 Template

2017-05-03

# 

**Über arc42**

arc42, das Template zur Dokumentation von Software- und Systemarchitekturen.

Erstellt von Dr. Gernot Starke, Dr. Peter Hruschka und Mitwirkenden.

Template Revision: 7.0 DE (asciidoc-based), January 2017

© We acknowledge that this document uses material from the arc 42 architecture template, <http://www.arc42.de>. Created by Dr. Peter Hruschka & Dr. Gernot Starke.

**Note**

Diese Version des Templates enthält Hilfen und Erläuterungen. Sie dient der Einarbeitung in arc42 sowie dem Verständnis der Konzepte. Für die Dokumentation eigener System verwenden Sie besser die *plain* Version.

# Einführung und Ziele

## Aufgabenstellung

## Studierende der Hochschule München benötigen innerhalb der Bachelor-/Master-Laufbahn diverse Fachbücher/CDs um einen intensiven und schnellen Lernerfolg zu erreichen. Diese Literatur ist oftmals kostspielig und wird darüber hinaus meist nur für einen kurzen Zeitraum (für gewöhnlich ein Semester) benötigt. Für diese Problemstellung soll das "ShareIt"-System eine elegante Möglichkeit bieten, um

* Studierenden nicht mehr benötigte Fachbücher/CDs für den Verleih anzubieten.
* benötigte Fachbücher einfach und komfortabel ausleihen zu können.

## Das ShareIt-System soll dabei als zentrale Verleihbibliothek für gebrauchte Fachliteratur/CDs dienen. Studierende, welche Exemplare ausleihen/anbieten wollen, müssen sich gegenüber dem System mittels einem Token authentisieren, um Falscheingaben und Missbrauch zu vermeiden. Die Implementierung wird ohne Front-End zur Verfügung gestellt und nur über eine REST-API implementiert. Als Datenübertragungsformat dient JSON.

## Die genaue Anforderungsbeschreibung ist unter Moodle der Veranstaltung "Software-Architektur" verfügbar.

## Qualitätsziele

## Folgende Qualitätsziele sind zwingend erforderlich:

// Tabelle mit klarer Prioritätenvergabe!!

## Hochverfügbarkeit des Systems

## Konsistente Datenhaltung

## Intuitive und einfache REST-Schnittstelle

* Sichere Authentifizierung von Nutzern (Schutz vor Missbrauch)
* Portierbarkeit in andere Umgebungen, z.B. anderes Frontend (Verwendung REST-Schnittstelle)

// evtl. noch Ablauf/Aktivitätsdiagramm hinzufügen

Mit den Qualitätszielen sollen verschiedene Ziele erreicht werden. Zum einen soll Studenten dauerhaft und zu jeder Tages- und Nachtzeit die Möglichkeit gegeben werden, auf den Service zuzugreifen. Für Abschlussarbeiten ist diese Verfügbarkeit besonders wichtig. (Stichwort Hochverfügbarkeit).

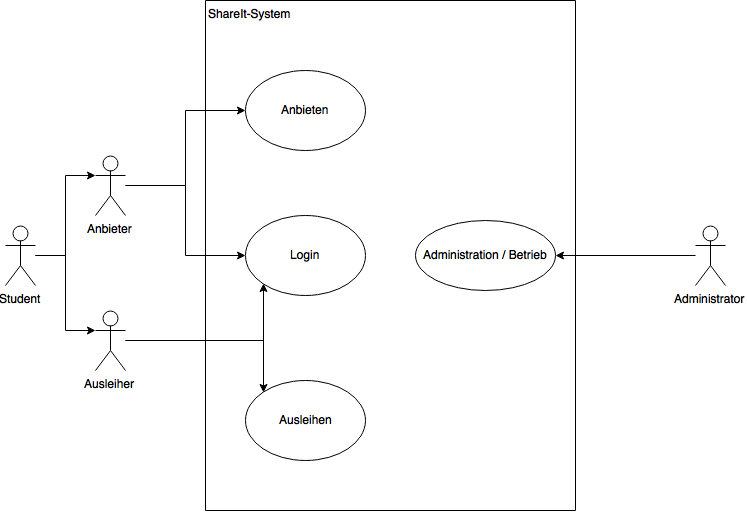
## Da Studenten ein Interesse daran haben, ihre Bücher ggf. wieder zurückzufordern, soll auch die Datenhaltung unbedingt konsistent sein. Somit ist garantiert, dass die Eigentümer wissen, wo ihre Bücher zu jedem Zeitpunkt sind. (Stichwort Konsistente Datenhaltung)

## Schließlich soll die Schnittstelle möglichst einfach bedienbar sein, sodass ein mögliches Frontend ohne größere Komplikationen darauf aufbauen kann.

