist [Wag10] zu entnehmen.

Der Zugriff auf die MySQL-Datenbank wird über den *Database Access Provider* geregelt. Dafür stellt dieser Methoden zum Abfragen sowie zum Einfügen und Updaten der Datenhaltung bereit. Bei der Datenhaltung handelt es sich um die relationale Datenbank aus [Wag10], die für diese Arbeit um einige Tabellen erweitert und auf einem MySQL-Server aufgesetzt wird (vgl. Abschnitt 4.2.5).

Die restlichen Bestandteile der Implementierung aus [Wag10] gehen über den Anwendungsfall hinaus oder sind für diesen nicht relevant. Aus diesem Grund wird an dieser Stelle nicht weiter auf diese eingegangen.

4.2.2. Verwendete Frameworks

In diesem Abschnitt werden die Frameworks vorgestellt, die in der Implementierung des Task Managers benutzt werden. Als Erstes wird das *Spring Framework* [Piv] erläutert, das zur Implementierung des Database Access Providers verwendet wird. Außerdem ermöglicht es mit Hilfe von sogenannten *Beans*, Serviceinstanzen in anderen Klassen optimal zu nutzen. Für die Implementierung der REST-API wird *Jersey* [Oraa] benutzt. Jersey definiert durch Annotationen die Endpunkte in den Endpunkt-Klassen.

Spring Framework

Das *Spring Framework* (kurz: Spring) ist ein Open-Source Framework, das die Entwicklung von Applikationen in Java EE[10] unterstützt. Hierbei werden verschiedene Module, die je nach Bedarf eingebunden werden können, bereitgestellt. Diese dienen zur Vereinfachung des Programmierens und erreichen eine Standardisierung der Implementierung von Java EE-Applikationen. Dabei arbeitet Spring unter anderem mit *Plain Old Java Objects (POJO)* in Form von *Spring Beans* und dem Konzept von *Dependency Injection (DI)*, das Objekten benötigte Ressourcen und Services zuweist, ohne dass diese danach suchen müssen. Spring Beans sind Objekte, die vom *Spring Inversion of Control (IoC) Container* für die DI instanziiert und verwaltet werden. Sie werden in einem zentralen XML-Dokument definiert und ihre Klassen erhalten entsprechende Annotationen, beispielsweise `@Component` für eine Komponente oder `@Service` für Services (vgl. Listing 4.1 und Listing 4.4) [Piv].

Für die Implementierung des Prototyps wurden die Module *Spring Core*, *Spring Data JPA* und *Spring TX* verwendet. Mit Spring Core werden die Services der Endpunkte definiert. Dafür erhält jede Klasse, die einen Service implementiert, die Annotation `@Service`. Dies wird für die Nutzung der DI benötigt. Wenn der Service in einer Klasse genutzt werden soll, kann dies durch die Annotation `@Autowired` bei der Deklaration des Services innerhalb dieser Klasse erreicht werden (vgl. Listing 4.4). So können dieselben Service-Instanzen über die gesamte Applikation durch DI genutzt werden.

Für die Verbindung zur Datenbank wird Spring Data JPA verwendet. Dafür wird im zentralen XML-Dokument eine Spring Bean, die die Zugriffsinformationen der Datenbank beinhaltet, definiert. Das Abfragen der Datenbank und das Speichern von Änderungen geschieht über die Java Persistence API[11].

Spring TX wird für das Transaktionenmanagement genutzt. Dies hat den Vorteil, dass auf Datenbankebene die Implementierung des Data Access Providers mit `@Transactional` annotiert werden kann und die Verwaltung von Transaktionen kaum zusätzlichen Code verlangt.

[10] https://javaee.github.io/glassfish/documentation
[11] https://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

Jersey

Jersey ist eine Referenzimplementierung des *JAX-RS*[12] Frameworks. Beide dienen dazu *RESTful Web Services* zu bauen. Dies sind Webservices, die auf dem Konzept von REST und Hypertext Transfer Protocol (HTTP)[13] Methoden basieren. Das bedeutet, ein RESTful Web Service definiert Endpunkte, die über ein *Uniform Resource Identifier (URI)* ansprechbar sind. Diese Endpunkte umfassen POST-, GET-, PUT- und DELETE-Methoden.

Jersey hilft dabei, diese Methoden durch Annotationen zu definieren. Dafür enthält die Klasse, welche die Endpunkte implementiert, eine `@Path`-Annotation. `@Path` definiert die Basis-URI für die HTTP-Funktionen der Endpunkte. Außerdem wird jeder Endpunkt mit der entsprechenden HTTP-Funktion, z.B. `@PUT` für eine PUT-Funktion, und einem Pfad annotiert (vgl. Listing 4.2). Dieser Pfad ist eine Erweiterung der Basis-URI und sorgt dafür, dass ein Endpunkt in Verbindung mit seiner HTTP-Funktion einen einzigartigen Bezeichner besitzt. Über diesen wird der Endpunkt von Nutzern der REST-API angefragt. Über Jersey-Annotationen lassen sich außerdem Pfad-Parameter angeben. Soll beim Aufruf im Pfad zu einem bestimmten Endpunkt ein Pfad-Parameter mitgeliefert werden, z.B. die UserID eines Nutzers, kann dieser mit `@PathParam` in der Signatur der Methode übergeben werden (vgl. Listing 4.2). Alle anderen Parameter, die zur Ausführung der Methode nötig sind, werden im Body der HTTP-Anfrage mitgeliefert. Damit die REST-API angesprochen werden kann, muss in der `web.xml` des Webservices Jersey als Servlet-Dispatcher für diese Anfragen registriert werden [Oraa].

4.2.3. Änderungen an der bisherigen Implementierung

In diesem Abschnitt wird auf Änderungen der Implementierung des Task Managers aus [Wag10], welche im Folgenden Originalimplementierung genannt wird, eingegangen. Das Einbinden der REST-API wird in Abschnitt 4.2.4 und das Erweitern der Datenbank in 4.2.5 vorgestellt.

Der Task Manager ist in seiner Originalimplementierung keine eigenständige Komponente, sondern wird direkt in ein Workflow Management System integriert. Um das Konzept der Verwaltung von Human Tasks für OpenTOSCA zum Aufsetzen von IoT-Anwendungen umzusetzen, sollte der Task Manager eigenständig sein und durch eine REST-API angesprochen werden. Dafür wird die Originalimplementierung in einen RESTful Web Service umgewandelt, der auf einem Tomcat bereitgestellt wird. Tomcat verlangt Webservices in Form von WARs, weshalb die entsprechenden Anpassungen innerhalb der Gradle[14]-Konfiguration vorgenommen werden. Hierfür wird das Plugin `war` eingebunden, das den Task Manager als WAR baut. Für den Bau eines WARs werden Konfigurationsdateien

[12] https://github.com/jax-rs/spec/blob/master/spec.pdf
[13] https://www.w3.org/Protocols/
[14] https://gradle.org/

benötigt, mit denen ein Webservice definiert werden kann. Daher wird ein Web-Ordner angelegt, der eine `web.xml` enthält. Eine `web.xml` dient zur Beschreibung der Bereitstellung einer Webanwendung auf einem Webserver. Durch diese Änderungen wird erreicht, dass der Task Manager als eigenständige Komponente innerhalb des Systems genutzt werden kann. Damit dieser von OpenTOSCA und der Smartphone-App ansprechbar ist, müssen noch zusätzliche Endpunkt-Klassen und Service-Interfaces implementiert werden, die die REST-API definieren (vgl. Abschnitt 4.2.4).

Für das Konzept der Task Types werden einige Klassen zur Originalimplementierung hinzugefügt. Diese werden benötigt, um die Task Types behandeln zu können, in die Datenbank zu speichern und ihnen Tasks und Rollen zuzuordnen. Die neuen Entitäten werden nach demselben Prinzip aufgebaut wie die bereits vorhandenen. Analog zu den anderen Entitäten der Originalimplementierung wird eine serialisierbare Klasse `TaskType` erstellt. Durch Instanzen dieser Klasse können Task Types von Datenbankeinträgen zu Objekten und umgekehrt gemappt werden. Dafür wird die *Java Persistance API (JPA)* genutzt. JPA hilft dabei, Daten einer relationalen Datenbank in Java-Objekten zu speichern, abzufragen und Updates durchzuführen. Umgekehrt werden Instanzen dieser Klasse auf die relationale Datenbank gemappt und in der Datenbank gespeichert. Für die Arbeit außerhalb der Datenbank-Schicht wird `TaskTypeWrapper`, der ein Interface `ITaskType` implementiert, genutzt. Hier wird ein JPA-Objekt der `TaskType`-Klasse in eine `TaskTypeWrapper`-Instanz gekapselt. Auf dieser Instanz werden Methoden außerhalb der Datenbank-Schicht aufgerufen. Die `TaskTypeWrapper`-Klasse leitet den Aufruf an die entsprechende Methode von `TaskType` weiter. Somit bleibt die Applikationslogik unabhängig von JPA. Weiterhin wird in der Datenbank-Schicht ein Mapping von Task Types auf Rollen und Tasks benötigt. Da ein Task Type mehrere Rollen und Tasks zugeordnet werden und gleichzeitig Tasks und Rollen mehrere Task Types besitzen können, werden zwei zusätzliche Datenbanktabellen hinzugefügt. Diese sind *Task Type Roles* und *Task Type Tasks* (vgl. Abschnitt 4.2.5). Hierfür benötigt JPA ebenfalls Entitätsklassen. Deshalb werden die serialisierbaren Entitätsklassen `TaskTypeGroups` und `TaskTypeTasks` hinzugefügt. Diese bilden, wie `TaskType`, die Daten der relationalen Datenbank auf Java-Objekte ab und umgekehrt. Dabei entspricht eine Instanz dieser Klassen jeweils einem Task-Type-Rollen-Paar bzw. einem Task-Type-Task-Paar. In der Applikationslogik werden den Task Types über den `TaskTypeWrapper` die Rollen und Tasks in Form von Listen zugeordnet.

Da Input- und Output-Parameter nicht in der vom Konzept geforderten Form in der Originalimplementierung vorhanden sind und eine Änderung der bestehenden Behandlung einen zu großen Aufwand mit sich ziehen würde, werden die Datenbanktabellen *Input Parameter* und *Output Parameter* erstellt. Aus diesem Grund muss analog zu den Task Types für beide eine serialisierbare Entitätsklasse, `OutputParameter` und `InputParameter`, zur Nutzung von JPA angegeben werden. Außerdem wird für beide ein Interface `IOutputParameter` und `IInputParameter` erstellt, das von den Klassen `OutputParameterWrapper` und `InputParameterWrapper` implementiert wird. Ein Input- bzw. Output-Parameter gehört jeweils zu genau einem Task. Daher wird dieser Task mittels seiner ID (`tiid`) als Attribut innerhalb der Entitätsklassen angegeben.