## // evtl. noch tabellarische Form hinzufügen

## Stakeholder

Stakeholder des Systems sind in erster Linie die Studenten der Hochschule München sowie die Administratoren des Systems. Studenten können dabei als Ausleiher und/oder als Anbieter fungieren, daher wird deren Funktion in zwei Bereiche aufgeteilt. Folgendes Use Case Diagramm verdeutlicht die Zusammenhänge:



Für die Administratoren ist die Beschreibung des Systems wichtig, da diese das System entwickeln, betreiben sowie supporten müssen.

Die eigentliche Zielgruppe des Systems ist an die Studenten der Hochschule München gerichtet, welche somit Hauptbestandteil des Systems sind und zwingend als Stakeholder aufgenommen werden müssen.

## // tabellarische Form hinzufügen!

# Randbedingungen

Wichtige Randbedingung ist z.B., dass eine sichere Authentifizierung am System erfolgen muss. Des Weiteren muss eine klar erkennbare Schichtenarchitektur befolgt werden, sodass keine unerlaubten Abhängigkeiten innerhalb der REST-Architektur bestehen.

Grundsätzlich sollen die Bestandteile des Systems als gesamtes als OpenSource-Projekt zur Vefügung stehen. Die Implementierung soll in der Programmiersprache Java erfolgen. Eingebettet in einen Jetty-Server soll das Jersey RESTful-Webservices Framework anhand der JAX-RS Bibliothek zur Implementierung verwendet werden. Für die Persistenzierung ist Hibernate vorgesehen.

Organisatorisch: Das Entwicklerteam soll aus mindestens zwei und höchstens vier Entwickler(inne)n zusammengesetzt sein.

# Kontextabgrenzung

// REST-Schnittstellenbeschreibung hinzufügen, z.B. swagger bzw. vielleicht auch mit JavaDoc API

# Lösungsstrategie

// Tabellen vervollständigen

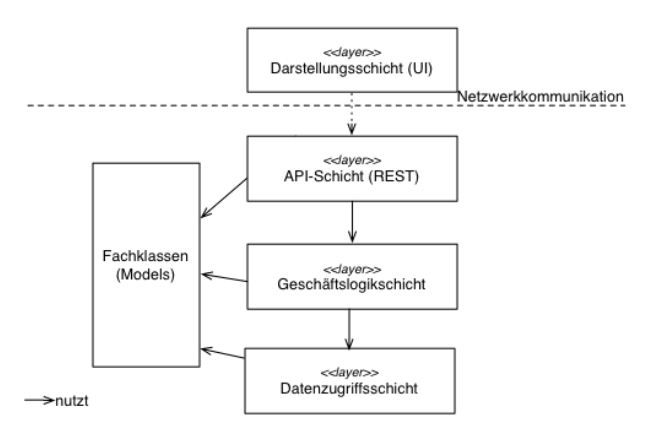
|  |  |
| --- | --- |
| Aufgabe/Problem | Lösungsansatz |
| Verwendung einer nach außen hin leicht verständlichen Technologie | Es wird bewusst auf eine Lösung mittels eines REST-Service gesetzt, da somit bereits ein bekanntes und verbreitetes Grundgerüst eines Webservice gewährleistet ist. |
|  |  |

# Die Schichtenarchitektur hilft dabei, eine möglichst klare Struktur zu schaffen und z.B. einzelne Schichten getrennt voneinander betrachten/austauschen zu können. Somit besteht z.B. die Möglichkeit, sowohl die REST-Api-Schicht als auch die Datenhaltungsschicht austauschen zu können, ohne dabei die Business Logic-Schicht anpassen zu müssen.

|  |  |  |
| --- | --- | --- |
| **Qualitätsmerkmal** | **Szenario** | **Maßnahmen** |
|  |  |  |
|  |  |  |

# Bausteinsicht

Die Architektur der Software ist wie folgt in Schichtenmodell aufgebaut:



Dabei dient die "Ressourcen"-Schicht als oberste Layer, welche alle Schnittstellen der REST-API implementiert und ggfs. JSON in Objekte umwandelt bzw. im Falle einer Antwort Fachklassenobjekte zurück in JSON umwandelt. Die Ressourcen-Schicht beinhaltet dabei keinerlei Überprüfung der ankommenden Daten. Die Prüfung auf fehlerhafte bzw. unvollständige Daten wird in der Geschäftslogikschicht durchgeführt. Diese Daten werden von der REST-API-Schicht an die Geschäftslogikschicht gesendet.

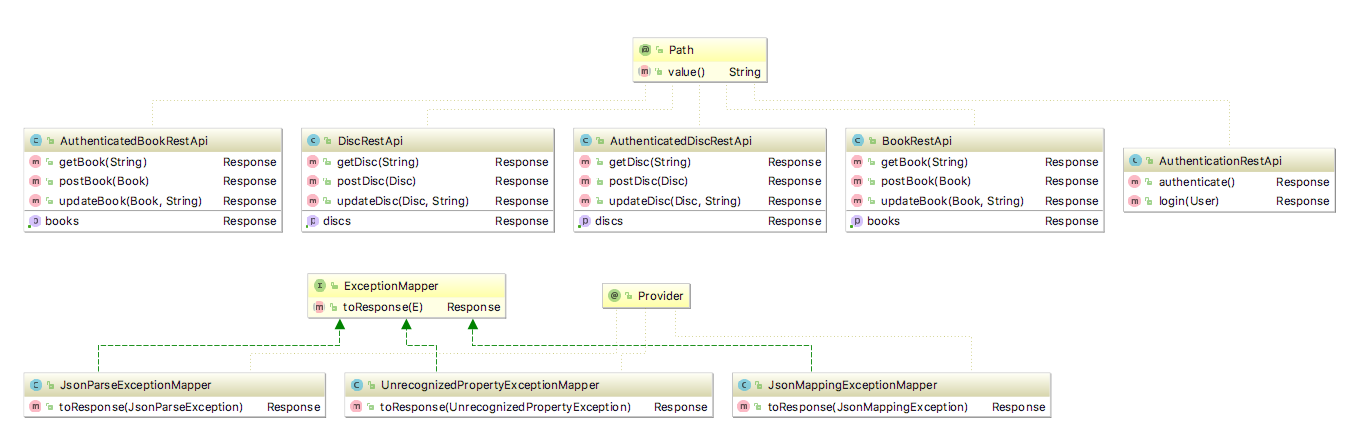
Die Geschätslogikschicht prüft die eingehenden Daten und definiert, ob Daten gespeichert werden und welche Antwort (Response) an den Client zurückgesendet wird. Dabei kommuniziert die Geschäftslogikschicht nicht direkt mit der Datenbank, sondern leitet diese Anfragen an die Datenzugriffsschicht weiter. Diese enthält Informationen zur Datenbankanbidung und stellt entsprechende Methoden für die Geschäftslogikschicht für Abfragen (inserts, update, exists, etc.) zur Verfügung.

Neben den Schichten werden sog. Fachklassen (Models) bereitgestellt, welche schichtenübergreifend Objekte für die Modellierung von Books, Discs, Mediums, Tokens und Users zur Verfügung stellt. Diese beinhalten keinerlei Logik und werden nur als Objekte verwendet, welche zu realen, abstrahierten Informationen zugeordnet werden.