Wegen genannter Änderungen wird eine neue Implementierung des `IDataAccessProvider`-Interface in `DataAccessRepositoryToscaImpl` angefertigt. Dieses entspricht dem Data Access Provider. Um das in Kapitel 3 vorgestellte Konzept entsprechend umsetzen zu können, wird dieses Interface um einige Methoden erweitert. Die Erweiterung des Interfaces ist nötig, da in den Klassen, die Zugang zu diesen Methoden benötigen, eine `IDataAccessProvider`-Instanz mit der Annotation `@Autowired` deklariert wird. Dies ist nötig, damit Spring im Hintergrund den richtigen Service laden und dieser somit von der Klasse genutzt werden kann. Die Implementierung des Interfaces wird in Form einer Spring Bean innerhalb der konfigurierenden XML-Datei deklariert. Es wird nicht direkt die implementierende Klasse innerhalb der nutzenden Klassen angeben. Dadurch muss nur im XML die Definition der Spring Bean ausgetauscht werden, falls eine andere Implementierung des Interfaces genutzt werden soll. In Listing 4.1 ist die Definition der Spring Bean für den Data Access Provider gegeben.

Listing 4.1 Beispiel einer Spring Bean Definition

```
1  <bean id="dataAccessTosca" class="com.htm.db.spring.DataAccessRepositoryToscaImpl"
       autowire="byName"/>
```

`Id` gibt den Bezeichner an, den der Service innerhalb einer nutzenden Klasse haben muss. Über diesen Bezeichner findet die DI von Spring den passenden Service. `Class` gibt an welche Implementierung des Services genutzt werden soll.

Im Nachfolgenden werden die Methoden des `IDataAccessProvider`-Interfaces, die durch `DataAccessRepositoryToscaImpl` implementiert werden, vorgestellt. Für Nutzer (`IUser`), Rollen (`IGroup`), Task Types (`ITaskType`) und Tasks (`ITaskInstance`) existieren jeweils Methoden zum Hinzufügen, Abfragen, Updaten und Löschen. Nachfolgend werden nur Methoden aufgelistet, die darüber hinaus gehen.

- *List<IWorkItem> getWorkItems(String tiid)*
  Diese Methode gibt alle Work Items in einer Liste zurück, die dem angefragten Task mit der ID `tiid` zugeordnet sind.
- *boolean deleteHumanTaskInstance(String tiid)*
  Diese Methode löscht den Task mit der ID `tiid` aus der Datenbank. Dabei werden auch alle dem Task zugeordneten Work Items, Input- und Output-Parameter gelöscht. Wurde der Task erfolgreich gelöscht, wird `true` zurückgegeben.
- *boolean addUserToGroup(String userId, String groupName)*
  Diese Methode fügt einen Nutzer `userId` zu einer Rolle `groupName` hinzu. Wurde der Nutzer erfolgreich zur Rolle hinzugefügt, wird `true` zurückgegeben.
- *Set<IUser> getUserByGroup(String groupName)*
  Diese Methode gibt alle Nutzer zurück, die der Rolle mit dem Rollennamen `groupName` zugeordnet sind.

- *Set<String> getUserAllGroups(String userId)*
  Diese Methode gibt alle Rollen eines Nutzers mit der UserID `userId` zurück.
- *Set<String> getGenericHumanRolesByGroup(String groupName)*
  Diese Methode gibt alle Generic Human Roles der Rolle mit dem Rollennamen `groupName` zurück.
- *boolean addGroupToTaskType(String taskTypeName, String groupName)*
  Diese Methode ordnet einem Task Type `taskTypeName` eine Rolle `groupName` zu.
- *Set<String> getTaskTypeAllGroups(String taskTypeName)*
  Diese Methode gibt alle Rollennamen von Rollen zurück, die dem Task Type `taskTypeName` zugeordnet sind.
- *Set<String> getTaskTypeByGroup(String groupName)*
  Diese Methode gibt alle Typnamen von Task Types zurück, die der Rolle `groupName` zugeordnet sind.
- *ITaskInstance createTask(String name,String title, String subject, String description, String priority, String[] taskTypeNames)*
  Diese Methode kreiert einen Task mit den angegebenen Werten und speichert diesen in die Datenbank. Gleichzeitig wird ein Work Item erstellt, das dem Task zugeordnet ist. In dieser Implementierung wird einem Task immer genau ein Work Item zugeordnet.
- *boolean createInputParameter(String key, String value, String tiid)*
  Diese Methode erstellt einen Input-Parameter mit den angegebenen Werten für den Task `tiid`. Wurde der Input-Parameter erfolgreich erstellt, wird `true` zurückgegeben.
- *boolean createOutputParameter(String key, String value, String tiid)*
  Diese Methode erstellt einen Output-Parameter mit den angegebenen Werten für den Task `tiid`. Wurde der Output-Parameter erfolgreich erstellt, wird `true` zurückgegeben.
- *boolean addTaskToTaskType(String tiid, String taskTypeName)*
  Diese Methode ordnet einem Task `tiid` einen Task Type `taskTypeName` zu. Wurde der Task Type erfolgreich zugeordnet, wird `true` zurückgegeben.
- *Set<IInputParameter> getInputParametersByTask(String tiid)*
  Diese Methode gibt alle Input-Parameter eines Tasks `tiid` zurück.
- *Set<IOutputParameter> getOutputParametersByTask(String tiid)*
  Diese Methode gibt alle Output-Parameter eines Tasks `tiid` zurück.
- *Set<ITaskInstance> getTasksByUser(String userId)*
  Diese Methode gibt alle Tasks, die von dem Nutzer mit der UserID `userId` beansprucht wurden, zurück.
- *Set<ITaskInstance> getTasksByTaskType(String taskTypeName)*
  Diese Methode gibt alle Tasks, die dem Task Type `taskTypeName` zugeordnet sind, zurück.

- *boolean updateOutputParameter(String tiid, HashMap outputParameters)*
  Diese Methode führt ein Update der Output-Parameter eines Tasks `tiid` durch. Dafür werden die Werte aus `outputParameters` genutzt. Wurden die Output-Parameter erfolgreich upgedatet, wird `true` zurückgegeben.
- *boolean updateState(String tiid, String state)*
  Diese Methode setzt den Status des Tasks `tiid` auf den in `state` enthaltenen Status. Dabei wird überprüft, ob die Statusänderung zulässig ist. Wurde der Status erfolgreich geändert, wird `true` zurückgegeben.
- *boolean claimTask(String tiid, String userId)*
  Diese Methode beansprucht den Task `tiid` für den Nutzer mit der UserID `userId`. Wurde der Task erfolgreich beansprucht, wird `true` zurückgegeben.
- *Set<String> getTaskTypeByTask(String tiid)*
  Diese Methode gibt alle Typnamen der Task Types zurück, die dem Task `tiid` zugeordnet sind.

Bei diesen Änderungen der Originalimplementierung ist anzumerken, dass diese nur auf den Anwendungsfall angepasst wurde. Das bedeutet, hier wird nur von einem Task Parent ausgegangen, weshalb diese in der Implementierung gänzlich vernachlässigt werden.

4.2.4. Einbinden der REST API

Wie schon in den vorherigen Abschnitten erwähnt, benötigt der Task Manager eine REST-API, um von OpenTOSCA und der Smartphone-App angesprochen zu werden. Dazu werden die in Abschnitt 3.5 vorgestellten Funktionen der *roles*-, der *users*-, der *taskTypes*- und der *tasks-API* realisiert. Für die Implementierung wird für jede API ein Service-Interface hinzugefügt. Diese Services besitzen Implementierungen, die auf die entsprechend benötigten Funktionen des Data Access Providers zugreifen. Die eigentlichen Endpunkte werden in der Endpunkt-Klasse der jeweiligen API definiert und rufen die Methoden ihres Services auf.

Die Endpunkt-Klassen umfassen `RolesEndpoint`, `UsersEndpoint`, `TaskTypesEndpoint` und `TasksEndpoint`. Diese Endpunkte stellen die Funktionen der APIs nach außen zur Verfügung. Dafür wird das Framework Jersey genutzt. Die Methoden der Endpunkt-Klassen erhalten die Annotationen `@GET`, `@PUT`, `@POST` oder `@DELETE`, je nachdem welche der API-Funktionen sie repräsentieren. Damit OpenTOSCA und die Smartphone-App die richtige Methode ansteuern können, wird in der Annotation `@Path` der Pfad zur jeweiligen Methode angegeben. Wird ein Parameter über den Pfad als Pfad-Parameter mitgegeben, wird dieser in der Methodensignatur mit `@PathParam` annotiert. Die Endpunkt-Klassen selbst erhalten eine `@Path`-Annotation, die den Basispfad der API enthält. Außerdem werden hier die mitgegebenen JSON-Objekte ausgewertet und der entsprechenden Methode des Services

übergeben. Die Antwort der Service-Methode wird als Response an OpenTOSCA bzw. die Smartphone-App weitergegeben. In Listing 4.2 ist `updateRole` dargestellt. Diese Methode implementiert den Endpunkt für *POST /roles/*{*role*} der roles-API.

Listing 4.2 Beispiel einer Endpunkt-Definition mittels Annotationen

```
1   @POST
2   @Path("/{role}")
3   public Response updateRole(@PathParam("role") String role, String jsonString) {
4       JSONParser parser = new JSONParser();
5       JSONObject json = null;
6       try {
7           json = (JSONObject) parser.parse(jsonString);
8           String[] genericHumanRoles = toStringArray((JSONArray)
                json.get("genericHumanRoles"));
9           boolean result = rolesService.updateRole(role, genericHumanRoles);
10          if (result) {
11              return Response.status(200).entity(result).build();
12          } else {
13              return Response.status(404).build();
14          }
15      } catch (ParseException e) {
16          e.printStackTrace();
17      }
18      return Response.status(500).build();
19  }
```

Da es sich um eine POST-Methode handelt, erhält `updateRole` eine `@POST`-Annotation. Weiterhin definiert `@Path` den Pfad innerhalb der API. Dieser enthält einen Platzhalter für den Namen der Rolle, die geändert werden soll. Der Wert dieses Platzhalters wird beim Aufruf der Methode als `role` an die Methode übergeben, weshalb die Annotation `@PathParam` verwendet wird. `UpdateRole` parst[15] das erhaltene JSON-Objekt und gibt die Werte seiner Schlüssel an die `updateRole`-Methode des `rolesService` weiter. Die in `reponse` enthaltene Antwort des `rolesService` wird schließlich zurückgegeben.

Um die erwähnten Services für die jeweilige APIs bereitzustellen, werden die Service-Interfaces `IRolesService`, `IUsersService`, `ITaskTypesService` und `ITasksService` definiert. Diese Interfaces deklarieren die Service-Methoden, die in den Endpunkt-Klassen aufgerufen werden. Jedes der Service-Interfaces erhält eine Implementierung, die durch die DI von Spring auf die Methoden des Data Access Providers zugreifen und dadurch die zur Ausführung notwendigen Methoden aufrufen kann. Hier werden die Implementierungen `RolesServiceImpl`, `UsersServiceImpl`, `TaskTypesServiceImpl` und `TasksServiceImpl` genutzt. Die Implementierung der `updateRole`-Methode der `RolesServiceImpl` ist zur Veranschaulichung in Listing 4.3 aufgezeigt.

[15] hier wird JSON.simple benutzt (https://code.google.com/archive/p/json-simple/wikis)

Listing 4.3 Beispiel der `updateRole`-Methode innerhalb der `RoleServiceImpl`

```
1  @Service
2  @Configurable
3  public class RolesServiceImpl implements IRolesService {
4
5      @Autowired
6      protected IDataAccessProvider dataAccessTosca;
7
8      @Override
9      public boolean updateRole(String id, String[] genericHumanRoles) {
10         boolean response = false;
11         try {
12             response = dataAccessTosca.updateGroup(id, genericHumanRoles);
13         } catch (DatabaseException e) { e.printStackTrace(); }
14         return response;
15     }
16     ....
17 }
```

Die Methode erhält über die zuvor ausgeführte Endpunkt-Methode die Parameter `id` und `genericHumanRoles`. Diese werden zum Aufruf der `updateGroup`-Methode des `dataAccessTosca`-Services, die das Update auf Datenbankebene durchführt, benötigt. Die Antwort dieser Methode wird zurückgegeben. Wie in Abschnitt 4.2.3 beschrieben, wird `dataAccessTosca` durch die Annotation `@Autowired` mit der Hilfe von Spring zur Verfügung gestellt.

Damit die API-Services in ihren Endpunkt-Klassen genutzt werden können, müssen die Implementierungen der Services mit `@Service` annotiert und eine Spring Bean, analog zu der in Listing 4.1 vorgestellten Bean, definiert werden. Dadurch wird der Service bei Spring registriert und kann mittels DI in den Endpunkt-Klassen verwendet werden. Wie dies in `RolesEndpoint` aussieht, wird in Listing 4.4 gezeigt.

`RoleEndpoint` kann durch die `@Autowired`-Annotation den bei Spring registrierten `IRolesService` nutzen. Dieser wird über den Bezeichner gefunden und nutzt die in der Spring Bean angegebene Implementierung. Weiterhin ist aufgezeigt, wie die Annotationen bei `RolesServiceImpl` aussehen.

Damit OpenTOSCA und die Smartphone-App die REST-API nutzen können, muss in der `web.xml` des Task Manager Jersey als Servlet-Dispatcher für diese Anfragen registriert werden. Daraufhin ist die REST-API von außen ansprechbar.

Listing 4.4 Beispiel einer Dependency Injection mit `@Autowired`

```
1  @Path("/roles")
2  @Controller
3  public class RolesEndpoint {
4
5      @Autowired
6      private IRolesService rolesService;
7      ....
8  }
9
10 @Service
11 @Configurable
12 public class RolesServiceImpl implements IRolesService {
13     ....
14 }
```

4.2.5. Datenbank

Der Task Manager besitzt eine relationale Datenbank. In einer relationalen Datenbank existieren Tabellen, in denen die Daten gespeichert werden. Eine solche Tabelle besteht aus Spalten und Zeilen, wobei eine Spalte ein Attribut repräsentiert und jede Zeile einen Datensatz innerhalb der Tabelle. Ein Datensatz wird mit einem einzigartigen Schlüssel identifiziert. Die Datenbank des Task Managers der Originalimplementierung wird größtenteils beibehalten. Allerdings müssen für die neue Implementierung einige Tabellen hinzugefügt werden. Der Aufbau der Datenbank mit allen Tabellen und Attributen sowie ihre Beziehungen untereinander ist in Abbildung 4.2 dargestellt. Hier ist anzumerken, dass die Tabellen für das User-Management (*User*, *Groups*) zwar bereits in der Originalimplementierung vorhanden sind, allerdings wurden diese nur zu Testzwecken verwendet. In diesem Prototyp werden die Tabellen jedoch zum Speichern der Nutzer und ihrer Rollen verwendet. Im Folgenden wird sowohl auf die neu hinzugefügten als auch auf die aus der Originalimplementierung genutzten Tabellen eingegangen.

Nicht alle der in der Originalimplementierung enthaltenen Datenbank-Tabellen werden für den Prototyp und seine Implementierung verwendet. Da unter anderem keine Task Parents berücksichtigt werden und Konzepte, wie z.B. Anhänge für Tasks, über den Anwendungsfall hinaus gehen, ist es nicht notwendig diese Tabellen zu verwenden. Deshalb wird im Nachfolgenden nur auf die vom Prototyp verwendeten Tabellen eingegangen.

- Die Tabelle *Human Task Instance* speichert Tasks und die zugehörigen Attribute ab. Es werden unter anderem die Presentation Details gespeichert, die die Informationen zur Darstellung eines Tasks enthalten. In dieser Tabelle werden nicht alle Attribute verwendet. Die ID des Tasks ist außerdem der Primärschlüssel der Tabelle.

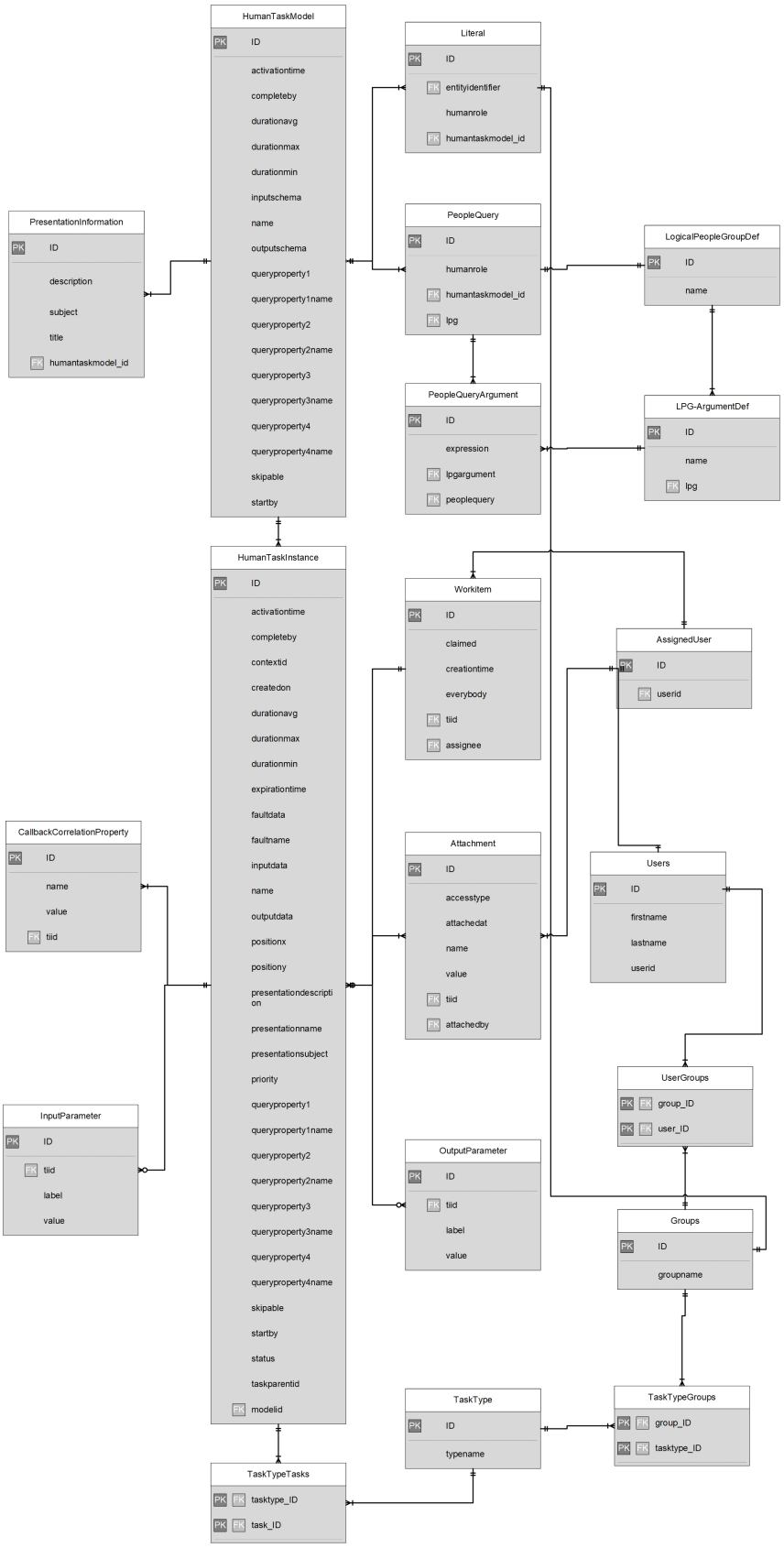


Abbildung 4.2.: Datenbankschema des Task Managers

- Die Work Items eines Tasks werden in *Work Item* gespeichert. Jeder Task besitzt nur ein Work Item. Dieses enthält unter anderem die UserID des zugewiesenen Nutzers und einen Indikator, der weitergibt, ob der Task beansprucht wurde. Der Primärschlüssel ist eine generierte ID. Die ID des Tasks sowie die ID des zugewiesenen Nutzers sind Fremdschlüssel.
- *Assigned User* mappt UserIDs auf generierte IDs. Diese IDs sind in Work Items als Fremdschlüssel vertreten und dienen dort zur Zuweisung der Nutzer.
- In *Literal* werden die Rollen und ihre Generic Human Roles gespeichert. Der Primärschlüssel ist eine generierte ID und der Name der Rolle ein Fremdschlüssel.
- Die Tabelle *Users* enthält alle Nutzer und ihre Informationen. Auch hier wird als Primärschlüssel eine generierte ID benutzt.
- In *Groups* werden alle Rollen und ihre Informationen mit einer generierten ID als Primärschlüssel gespeichert.
- *User Groups* wird dazu verwendet, Nutzer zu Rollen hinzufügen. Da ein Nutzer mehrere Rollen einnehmen und auch eine Rolle mehrere Nutzer haben kann, wird eine zusätzliche Tabelle benötigt, um diese n:m-Beziehung abzuspeichern. Die Attribute dieser Tabelle sind die generierte ID des Nutzers und die generierte ID der Rolle. Die Kombination aus beiden bildet den Primärschlüssel.

Um das Konzept der Human Tasks zum Aufsetzen von IoT-Anwendungen mit OpenTOSCA umsetzen zu können, werden einige Datenbank-Tabellen hinzugefügt. Diese werden im Folgenden vorgestellt:

- *Input Parameter:*
  `ID`: Automatisch generierte ID (Primärschlüssel)
  `TIID`: ID des Tasks, zu dem der Input-Parameter gehört
  `Label`: Bezeichner des Input-Parameters
  `Value`: Wert des Input-Parameters
  Mit dieser Tabelle werden die Input-Parameter abgespeichert.
- *Output Parameter:*
  `ID`: Automatisch generierte ID (Primärschlüssel)
  `TIID`: ID des Tasks, zu dem der Output-Parameter gehört
  `Label`: Bezeichner des Output-Parameters
  `Value`: Wert des Output-Parameters
  In dieser Tabelle werden die Output-Parameter gespeichert. Bei der Instanziierung sind die Werte der Output-Parameter leer.
- *Task Type:*
  `ID`: Automatisch generierte ID (Primärschlüssel)
  `Typename`: Einzigartiger Name des Task Types
  Diese Tabelle speichert die vorhandenen Task Types.