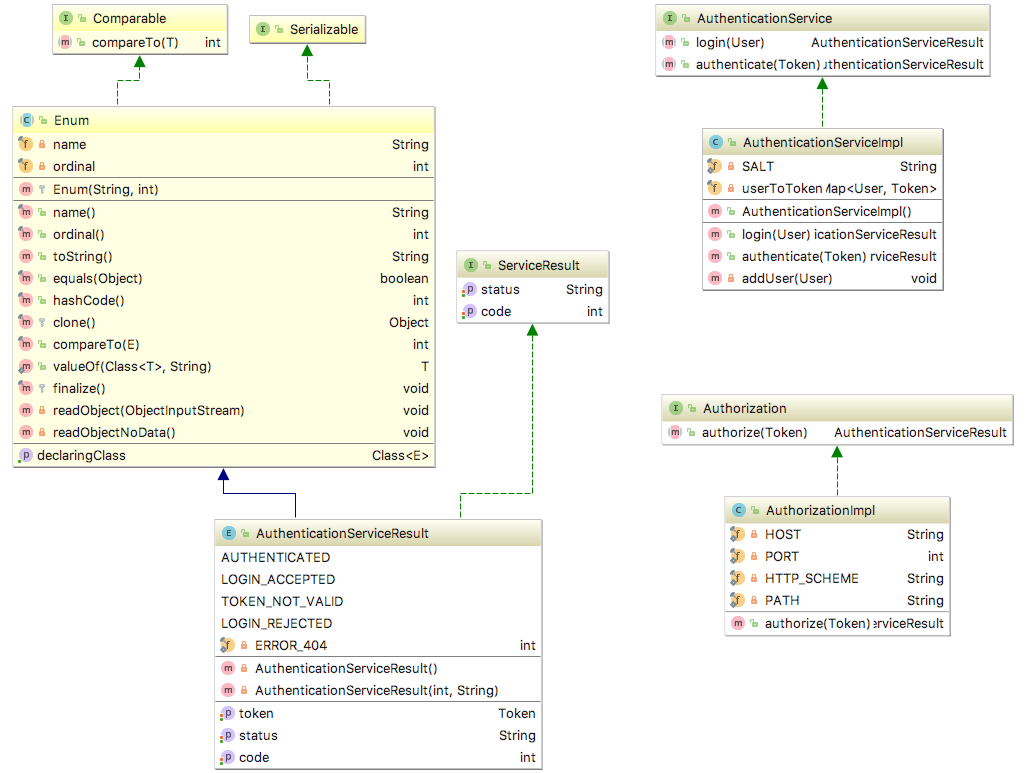
Hinweis: Die Darstellungsschicht (UI) ist nicht Bestandteil des ShareIt-Systems und muss vom Nutzer der REST-API selbst implementiert werden. (sofern benötigt).

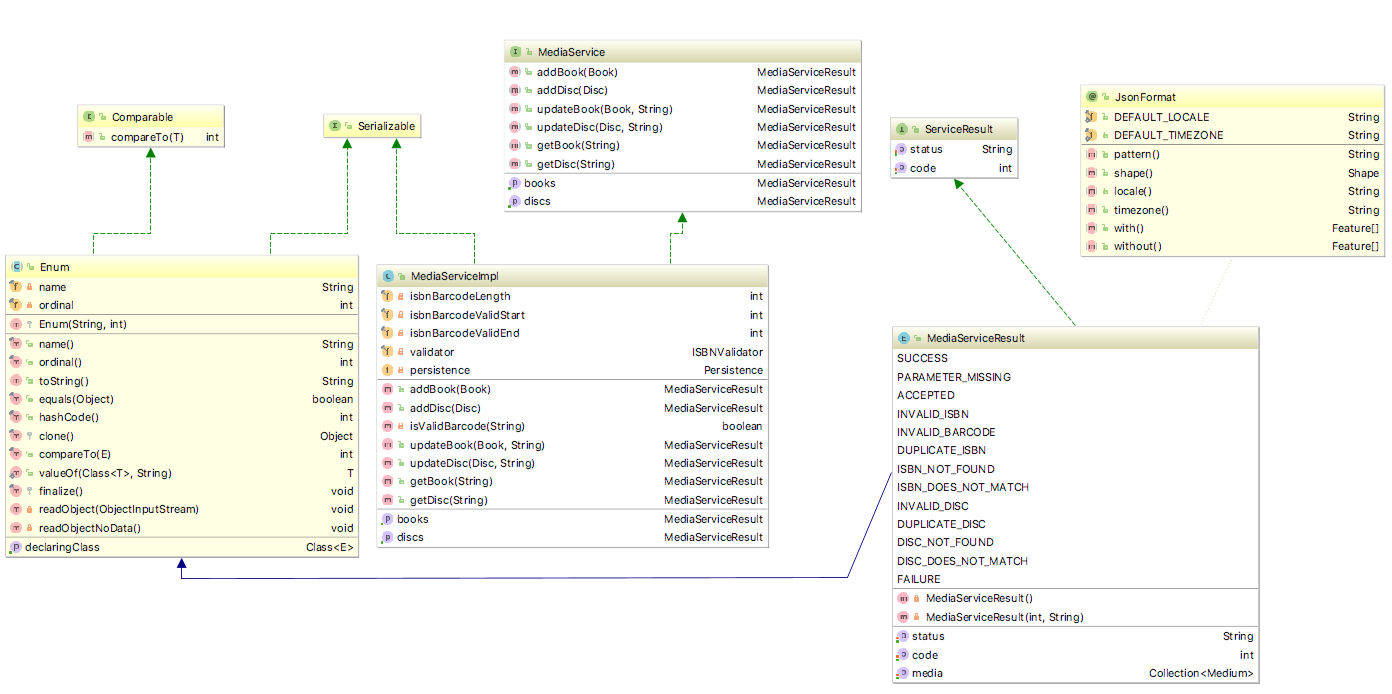
Im folgendem werden die Klassendiagramme der einzelnen Schichten dargestellt:

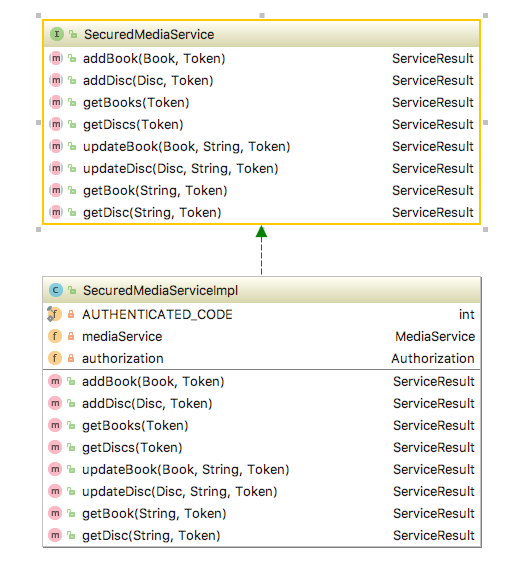
API-Schicht:



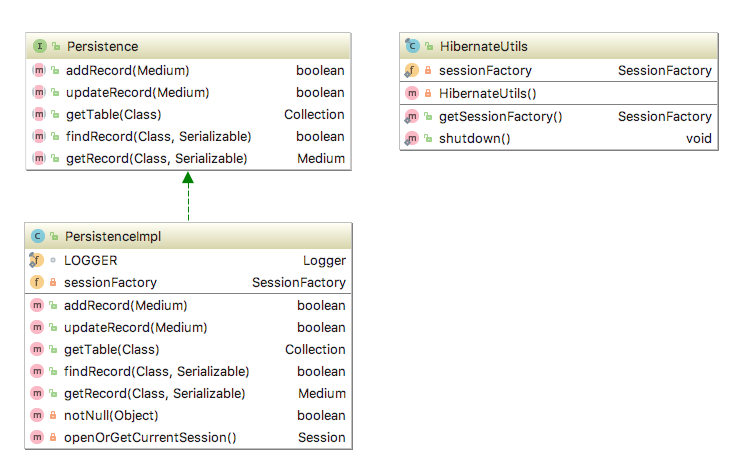
Geschäftslogikschicht:





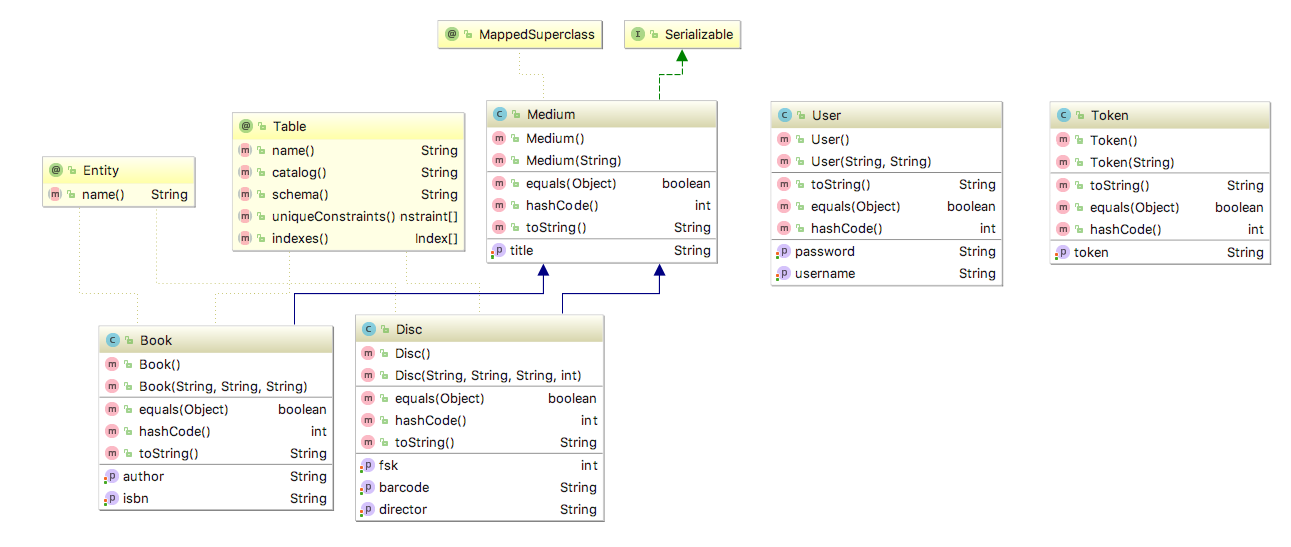


Datenhaltungsschicht:



Als Datenbank wird die In-Memory-Datenbank namens „HSQL“ eingesetzt, da diese für Testzwecke leicht zu debuggen ist. Die Datenbank wird mittels der Resourcen-Datei „hibernate.cfg.xml“ konfiguriert.

Fachklassen:



# Laufzeitsicht

## *<Bezeichnung Laufzeitszenario 1>*

* <hier Laufzeitdiagramm oder Ablaufbeschreibung einfügen>
* <hier Besonderheiten bei dem Zusammenspiel der Bausteine in diesem Szenario erläutern>

## *<Bezeichnung Laufzeitszenario 2>*

…

## *<Bezeichnung Laufzeitszenario n>*

…

# Verteilungssicht

## C:\Users\Nelson\IdeaProjects\shareit-summer-2017-teama\documentation\images\UMLDeploymentDiag.PNG

## Die Infrastruktur des ShareIt Systems ist recht übersichtlich, innerhalb des Jetty WebServers wird der ShareIt Service (inkl. Authentifizierungsservice) ausgeführt, während die Hibernate Datenbank auf der jeweiligen lokalen Maschine ausgeführt wird.

Das System wird auf Heroku unter folgender URL deployed:  
<https://stark-stream-71403.herokuapp.com>

# Querschnittliche Konzepte

**Fachliches Datenmodell**

Das fachliche Datenmodell beinhaltet die Entitäten, die für das System relevant sind, in diesem Fall Books und Discs (leiten beide von Medium ab), Copies sowie Token und User. Die jeweilige Funktion der einzelnen Entitäten ist mittels JavaDoc genau beschrieben.

**Logging**

Im gesamten System wird ein Logging-Tool verwendet (log4j), das dabei hilft, den Programmcode zur Runtime verständlicher und besser nachvollziehbar zu machen. log4j wird dabei in den jeweiligen Klassen über einen statischen "LogManager" initialisiert, der dann für die Ausgabe von Informationen, Fehlern, etc... verwendet werden kann.

Das Logging wird unter der

src/main/resources/log4j2.xml

konfiguriert.

**Sicherheit/Session Management**

Um beim Zugriff auf die verschiedenen Schnittstellen der REST-Api die Sicherheit der Daten zu gewährleisten, wurde eine querschnittliche Authentifizierung implementiert. Jeder Nutzer muss sich zunächst einloggen, wodurch er in den darauffolgenden Aktionen eindeutig identifiziert werden kann. Der unter "Fachliches Datenmodell" genannte Token wird hier einmalig generiert und dient beim Zugriff auf die Ressourcen der zuverlässigen Authorisierung.



**Einsatz des Build-Management-Tools “Maven”**

Für die Einbindung verschiedener Pakete sowie des Build-Management wird das Build-Management-Tools Maven eingesetzt. Maven wird mittels der “pom.xml” im root-Verzeichnis konfiguriert und enthält alle Abhängigkeiten zu Paketen von Drittherstellern. Maven sorgt dabei für ein einheitliches Library-Managemnet sowie einem Build-Management beim Deployment auf Heroku.

# Entwurfsentscheidungen

# Wie unter dem Punkt Lösungsstrategie bereits beschrieben, ging es bei den Entwurfsentscheidungen vor allem um die Architektur des WebServices, der den Shareit-Service bereitstellen soll. Hier standen zunächst die Alternative REST und SOAP zur Verfügung, aufgrund der höheren Flexibilität von REST (z.B. JSON und XML verfügbar), wurde letztlich REST ausgewählt.