- *Task Type Tasks:*
  `TaskTypeId`: Automatisch generierte ID des Task Types (Fremdschlüssel)
  `TaskId`: ID des Tasks (Fremdschlüssel)
  Ein Task Type kann mehreren Tasks zugeordnet werden und ein Task mehreren Task Types. Deshalb muss diese n:m-Beziehung mit Hilfe einer zusätzlichen Tabelle aufgelöst werden. Ein Datensatz besteht aus einer Task-ID mit einer der zugehörigen Task-Type-IDs. Der Primärschlüssel ist eine Kombination aus beiden.
- *Task Type Groups:*
  `TaskTypeId`: Automatisch generierte ID des Task Types (Fremdschlüssel)
  `GroupId`: Automatisch generierte ID der Rolle (Fremdschlüssel)
  Da ein Task Type mehreren Rollen und eine Rolle mehreren Task Types zugeordnet werden kann, wird eine Tabelle benötigt, die diese n:m-Beziehung auflöst. In dieser Tabelle besteht ein Datensatz aus einer Rollen-ID mit einer der zugehörigen Task-Type-IDs. Der Primärschlüssel ist eine Kombination aus beiden.

Die Datenbank wird für die Implementierung des Prototyps auf einem lokalen MySQL-Server bereitgestellt. Dafür muss das Skript zum Erstellen der Tabellen angepasst werden, da der verwendete SQL-Dialekt nicht von MySQL angenommen wurde. Damit eine Verbindung zwischen der Datenbank und dem Task Manager besteht, muss zusätzlich der MySQL Connector/J eingebunden werden. Außerdem werden die Zugangsdaten der Datenbank in der entsprechenden Spring Bean geändert.

4.3. Task Manager Client App

Um die Human Tasks, die beim Aufsetzen einer IoT-Anwendung mit OpenTOSCA anfallen, dem Nutzer zur Bearbeitung darzustellen, wird eine Android-App realisiert. Diese dient im Prototyp als Task Client des Task Managers, weshalb sie im weiteren Task Manager Client App genannt wird. Die Task Manager Client App gibt dem Nutzer die Möglichkeit, alle für ihn verfügbaren Tasks zu überblicken, zu beanspruchen und zu bearbeiten. Außerdem kann der Nutzer sich mittels der App registrieren und weitere Rollen, Task Types und Tasks hinzufügen.

In den folgenden Abschnitten wird auf die Umsetzung dieser Funktionen in der Implementierung der Task Manager Client App eingegangen. Zunächst wird ein Überblick über die Architektur gegeben. Daraufhin werden Details zur Implementierung der Task Manager Client App vorgestellt. Zuletzt wird die Datenbank erläutert.

4.3.1. Architektur Task Manager Client App

In Abbildung 4.3 ist die Architektur der Task Manager Client App gegeben. Der Aufbau ist einem *Model View Controller (MVC)* nachempfunden. Jede Komponente und ihre Bestandteile werden im Folgenden vorgestellt.

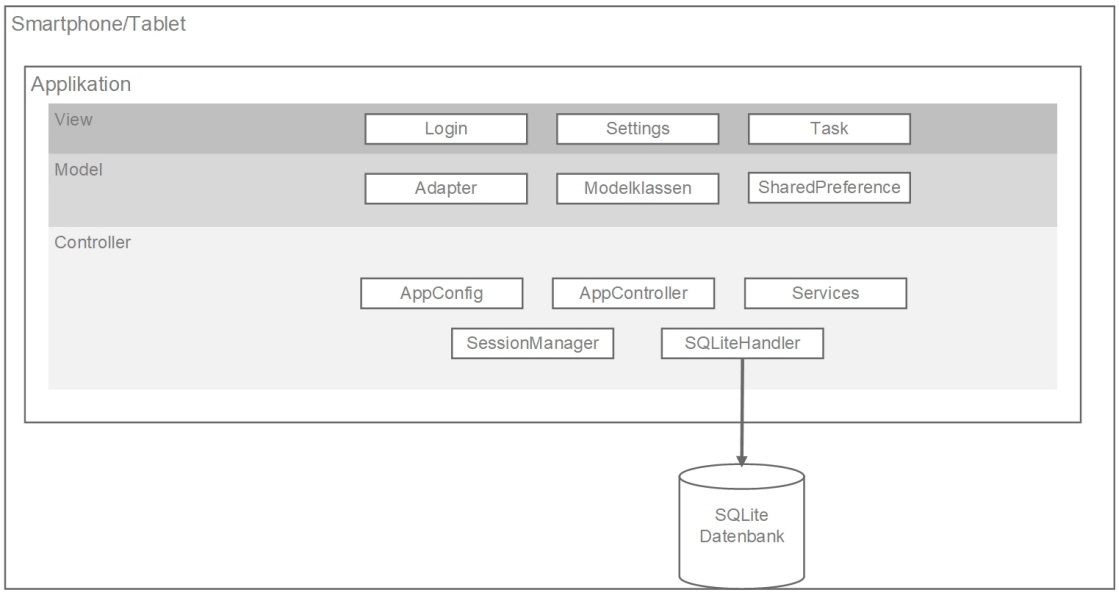


Abbildung 4.3.: Architektur der Task Manager Client App

Die Applikation wird auf einem Smartphone oder Tablet bereitgestellt. Auf demselben Gerät befindet sich außerdem die Datenbank der Task Manager Client App. Diese Datenbank enthält alle Daten, die zur Darstellung und Bearbeitung der Tasks zwischengespeichert und benötigt werden. Für die Implementierung der Datenbank wird SQLite [Good] genutzt.

Die Task Manager Client App wird für Android [Gooa] implementiert. Android ist ein Open-Source Betriebssystem und Software-Plattform für mobile Geräte. Es hat den Vorteil, dass die Programmierung einfach an die Bedürfnisse angepasst werden kann. Außerdem ist Android eines der am weitverbreitetsten Betriebssysteme für mobile Geräte. Dadurch wird die App von einer Vielzahl an Geräten unterstützt.

View enthält die *Activities* der App, mit denen der Nutzer interagiert. Eine Activity definiert eine Ansicht innerhalb der App. Diese besteht aus XML-Dokumenten, die die Beschreibung des Layouts enthalten und einer Java-Klasse, die den Quellcode zum Steuern der Activity beinhaltet. Für die Implementierung des Konzepts aus Kapitel 3 werden drei Ansichtgruppen *Login*, *Settings* und *Task* benötigt. Login enthält die Activities, die zum Einloggen und Registrieren in der App gebraucht werden. Die Activities, in denen Einstellungen vorgenommen werden können, befinden sich in Settings. Zur eigentlichen Taskbearbeitung werden die Activities aus Task benutzt. Hier werden die Listenansichten der für den Nutzer relevanten Tasks und der vom Nutzer beanspruchten Tasks sowie die Ansicht des Tasks zur Bearbeitung definiert.

Innerhalb der Activities werden in Folge von Nutzereingaben die REST-Funktionen der Task Manager API angefragt. Diese Anfragen werden in einer Queue gespeichert, die vom *AppController* des Controllers verwaltet wird. An dieser Stelle wird mit Hilfe der *Volley* [Gooc] Bibliothek die Anfragen an den Task Manager über die REST-API weitergeleitet und das Ergebnis an die anfragende Klasse zurückgegeben (vgl. Abschnitt 4.3.2). Die

Konfigurationen, um den Task Manager ansprechen zu können, werden in *AppConfig* hinterlegt. Services, die im Hintergrund der Applikation laufen sind in *Service* zu finden. Der *SessionManager* verwaltet die aktuelle Session des Nutzers und hinterlegt den aktuell angemeldeten Nutzer. Die Verbindung zur SQLite-Datenbank wird über den *SQLiteHandler* sichergestellt. Dieser besitzt außerdem Methoden zum Manipulieren dieser Datenbank.

Model stellt die Entitätsklassen für *Task*, *User*, *Role* und *TaskType* sowie *PresentationDetails*, *Inputparameter* und *Outputparameter* zur Verfügung. Diese Entitäten werden innerhalb der Applikation verwendet, um mit den Daten aus der Datenbank bzw. den vom Task Manager gesendeten Daten zu arbeiten. Außerdem hinterlegt *SharedPreference* die UserID des aktuellen Nutzers, falls diese von der Applikation benötigt wird. Zuletzt sorgt *Adapter* dafür, dass einzelne Attribute der Entitäten innerhalb der Ansichten angezeigt werden können. Beispiele dafür sind der Titel und die Kurzbeschreibung des Task in den Listen-Ansichten.

4.3.2. Details zur Implementierung

Die Task Manager Client App implementiert die in Abschnitt 3.6 vorgestellten Ansichten und die dort erwähnten Funktionen. In Anhang A ist für jede der Ansichten ein Beispiel zu finden. Diese Abbildungen sind Screenshots der hier vorgestellten Implementierung der Task Manager Client App. Im nachfolgenden Abschnitt werden einige Details zur Implementierung dieser erläutert.

Die Applikation nutzt die REST-API des Task Manager, um Tasks, Nutzer, Rollen und ihre Daten abrufen zu können. Dafür wird die Bibliothek Volley [Gooc] benutzt. Diese unterstützt automatisches *Scheduling* von Anfragen an eine API. Mit Hilfe von Volley wird im `AppController` eine `RequestQueue`[16] angelegt. Dabei muss berücksichtigt werden, dass `AppController` als Singleton-Klasse implementiert wird, damit global auf die `RequestQueue` zugegriffen werden kann. Die `RequestQueue` verwaltet die Anfragen an die Task Manager API. Die Anfragen können global in der Applikation mit *Volley Requests* gestellt werden. Dabei werden zwei Arten von Volley Requests zur Verfügung gestellt, `StringRequest` und `JsonObjectRequest`. In der Implementierung der Task Manager Client App werden Anfragen an die API des Task Managers meist nach Eingaben des Nutzers getätigt, z.B. wenn dieser den Knopf zum Senden der getätigten Eingaben drückt oder beim Aktualisieren der *AllTask-* und *MyTask-Ansicht*. In Listing 4.5 ist ein Beispiel für einen Volley Request gegeben. Dieser ist ein Auszug aus der Activity für das Login des Nutzers.

Der Volley Request fordert den Nutzer mit der UserID an, die ein Nutzer zuvor in die Login-Ansicht eingetragen hat. Hierfür wird ein `JsonObjectRequest getRequest` definiert. `GetRequest` wird im Konstruktor mitgeteilt, dass es sich bei der angefragten Methode um eine GET-Methode handelt. Außerdem wird der Pfad zur API-Funktion angegeben. Dieser ist in `AppConfig` hinterlegt und bekommt die eingegebene UserID angehängt. Da dem Request kein JSON-Objekt mitgegeben wird, wird für dieses `null` im Konstruktor

[16] Für eine Beispielimplementierung siehe https://developer.android.com/training/volley/requestqueue

angegeben. Danach werden zwei Methoden überschrieben, die angeben, wie die Antwort des Task Managers zu behandeln ist und wie bei einer fehlgeschlagenen Anfrage reagiert werden soll. In der Implementierung der Login-Activity wird nach einer erfolgreichen Anfrage eine Session gestartet. Die Nutzerdaten in die SQLite-Datenbank gespeichert und der Nutzer auf seine MyTask-Ansicht geleitet.

Listing 4.5 Beispiel eines Volley Requests

```
1          // Anfrage an die Task Manager API
2          JsonObjectRequest getRequest = new JsonObjectRequest(Request.Method.GET,
               AppConfig.USER_API + userId,
3                  null, new Response.Listener<JSONObject>() {
4              @Override
5              public void onResponse(JSONObject response) {
6                  ....
7              }
8          }, new Response.ErrorListener() {
9              @Override
10             public void onErrorResponse(VolleyError error) {
11                ....
12             }
13         });
14         // Hinzufuegen der Anfrage zur RequestQueue
15         AppController.getInstance().addToRequestQueue(getRequest);
```

Um die Anfragen an die REST-API zu verringern, wird eine eigene Datenbank für die Task Manager Client App bereitgestellt. Dadurch müssen weniger Anfragen an die Task Manager API gestellt werden. So muss unter anderem nicht jedes Mal, wenn die neu hinzugefügten Tasks abgerufen werden, zuvor eine Anfrage nach den Rollen des Nutzers getätigt werden. Hierfür kann eine eingebettete Datenbank mit der Open-Source Datenbank SQLite [Good] realisiert werden. Der Vorteil von SQLite ist, dass Android von Haus aus eine Implementierung dieser Datenbank mit sich bringt. SQLite ist eine relationale Datenbank, die mit SQL angesprochen werden kann. Außerdem stellt es Methoden für eine vereinfachte Interaktion bereit, d.h. es ist nicht nötig SQL-Anfragen zu schreiben, sondern die Daten können einer entsprechenden Methode übergeben werden. Damit alle Methoden, die direkt auf Funktionen von SQLite zugreifen, an einem Ort hinterlegt sind, werden sie im `SQLiteHandler` implementiert. Die Methoden des `SQLiteHandler` werden vor allem bei Erhalt einer Antwort der Task Manager API genutzt, um die erhaltenen Daten für die weitere Verarbeitung zu hinterlegen. Auch für die Darstellung der Tasks in der *Task-Ansicht* werden die Daten aus der SQLite-Datenbank genutzt. Listing 4.6 zeigt die Methode `addPresentationDetails`.

Hier werden die übergebenen Variablen in die Datenbanktabelle *Presentation Details*, welche als `TABLE_PRESENTATION` bezeichnet ist, gespeichert. Zuerst wird die Verbindung zur SQLite-Datenbank mit `getWritableDatabase` hergestellt. Danach werden die Variablen zusammen mit den entsprechenden Bezeichnungen der Attribute der Datenbanktabelle zu einem `ContentValues`-Objekt hinzugefügt. Statt einer SQL-Anfrage wird die Methode `insert`

auf der SQLite-Datenbank aufgerufen und der Tabellennamen sowie das `ContentValues`-Objekt übergeben. Die Methode gibt die generierte ID des Datensatzes zurück. Am Ende wird die Datenbankverbindung geschlossen.

Listing 4.6 Beispiel der `addPresentationDetails`-Methode innerhalb des `SQLiteHandler`

```
1     public long addPresentationDetails(String title, String subject, String description,
          int tiid) {
2         SQLiteDatabase db = this.getWritableDatabase();
3
4         ContentValues values = new ContentValues();
5         values.put(PRESENTATION_TITLE, title);
6         values.put(PRESENTATION_SUBJECT, subject);
7         values.put(PRESENTATION_DESCRIPTION, description);
8         values.put(PRESENTATION_TIID, tiid);
9
10        long i = db.insert(TABLE_PRESENTATION, null, values);
11        db.close();
12
13        return i;
14    }
```

Im Konzept aus Kapitel 3 wird eine Benachrichtigung des Nutzers über neu hinzugefügte Tasks gefordert. Da Android das Konzept von Benachrichtigungen unterstützt, wird diese in der Task Manager Client App implementiert. Dazu sollte die Applikation regelmäßig auf neue Tasks überprüfen. Um dabei nicht die reguläre Nutzung der Task Manager Client App zu stören bzw. zu behindern, wird diese regelmäßige Anfrage in einem Hintergrundservice `GetTasksService` definiert. `GetTasksService` wird, falls noch nicht gestartet, beim ersten Navigieren auf die MyTask-Ansicht durch den `AlarmManager` ausgelöst und in einem beliebig wählbaren Intervall wiederholt. Wenn der `GetTasksService` ausgelöst wird, sendet er einen Volley Request an die Funktion *GET /tasks* der Task Manager API. Diese Funktion gibt alle Tasks, die beim Task Manager gespeichert sind, zurück. Daraufhin wird für jeden Task überprüft, ob er für den Nutzer relevant ist. Sollte dies der Fall sein, wird er in der SQLite-Datenbank gespeichert, sofern er nicht bereits in dieser vorhanden ist oder von einem anderen Nutzer beansprucht wurde. Der `GetTaskService` zählt die Tasks, die in die SQLite-Datenbank eingefügt werden, mit. Wird mindestens ein für den Nutzer relevanter Task hinzugefügt, löst der Service eine *Android-Notification* [Goob] aus, die den Nutzer über die Anzahl der neuen Tasks informiert.

Möchte ein Nutzer einen Task bearbeiten, kann er diesen entweder in der AllTask-Ansicht oder, falls er ihn bereits beansprucht hat, in der MyTask-Ansicht auswählen. Der Task wird durch die Auswahl an die Task-Ansicht weitergegeben. Diese liest zuerst die Presentation Details und, falls vorhanden, die Input- und Output-Parameter aus der SQLite-Datenbank aus, um diese anzuzeigen. Ist der Task noch nicht beansprucht, befindet er sich im Status `created`. In diesem Status wird ein Feld mit einem Knopf zum Beanspruchen angezeigt. Wird der Knopf betätigt, schickt die Task-Ansicht einen Volley Request an die Task Manager API zum Beanspruchen des Tasks durch den Nutzer. Hierfür wird die UserID des

Nutzers als JSON-Objekt mitgegeben (vgl. Abschnitt 3.5). In beiden Datenbanken wird der Nutzer als der zugewiesene Nutzer abgespeichert und der Status ändert sich in `ready`. Das hat zur Folge, dass ein Knopf zum Starten des Tasks angezeigt wird. Außerdem ist der Task nun in der MyTask-Ansicht der Applikation zu finden. Nach dem Starten des Tasks durch den Nutzer, wird er in den Status `inProgress` gesetzt. Diese Statusänderung teilt die Task-Ansicht mittels Volley-Request dem Task Manager mit. Der „Done“-Knopf am unteren Ende der Ansicht, kann erst gedrückt werden, wenn der Task gestartet wurde. Hat der Nutzer den Task durchgeführt und, wenn nötig die Output-Parameter ausgefüllt, werden mit dem Drücken des „Done“-Knopfs zwei Volley Requests an die Task Manager API gesendet. Zum einen werden die ausgefüllten Output-Parameter weitergeleitet und zum anderen wird der Task als `completed` markiert. Sobald der Task als `completed` markiert ist, wird er aus der SQLite-Datenbank gelöscht.

4.3.3. Datenbank

Wie schon im vorherigen Abschnitt erwähnt, erhält die Task Manager Client App ihre eigene Datenbank. Diese dient zum Zwischenspeichern der vom Task Manager angeforderten Daten. Dabei handelt es sich, um eine relationale Datenbank, die beim Installieren der Applikation auf dem Android-Gerät bereitgestellt wird und beim Einloggen eines Nutzers mit dessen Daten initialisiert wird. Beim Ausloggen werden alle Daten der Datenbank automatisch gelöscht, da immer nur die für den aktuellen Nutzer relevanten Daten gespeichert werden.

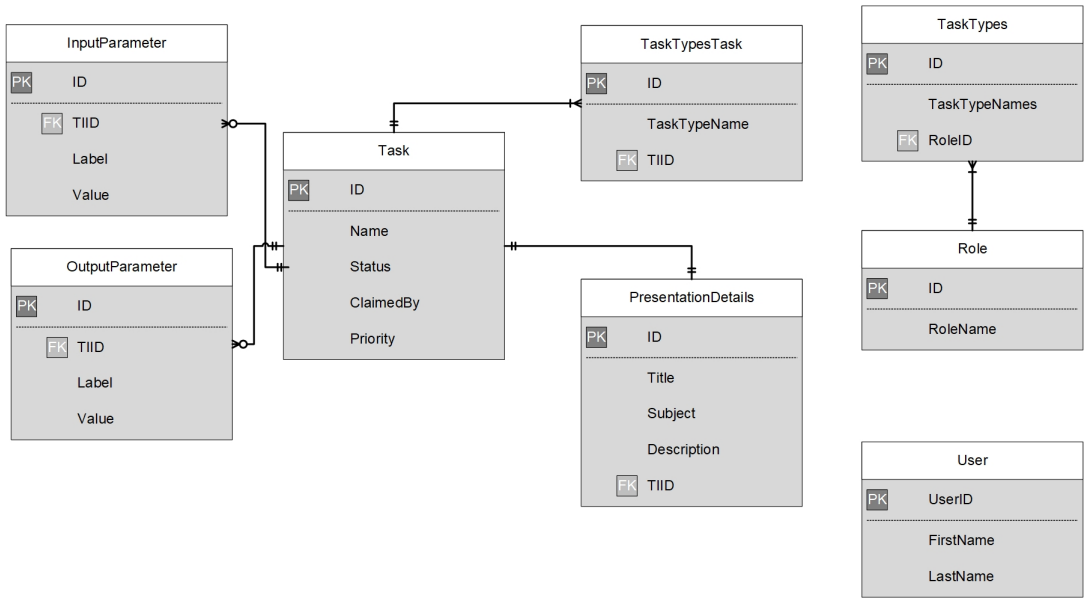


Abbildung 4.4.: Datenbankschema der Task Manager Client App

Die Datenbank wird durch die Open-Source Datenbank SQLite realisiert (vgl. Abschnitt 4.3.2). Sie besitzt Tabellen zur Nutzer- und Taskverwaltung. Es werden nur Tasks gespeichert, die der Nutzer beansprucht hat oder beanspruchen kann, also die Tasks, deren Task Types den Rollen des Nutzers zugeordnet sind. In Abbildung 4.4 sind alle Datenbanktabellen und ihre Beziehungen als Datenbankschema dargestellt.

Es ist zu erkennen, dass einige Bestandteile eines Tasks in eigene Tabellen ausgelagert werden. Somit können die Bestandteile eines Tasks, die zu diesem Zeitpunkt gebraucht werden, bearbeitet werden. Dazu zählen z.B. die Presentation Details zur Anzeige des Task in den Listen und der Task-Ansicht oder die Output-Parameter, wenn diese an den Task Manager geleitet werden sollen.

Im Nachfolgenden werden alle Tabellen und ihre Attribute kurz vorgestellt:

- *Task:*
  `ID`: vom Task Manager mitgegebene ID des Tasks (Primärschlüssel)
  `Name`: Bezeichnung des Tasks
  `Status`: Aktueller Status des Tasks
  `ClaimedBy`: UserID des Nutzers, der den Task beansprucht hat
  `Priority`: Priorität des Tasks
  In dieser Tabelle werden die Tasks, die für den Nutzer relevant sind, gespeichert. Einige der Attribute werden allerdings in andere Tabellen ausgelagert, unter anderem Input-Parameter und Presentation Details. Falls der Task noch nicht beansprucht ist, ist der Wert von `ClaimedBy` `null`.