# Qualitätsanforderungen

**Inhalt.**

Dieser Abschnitt enthält möglichst alle Qualitätsanforderungen als Qualitätsbaum mit Szenarien. Die wichtigsten davon haben Sie bereits in Abschnitt 1.2 (Qualitätsziele) hervorgehoben.

Nehmen Sie hier auch Qualitätsanforderungen geringerer Priorität auf, deren Nichteinhaltung oder -erreichung geringe Risiken birgt.

**Motivation.**

Weil Qualitätsanforderungen die Architekturentscheidungen oft maßgeblich beeinflussen, sollten Sie die für Ihre Stakeholder relevanten Qualitätsanforderungen kennen, möglichst konkret und operationalisiert.

## Qualitätsbaum

**Inhalt.**

Der Qualitätsbaum ( a la ATAM) mit Qualitätsszenarien an den Blättern.

**Motivation.**

Die mit Prioritäten versehene Baumstruktur gibt Überblick über die oftmals zahlreichen Qualitätsanforderungen.

* Baumartige Verfeinerung des Begriffes „Qualität“, mit "Qualität" oder Nützlichkeit als Wurzel.
* Mindmap mit Q-Oberbegriffen als Hauptzweige

In jedem Fall sollten Sie hier Verweise auf die Szenarien des folgenden Abschnittes aufnehmen.

## Qualitätsszenarien

**Inhalt.**

Konkretisierung der (in der Praxis oftmals vagen oder impliziten) Qualitätsanforderungen durch (Qualitäts-)Szenarien.

Diese Szenarien beschreiben, was beim Eintreffen eines Stimulus auf ein System in bestimmten Situationen geschieht.

Wesentlich für die meisten Softwarearchitekten sind zwei Arten von Szenarien:

* Nutzungsszenarien (auch genannt Anwendungs- oder Anwendungsfallszenarien) beschreiben, wie das System zur Laufzeit auf einen bestimmten Auslöser reagieren soll. Hierunter fallen auch Szenarien zur Beschreibung von Effizienz oder Performance. Beispiel: Das System beantwortet eine Benutzeranfrage innerhalb einer Sekunde.
* Änderungsszenarien beschreiben eine Modifikation des Systems oder seiner unmittelbarer Umgebung. Beispiel: Eine zusätzliche Funktionalität wird implementiert oder die Anforderung an ein Qualitätsmerkmal ändert sich.

**Motivation.**

Szenarien operationalisieren Qualitätsanforderungen und machen deren Erfüllung mess- oder entscheidbar.

Insbesondere wenn Sie die Qualität Ihrer Architektur mit Methoden wie ATAM überprüfen wollen, bedürfen die in Abschnitt 1.2 genannten Qualitätsziele einer weiteren Präzisierung bis auf die Ebene von diskutierbaren und nachprüfbaren Szenarien.

**Form.**

Entweder tabellarisch oder als Freitext.

# Risiken und technische Schulden

Es wurde bewusst kein Framework für die Authentifizierung der REST-Benutzer eingesetzt, da die Integration für die Größe des Projekts sowie den zeitlichen Rahmen nicht angemessen wäre. Im Falle eine Erweiterung des ShareIt-Systems muss der Einsatz eines Authentifizierungs-Frameworks wie z. B. OpenID bzw. OAuth2 neu bewertet werden.

Aktuell müssen Benutzer manuell vom Administrations-Team hinzugefügt werden. Eine Registrierung sowie Entfernung eines Accounts im ShareIt-System ist für einen Benutzer der REST-Schnittstelle aktuell nicht möglich und wurde aufgrund mangelnder Ressourcen derzeitig noch nicht umgesetzt.

# Glossar

|  |  |
| --- | --- |
| Begriff | Definition |
| *ShareIt* | *Name des Projekts* |
| *REST-Service* | *Ein standardisiertes Schnittstellenkonzept auf Basis des HTTP-Protokolls* |
| *JAX-RS* | *Framework zur Umsetzung des REST-Services* |
| *JAX-B* | *(Un-)Marshalling-Framework von JSON oder XML-Objekten* |
| *(Un-)Marshalling* | *(De-)Serialisieren von Fachklassenobjekten zu einem einheitlichem Format (z. B. JSON oder XML)* |
| *Jetty* | *Web-Applications-Container mit Servlet-Funktionalität* |
| *OpenSource* | *Kostenlose sowie quelloffene Software* |