- *Input Parameter:*
  `ID`: Automatisch generierte ID (Primärschlüssel)
  `TIID`: ID des zugehörigen Tasks
  `Label`: Bezeichner des Input-Parameters
  `Value`: Wert des Input-Parameters
  In dieser Tabelle werden die Input-Parameter gespeichert. Die Task-ID `TIID` ist ein Fremdschlüssel und definiert, zu welchem Task ein Input-Parameter gehört.

- *Output Parameter:*
  `ID`: Automatisch generierte ID (Primärschlüssel)
  `TIID`: ID des zugehörigen Tasks
  `Label`: Bezeichner des Output-Parameters
  `Value`: Wert des Output-Parameters
  In dieser Tabelle werden die Output-Parameter gespeichert. Die Task-ID `TIID` ist ein Fremdschlüssel und definiert, zu welchem Task ein Output-Parameter gehört.

- *Presentation Details:*
  `ID`: Automatisch generierte ID (Primärschlüssel)
  `Title`: Titel des Tasks
  `Subject`: Kurze Beschreibung des Tasks
  `Description`: Ausführliche Beschreibung des Tasks
  `TIID`: ID des zugehörigen Tasks

In dieser Tabelle werden die Presentation Details gespeichert. Sie werden genutzt, um dem Nutzer den Task anzuzeigen. Die Task-ID `TIID` ist ein Fremdschlüssel und definiert, zu welchem Task die Presentation Details gehören.

- *Task Types Task:*
  `ID`: Automatisch generierte ID (Primärschlüssel)
  `TaskTypeName`: Bezeichner des Task Types
  `TIID`: ID des zugehörigen Tasks
  In dieser Tabelle wird die n:m-Beziehung zwischen Task und Task Type aufgelöst. Dies ist nötig, da ein Task mehrere Task Types besitzen kann und umgekehrt. Ein Datensatz besteht aus einer ID, einem Task Type und einer der dazugehörigen Tasks. `TIID` ist ein Fremdschlüssel.
- *Task Types:*
  `ID`: Automatisch generierte ID (Primärschlüssel)
  `TaskTypeName`: Bezeichner des Task Types
  `RoleID`: ID der zugehörigen Rolle
  In dieser Tabelle wird die n:m-Beziehung zwischen Rolle und Task Type aufgelöst. Dies ist nötig, da eine Rolle mehrere Task Types besitzen kann und umgekehrt. Ein Datensatz besteht aus einer ID, einem Task Type und einer der dazugehörigen Rollen. `RoleID` ist ein Fremdschlüssel.
- *Role:*
  `ID`: Automatisch generierte ID (Primärschlüssel)
  `RoleName`: Bezeichner der Rolle
  In dieser Tabelle werden die Rollen des Nutzers gespeichert.
- *User:*
  `UserID`: Einzigartige UserID des Nutzers (Primärschlüssel)
  `FirstName`: Vorname des Nutzers
  `LastName`: Nachname des Nutzers
  In dieser Tabelle wird der Nutzer gespeichert. Es existiert nur ein Eintrag. Die Tabelle wird genutzt, um die Nutzerinformationen des aktuellen Nutzers der Applikation zu hinterlegen. `UserID` entspricht hierbei der beim Task Manager hinterlegten UserID.

5. Verwandte Arbeiten

In diesem Kapitel werden JBoss jBPM [Jbpb] und WSO2 Business Process Server [Wsoa] vorgestellt, die sich ebenfalls mit Human Tasks beschäftigen. Beide Anwendungen implementieren einen Service, der auf WS Human Task basiert. Allerdings sind beide Services ein Teil eines gesamten Workflow Management Systems.

Ein Workflow Management System dient zur Ausführung von modellierten Arbeitsabläufen. Die dafür verwendete Komponente heißt *Workflow Engine*. Sie interpretiert zur Laufzeit Ereignisse und reagiert mit der zuvor definierten Operation. Um einen Arbeitsablauf zu modellieren, werden Prozesssprachen verwendet. Beispiele hierfür sind BPMN 2.0[1] und BPEL[2] [SS06].

JBPM ist ein Workflow Management System, mit dem sich unter anderem Prozessmanagement und Arbeitsabläufe umsetzen lassen. Für die Ausführung muss eine Definition in Prozesssprache vorliegen. JBPM unterstützt Prozesssprachen wie BPMN 2.0 und BPEL. Das System kann entweder als eigenständiger Service genutzt oder in einen vorhandenen Service integriert werden. Für Human Tasks wird ein Service, der auf der Spezifikation WS Human Task basiert, implementiert. Dafür stellt jBPM einen speziellen Human-Task-Knoten innerhalb der Modellierung des Prozesses zur Verfügung. Dieser kann an der entsprechenden Stelle des durchzuführenden Prozesses platziert und konfiguriert werden. Der Human-Task-Service kümmert sich schließlich um den Lebenszyklus des Tasks und speichert alle nötigen Zustände, Task-Listen und Informationen. Außerdem stellt der Human-Task-Service sowohl eine Java-API mit den für den Lebenszyklus nötigen Methoden als auch eine Benutzeroberfläche für die Bearbeitung von Tasks durch den Nutzer bereit [Jbpa; Jbpb].

WSO2 Business Process Server (BPS) ist eine Komponente des WSO2 Enterprise Integrator. Mit diesem Workflow Management System lassen sich Arbeitsabläufe und Human Tasks verwalten, bereitstellen, anzeigen und ausführen. Dies geschieht alles in einer Server-Instanz. Wie jBPM unterstützt auch WSO2 BPS die Prozesssprachen BPM 2.0 und BPEL. Human Tasks werden in einer Benutzeroberfläche angezeigt, die ebenfalls auf der Server-Instanz liegt. Durch diese können die Tasks beansprucht und ausgeführt werden. Zusätzlich zur einem Human-Task-Knoten innerhalb der Modellierung muss in der Benutzeroberfläche ein Archiv mit der Definition des Tasks in XML hochgeladen werden. Die XML-Repräsentation muss der in WS Human Task vorgestellten Repräsentation

[1] siehe https://www.omg.org/spec/BPMN/2.0
[2] siehe http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html

entsprechen. Neben der Benutzeroberfläche stellt WSO2 BPS eine REST-API für die Bearbeitung der Tasks zur Verfügung. Über beide lassen sich die nötigen Operationen für die Bearbeitung der Tasks ausführen [Wsoa; Wsob].

Keine der beiden Implementierungen ist optimal für die Rolle des Human Task Managers beim Aufsetzen von IoT-Anwendungen mit OpenTOSCA geeignet. Das liegt daran, dass beide Implementierungen zwar einen Human-Task-Service besitzen, doch dieser nur in Verbindung mit der gesamten Anwendung nutzbar wäre. Allerdings benötigt der Anwendungsfall kein gesamtes Workflow Management System in der Rolle des Task Managers. Außerdem besitzt OpenTOSCA bereits eine Workflow Engine, die jedoch mit keiner der beiden Human-Task-Services kompatibel ist. Eine Migration zu einer anderen Workflow Engine ist nicht erwünscht, da dies zu zeitintensiv wäre und viele Änderungen an der Implementierung von OpenTOSCA benötigen würde. Deshalb sind beide Anwendungen zu schwergewichtig und umfangreich für die Verwaltung von Human Tasks beim Aufsetzen mit OpenTOSCA. Weiterhin müsste das Konzept der Task Types bei beiden Anwendungen neu hinzugefügt und gegebenenfalls die APIs um Funktionen erweitert werden, was Änderungen an der gesamten Implementierung mit sich ziehen würde.

Diese Punkte erfordern bei der Implementierung von [Wag10] ebenfalls Änderungen. Jedoch sind diese hier leichter umzusetzen, da diese Implementierung sich auf das Verwalten der Human Tasks beschränkt und dadurch leichtgewichtiger als jBPM und WSO2 BPS ist.

6. Zusammenfassung und Ausblick

Zu Beginn der Arbeit werden die Ziele und der Anwendungsfall definiert. Es soll ein Konzept, welches Human Tasks beim Aufsetzen von IoT-Anwendungen mit OpenTOSCA unterstützt, entwickelt und in Form eines Prototyps implementiert werden.

Zuerst wird sich mit den benötigten Grundlagen für diese Arbeit beschäftigt. Zum einen wird ein Überblick über OpenTOSCA und die darunterliegende Spezifikation TOSCA gegeben sowie ein Beispielszenario für das Aufsetzen einer IoT-Anwendung mit diesem System. Zum anderen werden die wichtigsten Aspekte der WS-Human-Task-Spezifikation vorgestellt. Diese beinhalten unter anderem die Architektur einer Human-Task-Anwendung, die Nutzerzuweisungen und die Darstellung der Tasks.

Anschließend wird auf das erarbeitete Konzept, das auf WS Human Task basiert, eingegangen. Dabei wird zuerst ein Überblick über Komponenten und ihre Beziehungen untereinander gegeben. Außerdem werden einige grundlegende Begrifflichkeiten erläutert. Das System umfasst einen Task Manager, der von OpenTOSCA Tasks erhält, und eine Smartphone-App. Diese dient zur Darstellung der Tasks für den Nutzer, der sie bearbeiten soll. Wie OpenTOSCA kommuniziert die Task Manager Client App mit dem Task Manager über eine REST-API. Für diese REST-API werden entsprechende Funktionen, die ihre Parameter in Form eines JSON-Objekts erhalten, definiert. Um die Parameter als JSON-Objekte übergeben zu können, wird die XML-Repräsentation der Tasks in eine entsprechende JSON-Repräsentation überführt. Die Grundlage für den Task Manager liefert die Implementierung aus [Wag10], die bereits die Verwaltung von Tasks unterstützt. Allerdings müssen einige Erweiterungen entsprechend des Anwendungsfalls vorgenommen werden. Diese betreffen unter anderem das Hinzufügen der REST-API oder die Erweiterung um Task Types. Am Ende wird auf die Darstellung der Tasks innerhalb der Smartphone-App eingegangen und es werden die Ansichten für die Benutzeroberfläche vorgestellt.

Daraufhin wird die Implementierung der einzelnen Komponenten aufgezeigt. Dabei werden zunächst die verwendeten Technologien genannt und die Architektur des Task Managers erklärt. Daraufhin folgt ein Überblick über die Änderungen der Implementierung des Task Managers aus [Wag10]. Vor allem wird die Erweiterung um die Konzepte der Task Types, Input- und Output-Parameter sowie die neue Implementierung des Data Access Providers beschrieben. Ein weiterer Fokus liegt auf dem Einbinden der REST-API, deren Endpunkte und Services mit Jersey bzw. Spring Framework bereitgestellt werden. Außerdem wird der Aufbau der relationalen Datenbank des Task Managers erläutert. Diese wird um einige Tabellen erweitert und auf einem MySQL-Server verwaltet. Im Anschluss werden die Implementierungsdetails der Task Manager Client App vorgestellt. Diese

wird in Android realisiert. Auch hier folgt eine Beschreibung der Architektur. Weiterhin wird darauf eingegangen, wie die Task Manager Client App die REST-API des Task Managers anspricht. Dies wird mit Hilfe der Bibliothek Volley erreicht. Die vom Konzept geforderten Benachrichtigungen werden schließlich durch Android-Notifications und einen Hintergrundservice implementiert. Die Task Manager Client App besitzt eine Datenbank, deren Tabellen durch SQLite-Funktionen angesprochen werden.

Zuletzt werden verwandte Arbeiten vorgestellt. Diese umfassen jBPM [Jbpb] und WSO2 Business Process Server [Wsob]. Beide Anwendungen sind Workflow Management Systeme, die einen Human-Task-Service basierend auf WS Human Task implementieren.

Die Ziele der Arbeit werden durch das Erstellen eines Konzepts, das Human Tasks beim Aufsetzen von IoT-Anwendungen mit OpenTOSCA unterstützt, und der Implementierung eines auf diesem Konzept basierenden Prototyps erreicht.

Ausblick

Das vorgestellte Konzept berücksichtigt nur grundlegende Punkte der WS-Human-Task-Spezifikation. Daher ist es um die noch nicht umgesetzten Aspekte erweiterbar. Es könnten noch Anhänge und Kommentare eingeführt werden. Außerdem das Weitergeben von Tasks und andere administrative Funktionen könnten umgesetzt werden. Allerdings muss beachtet werden, dass die hier vorgestellte Nutzerzuweisung über Rollen und deren Task Types, vor allem für administrative Funktionen, gegebenenfalls nicht ausreichend ist. Deshalb müssten hierfür die Generic Human Roles gänzlich eingebunden werden. Durch die Angabe der Generic Human Roles bei der Erstellung der Rollen wurde dafür schon ein Grundstein gelegt.

Allgemein lässt sich das System leicht mit zusätzlichen Funktionen und Komponenten erweitern, da jede der Komponenten eigenständig ist und der Task Manager eine entsprechende REST-API vorweist. Diese REST-API kann einfach um benötigte Funktionen erweitert werden. Dadurch ist es möglich, dass auch andere potenzielle Task Parents Tasks an den Task Manager weiterleiten und mehrere Arten der Darstellung, z.B. eine Webanwendung, unterstützt werden können.

In der aktuellen Implementierung wurden Sicherheitsaspekte nicht betrachtet. Diese könnten in weiterführenden Arbeiten berücksichtigt werden. So könnte die Sicherheit des Systems deutlich erhöht werden, wenn Nutzer eine Passwort-Authentifizierung gegenüber dem Task Manager durchführen müssten oder die Datenbank eine Verschlüsselung erhalten würde.

Die Funktionen der Smartphone-App lassen sich beispielsweise durch eine Suchfunktion für Tasks erweitern. Außerdem wäre es möglich, die Priorität von Tasks in der Sortierung der MyTasks- und der AllTask-Ansicht oder auch bei Anfragen an die API zu berücksichtigen.

Zuletzt könnte die Benutzbarkeit der Smartphone-App überarbeitet werden. Das aktuelle Layout ist sehr rudimentär und besteht größtenteils aus Feldern, die vom Nutzer ausgefüllt werden. Hier könnten künftig Auswahlmöglichkeiten, z.B. für vorhandene Task Types oder Rollen, verwendet werden. Dies erleichtert das Arbeiten mit der Smartphone-App, da sich der Nutzer nicht merken muss, welche Rollen oder Task Types im System vorhanden sind.

A. Ansichten der Task Manager Client App

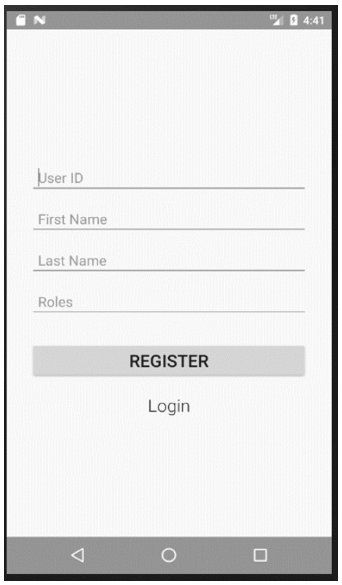


Abbildung A.1.: Register-Ansicht

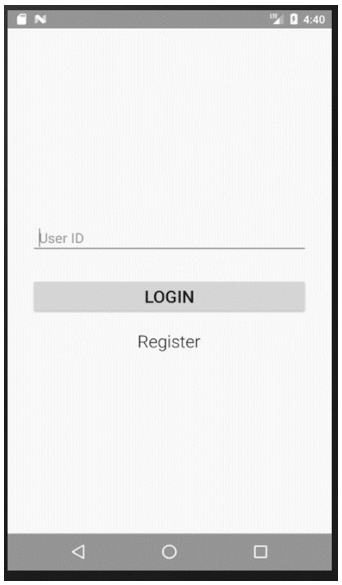


Abbildung A.2.: Login-Ansicht

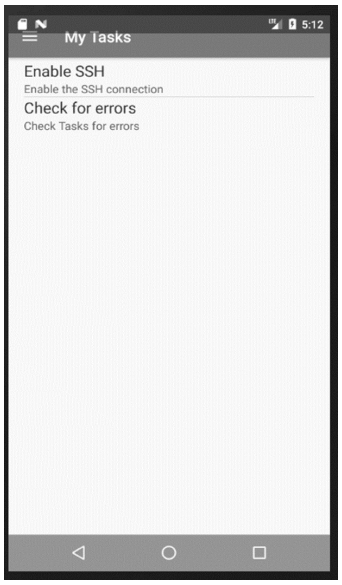


Abbildung A.3.: Liste der vom Nutzer beanspruchten Tasks

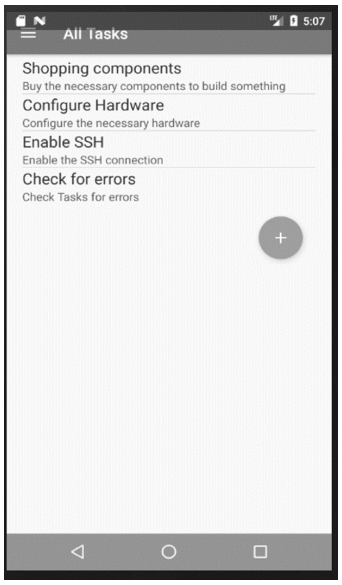


Abbildung A.4.: Liste der für den Nutzer relevanten Tasks

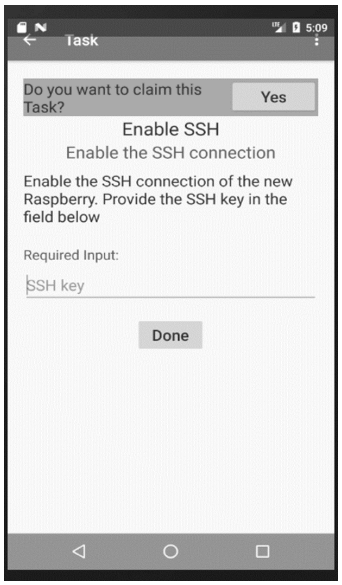


Abbildung A.5.: Task-Ansicht eines nicht beanspruchten Tasks

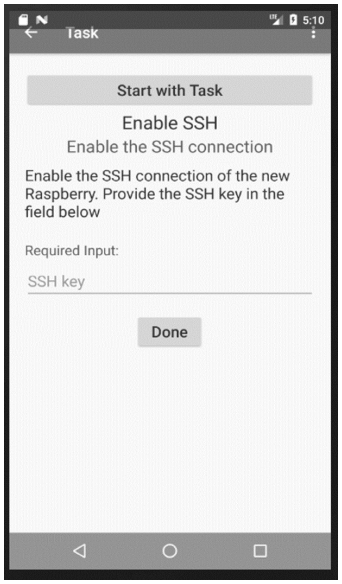


Abbildung A.6.: Task-Ansicht eines beanspruchten, aber noch nicht gestarteten, Tasks

Abbildung A.7.: Task-Ansicht eines beanspruchten, gestarteten Tasks

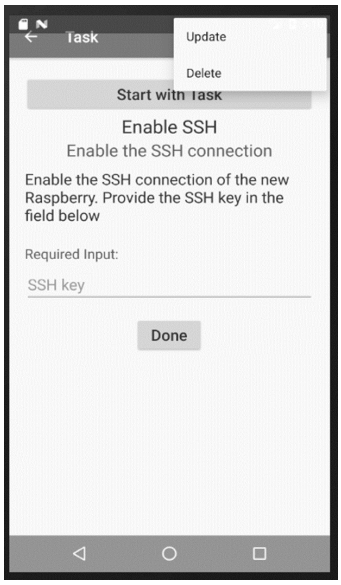


Abbildung A.8.: Menü-Ansicht, um einen Task zu bearbeiten oder zu löschen

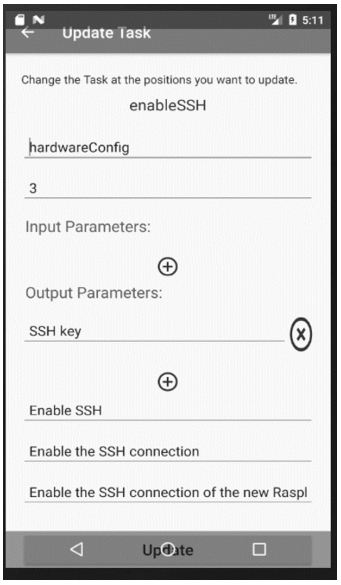


Abbildung A.9.: Update-Ansicht eines Tasks

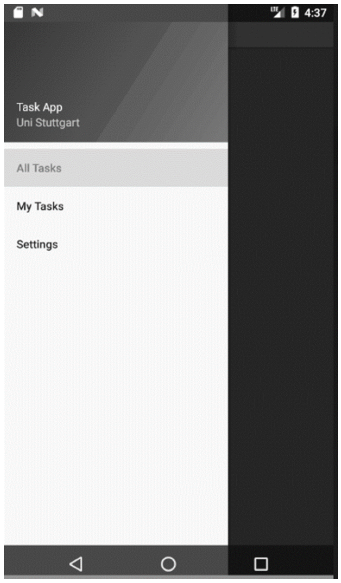


Abbildung A.10.: Navigation der App

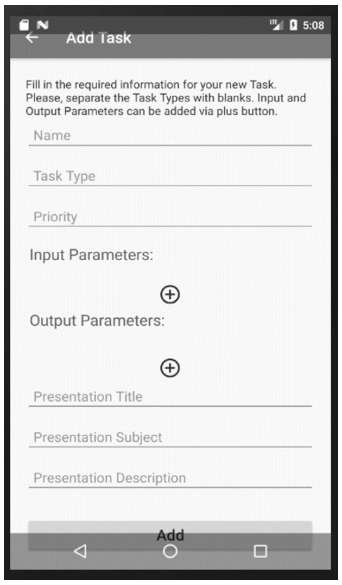


Abbildung A.11.: Erstellen eines Tasks ohne Input- und Output-Parameter

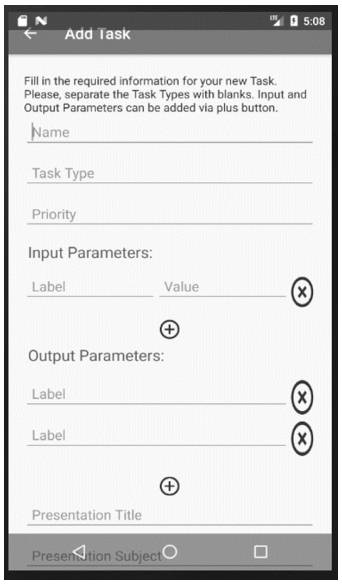


Abbildung A.12.: Erstellen eines Tasks mit Input- und Output-Parameter

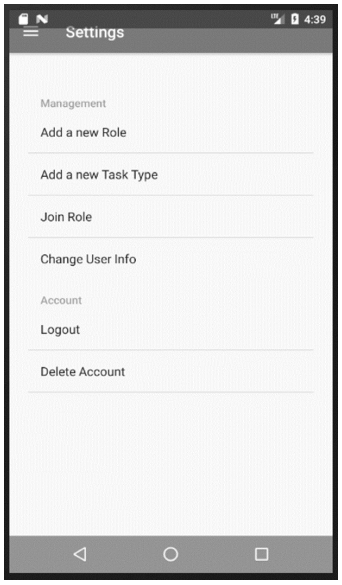


Abbildung A.13.: Einstellungen der App

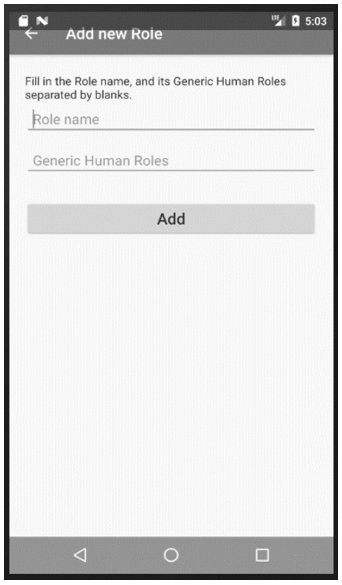


Abbildung A.14.: Hinzufügen einer neuen Rolle

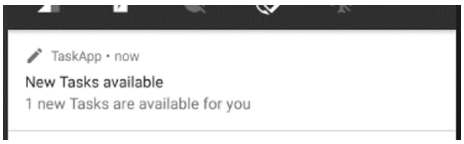


Abbildung A.19.: Beispiel der Notification der Smartphone-App

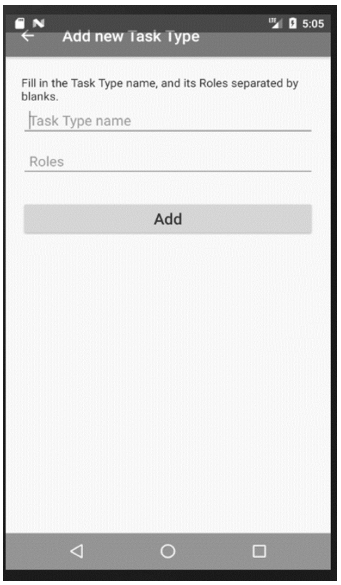


Abbildung A.15.: Hinzufügen eines neuen Task Types

Abbildung A.16.: Bearbeiten der Accountdaten

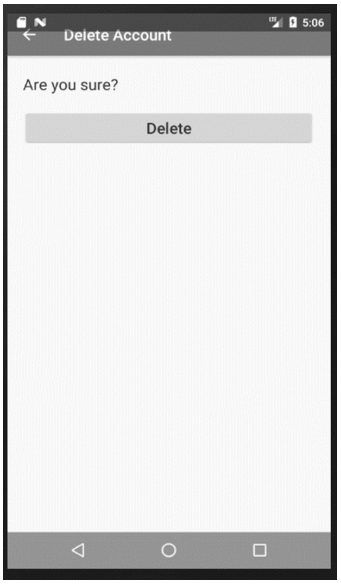


Abbildung A.17.: User-Account löschen

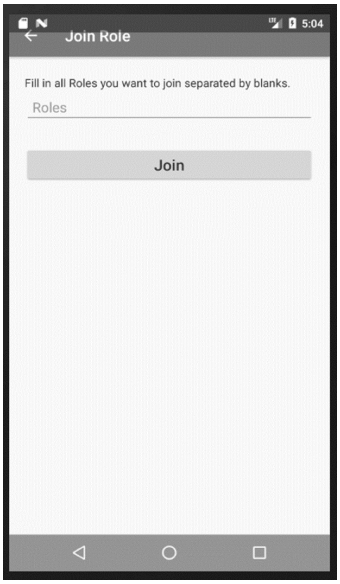


Abbildung A.18.: Einer Rolle beitreten

Literaturverzeichnis

[Atz+10] L. Atzori, A. Iera, G. Morabito. „The Internet of Things: A survey“. In: *Computer Networks* 54.15 (2010), S. 2787 –2805. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2010.05.010. URL: http://www.sciencedirect.com/science/article/pii/S1389128610001568 (zitiert auf S. 15).

[Bin+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. „OpenTOSCA – A Runtime for TOSCA-based Cloud Applications“. In: *11th International Conference on Service-Oriented Computing*. LNCS. Springer, 2013 (zitiert auf S. 15, 20).

[Bre+14] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, J. Wettinger. „Combining Declarative and Imperative Cloud Application Provisioning Based on TOSCA“. In: *2014 IEEE International Conference on Cloud Engineering*. 2014, S. 87–96. DOI: 10.1109/IC2E.2014.56 (zitiert auf S. 20).

[FT02] R. T. Fielding, R. N. Taylor. „Principled Design of the Modern Web Architecture“. In: *ACM Trans. Internet Technol.* 2.2 (Mai 2002), S. 115–150. ISSN: 1533-5399. DOI: 10.1145/514183.514185. URL: http://doi.acm.org/10.1145/514183.514185 (zitiert auf S. 40).

[Fen+09] X. Feng, J. Shen, Y. Fan. „REST: An alternative to RPC for Web services architecture“. In: *2009 First International Conference on Future Information Networks*. 2009, S. 7–10. DOI: 10.1109/ICFIN.2009.5339611 (zitiert auf S. 40).

[GT11] B. Gil, P. Trezentos. „Impacts of Data Interchange Formats on Energy Consumption and Performance in Smartphones“. In: *Proceedings of the 2011 Workshop on Open Source and Design of Communication*. OSDOC '11. Lisboa, Portugal: ACM, 2011, S. 1–6. ISBN: 978-1-4503-0873-1. DOI: 10.1145/2016716.2016718. URL: http://doi.acm.org/10.1145/2016716.2016718 (zitiert auf S. 35).

[Jbpa] *jBPM - Documentation - Chapter 7. Human Tasks*. URL: https://docs.jboss.org/jbpm/v6.0/userguide/jBPMTaskService.html (zitiert auf S. 75).

[Jbpb] *jBPM*. URL: https://www.jbpm.org/ (zitiert auf S. 75, 78).

[Nur+] N. Nurseitov, M. Paulson, R. Reynolds, C. Izurieta. „Comparison of JSON and XML data interchange formats: a case study.“ In: () (zitiert auf S. 35).

[OAS A] OASIS. *Web Services – Human Task (WS-HumanTask) Specification Version 1.1*. 17. August 2010. URL: http://docs.oasis-open.org/bpel4people/ws-humantask-1.1-spec-cs-01.html (zitiert auf S. 15, 21, 22, 31, 38, 39, 51).

[SS06] K. P. Stoilova, T. A. Stoilov. „Evolution of the workflow management systems“. In: *Scientific Conference on Information, Communication and Energy Systems and Technologies-ICEST*. 2006, S. 225–228 (zitiert auf S. 75).

[Sil+16] A. C. F. da Silva, U. Breitenbücher, K. Képes, O. Kopp, F. Leymann. „OpenTOSCA for IoT: Automating the Deployment of IoT Applications Based on the Mosquitto Message Broker“. In: *Proceedings of the 6th International Conference on the Internet of Things*. IoT'16. Stuttgart, Germany: ACM, 2016, S. 181–182. ISBN: 978-1-4503-4814-0. DOI: 10.1145/2991561.2998464. URL: http://doi.acm.org/10.1145/2991561.2998464 (zitiert auf S. 15, 20, 21).

[Sil+17] A. C. F. da Silva, U. Breitenbücher, P. Hirmer, K. Képes, O. Kopp, F. Leymann, B. Mitschang, R. Steinke. „Internet of Things Out of the Box: Using TOSCA for Automating the Deployment of IoT Environments“. Englisch. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER)*. Hrsg. von D. Ferguson, V. M. Muñoz, J. Cardoso, M. Helfert, C. Pahl. Bd. 1. ScitePress. SciTePress Digital Library, 2017, S. 358–367. ISBN: 978-989-758-243-1. DOI: 10.5220/0006243303580367. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2017-28&engl=0 (zitiert auf S. 15, 20).

[Wag10] S. Wagner. „A Concept of Human-oriented Workflows“. Englisch. Diplomarbeit. Universität Stuttgart Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2010, S. 121. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2987&engl=0 (zitiert auf S. 17, 34, 39, 50, 53–55, 57, 76, 77).

[Wsoa] *WSO2 Business Process Server*. URL: https://wso2.com/products/business-process-server/ (zitiert auf S. 75, 76).

[Wsob] *WSO2 Enterprise Integrator Documentation - Managing BPEL Processes and Human Tasks*. URL: https://docs.wso2.com/display/EI630/Managing+BPEL+Processes+and+Human+Tasks (zitiert auf S. 76, 78).

[ECMce] ECMA International. *The JSON Data Interchange Syntax*. December 2017. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (zitiert auf S. 35).

[Gooa] Google Developers. *Documentation for app developers*. URL: https://developer.android.com/docs/ (zitiert auf S. 68).

[Goob] Google Developers. *Notifications Overview*. URL: https://developer.android.com/guide/topics/ui/notifiers/notifications (zitiert auf S. 51, 71).

[Gooc] Google Developers. *Volley overview*. URL: https://developer.android.com/training/volley/ (zitiert auf S. 68, 69).

[Good] Google Developers. *android.database.sqlite*. URL: https://developer.android.com/reference/android/database/sqlite/package-summary (zitiert auf S. 68, 70).

[OAS N] OASIS. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. 25 November 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (zitiert auf S. 19, 20).

[Oraa] Oracle Corporation. *Jersey 2.27 User Guide*. URL: https://jersey.github.io/documentation/latest/index.html (zitiert auf S. 56, 57).

[Orab] Oracle Corporation. *MySQL 8.0 Reference Manual*. URL: https://dev.mysql.com/doc/refman/8.0/en/.

[OraJu] Oracle Corporation. *JAX-RS: Java API for RESTful Web Services*. 9 Juni 2017. URL: https://github.com/jax-rs/spec/blob/master/spec.pdf.

[Piv] Pivotal Software. *Spring Framework Documentation Version 5.0.9.RELEASE*. URL: https://docs.spring.io/spring/docs/5.0.9.RELEASE/spring-framework-reference/ (zitiert auf S. 56).

Alle URLs wurden zuletzt am 10. 10. 2018 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift