HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

Rapid Online Engineering
forventis
inventive solutions

PocketDoc – App für die Selbstdiagnose

Studienarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Herstsemester 2014

Autor(en):	Nathan Bourquin
	Oliver Frischknecht
Betreuer:	Dr. Eduard Glatz
Projektpartner:	Forventis GmbH, Wildbachstrasse 50, 8008 Zürich
	Panter AG, Zentralstrasse 37, 8003 Zürich
Gegenleser:	Dr. Eduard Glatz

PocketDok - App für die Selbstdiagnose

Nathan Bourquin Oliver Frischknecht

12. März 2015

Abstract

Um die Routinearbeit von Ärzten zu automatisieren wurde ein Algorithmus entwickelt, womit sich Menschen jederzeit überall einen medizinischen Rat einholen können, ohne dafür beim Arzt vorbeizugehen. Anhand von Ja-/Nein Symptomabfragen können zurzeit rund 50 Bagatellerkrankungen zuverlässig erkannt werden. PocketDoc ermittelt auf Grund der Antworten, welche Bagatellkrankheiten in Frage kommen und empfiehlt dann entsprechende Therapien / Vorgehensschritte. Diese Automatisierung eröffnet ein markantes gesellschaftliches und ökonomisches Potential.

Ziel dieser Arbeit war es auf Grund eines bestehenden Prototypen und einem Excel File, worin die ganze Logik der Diagnostizierung vorhanden war, ein Backend mit Anschluss an eine Datenbank und dazu ein Verwaltungstool zu entwickeln.

Das Resultat der Arbeit besteht aus drei Komponenten. Es wurde eine Postgres Datenbank aufgesetzt, welche die Struktur des Excel Files wiederspiegelt. Zudem wurde ein Backend realisiert, welches den „Morbi cognitio"-Algorithmus beinhaltet. Der Algorithmus stellt den Kern der Selbstdiagnostizierung dar und basiert auf einer robusten Datenbasis, welche der Kunde auf Grund seiner Erfahrung im Verwaltungstool erfasst. Das Backend bietet eine auf REST basierende Schnittstelle für das Verwaltungstool und wurde mit Java Servlets umgesetzt. Als weiteres Resultat entstand zusammen mit HTML und AngularJS ein dynamisches, webbasiertes Verwaltungstool.

PocketDoc bietet einen erweiterbaren Grundbaustein für den Kunden und unterstützt ihn bei der Weiterentwicklung einer App und dem Verwalten der Daten, durch eine übersichtliche Darstellung.

Inhaltsverzeichnis

Teil I

Technischer Bericht

1 Management Summary

1.1 Ausgangslage

Wer kennt es nicht? Kopfschmerzen, Bauchschmerzen, Müdigkeit und Grippe. Es stellt sich nur immer die Frage, ist ein Besuch beim Doktor notwendig? Viele Hausärzte sind dadurch extrem ausgelastet. Die beiden Firmen Forventis GmbH [6] und Panter AG [18] haben einen Algorithmus entwickelt mit welchem zurzeit rund 50 Bagatellerkrankungen anhand von Ja-/Nein Symptomabfragen zuverlässig erkannt werden. Im Falle einer Bagatellkrankheit gibt das System Therapieempfehlungen ab. Bisher ist die Logik dieses Algorithmus in einer Excel-Datei [14] implementiert. Die grosse Komplexität und die limitierten Abstraktionsmöglichkeiten hindern den Fortschritt der Forschung. Die Aufgabe dieser Arbeit ist es den Kunden mit einem robusten Grundstein zu unterstützen. Der Algorithmus soll in Java Code umgesetzt werden und die Datenbasis soll in einer Datenbank gespeichert werden können. Dies fördert die einfache Erweiterbarkeit der Daten und eine einfache Anpassung des Algorithmus. Mittels eines Admin Tools, welches über den Browser verfügbar ist, soll der Kunde die Datenbasis einfach erweitern und testen können.

1.2 Vorgehen, Technologien

Das bereits existierende Excel File bietet nicht die optimale Anpassungsmöglichkeit für den Kunden. In einem ersten Schritt werden die Daten daraus in eine PostgreSQL [31] Datenbank extrahiert. Danach wird mit Java ein Backend entwickelt, welches für die Berechnung und Aufbereitung von Daten zuständig ist. Es wird im Servletcontainer Tomcat ausgeführt und bietet eine auf REST basierte Schnittstelle [33], welche dann zur Kommunikation mit dem Admin Tool benötigt wird. In einem letzten Schritt wird ein Admin Tool mit HTML und AngularJS [25] entwickelt, es stellt eine dynamische Webseite dar, welche sofort auf alle Änderungen reagiert, somit bietet sich eine angenehme Möglichkeit die Daten zu bearbeiten.

1.3 Ergebnisse

Das Ergebnis des Projekt „PocketDoc“ ist ein einfach bedienbares flaches Web Interface, welches speziell für den Firefox entwickelt wurde. Dieses bietet die Möglichkeit die Datenbasis für den Algorithmus bequem zu bearbeiten. Dabei greift das Admin Tool über eine RESTful API auf die Datenbank zu und bietet somit alle CRUD Operationen auf die gesamte Datenbasis. Das Backend ist ein zweites Ergebnis des Projekts. Zum einen bietet es eine klar definierte Schnittstelle an, um über HTTP Requests die Datenbasis zu bearbeiten. Zum anderen wertet das Backend die Antworten aus dem Testlauf aus und berechnet die nächste Frage und die Punkteverteilung.

1.4 Ausblick

Die Vision des Kunden ist es, dass Menschen sich jederzeit, überall und unabhängig von ihrem Versicherungsmodell einen ersten medizinischen Rat holen können. Das von uns entwickelte Backend bietet den Grundstein, um ein App-basiertes medizinisches Expertensystem zu entwickeln. Diese App ist ein erster Schritt in die Richtung der Automatisierung von grundlegenden ärztlichen Tätigkeiten und eröffnet damit gesellschaftliches und ökonomisches Potential.

2 Anforderungsanalyse

2.1 Use Cases

2.1.1 Admin Tool

2.1.1.1 Use Case Diagramm

Das folgende Use-Case Diagramm beschreibt alle Aktionen die ein User bei der Benutzung des Systems ausführen kann. Im Kapitel 2.1.1.2 werden alle Use Cases grob beschrieben. Einzelne wichtige Use Cases werden dann im Kapitel 2.1.1.3 genauer beschrieben.

Abbildung 1: Use Case Übersicht

2.1.1.2 Use Case Übersicht

Nr.	Name	Ziel	Primary Actor
UC1	Diagnose verwalten	Diagnose ist hinzugefügt, bearbeitet, angezeigt oder gelöscht	Admin
UC2	Frage/Antwort verwalten	Frage/Antwort ist hinzugefügt, bearbeitet, angezeigt oder gelöscht	Admin
UC3	Handlungsempfehlung verwalten	Handlungsempfehlung ist hinzugefügt, bearbeitet, angezeigt oder gelöscht	Admin
UC4	Syndrom verwalten	Syndrom ist hinzugefügt, bearbeitet, angezeigt oder gelöscht	Admin
UC5	Testregel verwalten	Testregel ist hinzugefügt, bearbeitet, angezeigt oder gelöscht	Admin
UC6	Einstellung verwalten	Setting ist bearbeitet oder angezeigt	Admin
UC7	Symptomvergleich verwalten (*Feature*)	Der Benutzer kann einstellen wie die Handlungsempfehlungen reagieren wenn eine Fragerunde zum zweiten mal durchgeführt wird. Dieser Use Case wurde als Feature deklariert.	Admin
UC8	Regeln überprüfen	Es kann überprüft werden, ob eine bestimmte Diagnose durch den Zusammenschluss von Antworten erreichbar ist.	Admin
UC9	Diagnose Testlauf	Es kann überprüft werden, ob die Fragen in einer sinnvollen Reihenfolge gestellt werden. Zudem kann anhand der abgegebenen Antworten überprüft werden, welche Diagnosen und welche Handlungsempfehlungen damit erreicht werden.	Admin

Tabelle 1: Use Case Übersicht

2.1.1.3 Use Case Beschreibung

Die Use Cases 1, 2, 3, 4, 5 und 6 funktionieren alle nach dem gleichen Prinzip. Aus diesem Grund wird nur UC1 genauer Beschrieben.

Nr.	UC1
Name	Diagnose verwalten
Primary Actor	Admin
Preconditions	Es sind bereits Diagnosen und Fragen vorhanden.
Main Success Scenario	1. Dem Benutzer wird eine Liste von allen Diagnosen angezeigt 2. Der Benutzer wählt eine Diagnose aus 3. Dem Benutzer werden alle Eigenschaften einer Diagnose angezeigt, zudem werden alle Scoreverteilungen, welche für Diagnosen möglich sind angezeigt (Scoreverteilung Antworten zu Diagnosen). 4. (optional) Der Benutzer verändert die Eigenschaft einer Diagnose 5. (optional) Die Änderungen werden dem Backend mitgeteilt
Postconditions	Alle Veränderungen sind in der Datenbank eingetragen
Alternative Flow	4. (a) Der Benutzer verändert eine Scoreverteilung (b) Der Benutzer löscht eine Diagnose
Ergänzungen	Keine

Tabelle 2: Use Case 1: Diagnose verwalten

Nr.	UC8
Name	Regeln überprüfen
Primary Actor	Admin
Preconditions	Es sind bereits Antworten, Diagnosen und Testregeln vorhanden.
Main Success Scenario	1. Benutzer startet den Testdurchlauf 2. Testantworten werden an Backend geschickt. 3. Backend rechnet das Resultat mithilfe der Antworten aus, überprüft diese und antwortet dem Frontend. 4. Es wird angezeigt ob die geschwünschte Diagnose erreicht werden konnte. Zudem wird die Rangliste der Diagnosen mit DG-Punkte angezeigt.
Postconditions	
Ergänzungen	Keine

Tabelle 3: Use Case 8: Regeln überprüfen

Nr.	UC9
Name	Diagnose Testlauf
Primary Actor	Admin
Preconditions	Es sind bereits Fragen, Antworten, Diagnosen und Handlungsempfehlungen vorhanden
Main Success Scenario	1. Frage wird gestellt 2. User beantwortet Frage 3. Antwort wird dem Backend mitgeteilt 4. Backend ermittelt Diagnosen und Handlungsempfehlungen 5. Admin Tool erhält Diagnosen und Handlungsempfehlungen 6. Zeige Diagnosen und Handlungsempfehlungen 7. (optional) User will weitere Fragen beantworten
Postconditions	Keine
Alternative Flow	3. (a) Der User will die Fragerunde jetzt abbrechen und das Resultat sehen (b) Es gibt keine weiteren Fragen (c) Wiederhole vorgehen (zurück zu 1.)
Ergänzungen	

Tabelle 4: Use Case 9: Diagnose Testlauf

2.1.2 Backend

Folgende Aufgaben sollen durch das Backend abgedeckt werden:

- „Morbi cognitio“-Algorithmus
- Vielversprechendste nächste Frage berechnen
- Handlungsempfehlungen berechnen

2.1.2.1 Morbi cognitio Algorithmus

Dieser Algorithmus bestimmt das Verhalten der Selbstdiagnose und war eine Vorgabe des Kunden. Um dem Algorithmus in diesem Projekt einen Namen zu geben erbot sich die lateinische Übersetzung von Diagnose-Algorithmus. [17] Auf Grund der erhaltenen Antworten vom User berechnet der Algorithmus eine Rangliste von Diagnosen. Jede Antwort hat eine Punkteverteilung, also pro Diagnose einen Wert, welcher dafür sorgt, dass beim Auftreten der Antwort Punkte auf die Diagnosen verteilt werden. Es resultiert also eine Punktetabelle und je nachdem wie oft eine Diagnose schon in „pole position“ ist und wie weit, punktemässig, die Diagnose von der 2. platzierten entfernt ist, liefert der Algorithmus eine Diagnose. Bei verfrühtem abbrechen der Fragerunde wird die ersteplatzierte Diagnose angegeben.

$$DGScore = \sum Punkteverteilung von Antworten \tag{1}$$

2.1.2.2 Vielversprechendste nächste Frage

Die oben erwähnte Punktetabelle dient als Grundlage für die Bestimmung der vielversprechendsten nächsten Frage. Es soll also aus den Fragen folgende ausgesucht werden, welche die grösste Differenz zwischen erst- und zweitplatzierter Diagnose aufweist. Eine vielversprechendste nächste Frage kann erst berechnet werden, wenn alle Fragen welche Informationen zur Person liefern beantwortet wurden. Zur Berechnung gibt es zwei Ansätze:

Beim ersten Ansatz wird zu jeder möglichen Antwort eine virtuelle DG Tabelle erstellt. Anschliessend wird verglichen bei welchen virtuellen Tabelle der Abstand der ersten Diagnose zur zweiten Diagnose am grössten ist, wobei die alte Top 1 Diagnose nicht an erster Stelle stehen muss. Ist die zu diesem Punkt berechnete Frage von einer Antwort abhängig, dann wird die höchste Frage der Abhängigkeitskette als vielversprechendste nächste Frage genommen.

Beim zweiten Ansatz wird zu jeder Frage ein "Discriminance-Wert"berechnet. Dieser berechnet für jede Antwort den Unterschied zwischen der erst- und der zweitplatzierten Diagnose. Dabei wird jede weitere Frage berücksichtigt die von dieser Antwort abhängig ist.

Bei beiden Ansätzen werden Fragen bevorzugt, die abhängig zur zuletzt gegebene Antwort sind. Der Vorteil vom ersten Ansatz ist die Berücksichtigung aller Diagnosen nicht nur der obersten Beiden, es gibt jedoch einen Performance Nachteil. Dieser Performance Nachteil ist jedoch nicht weiter schlimm, da Computer heutzutage so Leistungsfähig sind, dass es für diese Datenmenge nur Millisekunden braucht um sie zu verarbeiten.

2.1.2.3 Handlungsempfehlungen

Die vom User angegebenen Antworten werden zusätzlich analysiert. Gewisse Symptome und Informationen bilden zusammen Syndrome. Auf Grund der Syndrome und der Symptome wird, ähnlich wie beim „Morbi cognitio"-Algorithmus, eine zweite Punktetabelle erstellt. Dabei werden die Symptom- und die Syndrompunkte einfach zusammengezählt. Aus dieser zweiten Punktetabelle werden dann schlussendlich die Hanldungsempfehlungen berechnet.

$$Score = \sum PunkteverteilungSymptome + \sum PunkteverteilungSyndrome \tag{2}$$

2.1.2.4 Beispiel

Man nehme ein System mit zwei Fragen und zwei Diagnosen. Die Scoreverteilung sieht folgendermassen aus:

Frage	Antwort	Diagnose 1	Diagnose 2	Diagnose 3
1	Ja	10	0	40
	Nein	0	30	5
2	Ja	10	20	0
	Nein	5	0	0

Tabelle 5: Antwort zu Diagnosen Scoreverteilung

Nun angenommen es wurde noch keine Frage beantwortet. Die vielversprechendste nächste Frage wäre in diesem Fall die Frage 1, da die Antwort Ja in der Diagnoserangliste die grösste Differenz zwischen der erst- und der zweitplatzierten Diagnose bewirkt. Es verteilt 40 Punkte auf Diagnose 3.

Würde der Benutzer beim Fragedurchlauf beide Fragen mit Ja beantworten, dann hätte Diagnose 1 20 Punkte, Diagnose 2 20 Punkte und Diagnose 3 40 Punkte. Somit wäre Diagnose 3 die höchste Diagnose.

Da die Berechnung von Handlungsempfehlungen gleich funktioniert wird für das nächste Beispiel die gleiche Scoreverteilung nochmal genomen nur werden die Diangosen mit Handlungsempfehlungen ersetzt. Die Antworten sind aus der Tabelle 5 von oben nach unten durchnummeriert. Zudem werden folgende Syndrome deklariert:

Syndrom	Antworten	Handl. 1	Handl. 2	Handl. 3
1	1, 3	60	0	0
2	2, 3	0	20	0

Tabelle 6: Syndrome zu Hanldungsempfehlungen Scoreverteilung

Werden nun Fragen 1 und 2 beide mit Ja beantwortet, dann wird Syndrom 1 aktiviert. Die Rangliste wäre Handlungsempfehlung 1 80 Punkte, Handdlungsempfehlung 2 20 Punkte und Handlungsempfehlung 3 40 Punkte.

2.2 Nicht-funktionale Anforderungen

2.2.1 Benutzbarkeit

2.2.1.1 Nicht-blockierend

Beim Erfassen einer Diagnose werden verschiedene Testregeln überprüft. Diese Überprüfung braucht Zeit und stellt Anfragen an das Backend. Es ist wichtig, dass dieser Vorgang nicht dazu führt, dass das User Interface (Admin Tool) einfriert. Es soll dem Benutzer visualisiert werden, dass das Programm im Hintergrund diesen Vorgang ausführt.

Beim Testdurchlauf wird ebenfalls mit Hilfe des Backends gearbeitet vor allem bei der Ermittlung der Diagnose und Handlungsempfehlungen, jedoch auch beim Laden der nächsten Frage. An den erwähnten Stellen muss dem Benutzer grafisch angezeigt werden, dass das Programm am Arbeiten ist.

2.2.1.2 Bedienbarkeit

Das Admin Tool soll so aufgebaut sein, dass es für den Kunden möglich ist mit 3 Mausklicks auf jede Funktionalität zugreifen zu können. Die Navigation im Programm soll also nicht verschachtelt sein, sondern eine flache Hierarchie aufweisen.

2.2.2 Sicherheit

2.2.2.1 Datenschutz

Die Datensicherheit ist durch Heroku [11] vorgegeben. Die Datenbanken sind mittels SSL Verbindungen ansprechbar. Die Daten könnten verschlüsselt abgespeichert werden jedoch wird dies nicht gemacht.

2.2.3 Leistung

2.2.3.1 Datenbankverbindungen

Es sollen nie mehr als 20 Verbindungen zur Datenbank gleichzeitig bestehen. Sonst sollte die Architektur nochmals überarbeitet werden.

2.2.3.2 Antwortzeit

Die Antwortzeit kann nicht gewährleistet werden, da Heroku die volle Kontrolle über das Trafficing und die zur Verfügung gestellte Rechenleistung hat.

2.2.3.3 Skalierbarkeit

Durch die Verwendung von Heroku ist die Skalierbarkeit extrem erleichtert der Kunde kann auf einen Knopfdruck mehr Leistung/Speicher/Traffic usw. anfordern.

2.2.4 Portierbarkeit

Das Admin Tool soll für die neuste Version von Mozilla Firefox optimiert sein. Und lediglich im Browser funktionieren.

2.2.5 Wartbarkeit

2.2.5.1 Erweiterbarkeit

Das Backend soll einfach um Features erweitert werden können. Um dies zu gewährleisten wird beim Design auf eine modulare Architektur geachtet.

2.2.6 Internationalisierung

Das Admin Tool soll ausschließlich in Deutsch geführt werden.

2.3 Domainmodell

Auf folgendem Bild ist das Domainmodell der PocketDoc Domäne zu sehen. Das Domainmodell soll nur das Problemmodell abstrakt darstellen und noch keine Lösungen präsentieren.

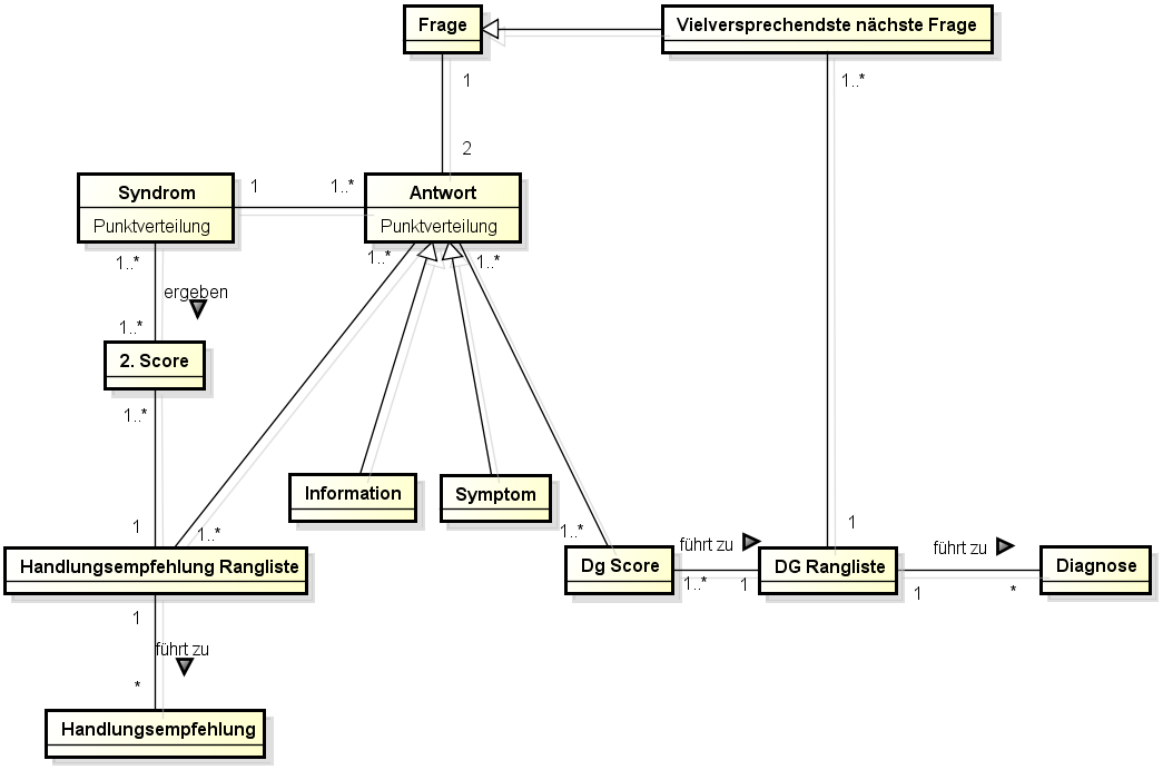


Abbildung 2: Das Domainmodell

Das Domainmodell besteht aus folgenden Objekten.

- **Frage:** Die Fragen werden dem Benutzer gestellt. Eine Frage hat zwei Antworten und kann von der Antwort einer anderen Frage abhängig sein.
- **Antwort:** Die möglichen Antworten auf eine Frage. Die zwei Möglichkeiten sind Ja und Nein.
- **Symptom:** Ein Symptom ist eine Frage, welche durch die Beantwortung mit Ja zeigt was der Benutzer für ein Leiden hat.
- **Information:** Eine Information ist eine Frage, welche durch die Beantwortung dem System mehr Aufschluss über den Benutzer und dessen Person gibt. Darunter Fallen z.B. das Alter oder das Geschlecht.
- **Syndrom:** Mehrere Antworten zusammen. Werden nur für Handlungsempfehlungen gebraucht.
- **Dg Score:** Aus verschiedenen Antworten werden den verschiedenen Diagnosen ein Score vergeben.
- **2. Score:** Aus verschiedenen Antworten und Syndromen werden den verschiedenen Handlungsempfehlungen ein Score vergeben.
- **Handlungsempfehlung:** Ein Tipp wie der Benutzer als nächstes vorgehen sollte.
- **Diagnose:** Die Diagnose ist die Bezeichung davon, was der Kunde warscheinlich hat.
- **DG Rangliste:** Die Rangliste der verschiedenen Diagnosen, nach dem DG Score geordnet.
- **Handlungsempfehlungs Rangliste:** Die Rangliste der verschiedenen Handlungsempfehlungen, nach dem 2. Score geordnet.

- **Vielversprechendste nächste Frage:** Damit dem Benutzer eine sinnvolle Frage gestellt wird um seine Diagnose so schnell wie möglich zu ermitteln, wird aus der DG Rangliste die beste nächste Frage berechnet.

Im Domainmodel sieht man gut, dass für die Berechnung der Handlungsempfehlungen sowohl die Symtpome als auch die Syndrome verwendet werden. Die Diagnose hingegen wird ausschliesslich aus den aufgetretenen Symptomen berechnet.

3 Design

3.1 Architektur

3.1.1 Überblick

Die Abbildung 3 zeigt die grobe Struktur unserer Systemkomponente. Das Admin Tool ist das Werkzeug, um alle Daten zu bearbeiten und die Eingaben anhand eines Testlaufs zu überprüfen. Dabei spielt die enge Zusammenarbeit zwischen Admin Tool und Backend eine entscheidende Rolle. Das Admin Tool beinhaltet nämlich keinerlei Business Logik sondern ist nur eine Präsentation der Daten von der Datenbank. Die Business Logik befindet sich ausschliesslich im Backend. Dort ist der „Morbi Cognitio“-Algorithmus und auch der Algorithmus zur Bestimmung der nächsten vielversprechendsten Frage implementiert. Das Backend bereitet also die zur Verfügung stehenden Daten auf, berechnet Resultate oder speichert neue Eingaben, welche es vom Admin Tool erhält, in der Postgres Datenbank.

Abbildung 3: Grundstruktur der Systemkomponente

Es wurde eine Bereitstellung des PocketDoc Systems im Cloud Service Heroku [11] gewünscht, deshalb ergab sich das in Abbildung 4 dargestellte Deployment-Diagramm. Die Kommunikation zwischen Datenbank und Backend erfolgt mittels JDBC [29], zwischen Backend und Admin Tool mittels HTTP.

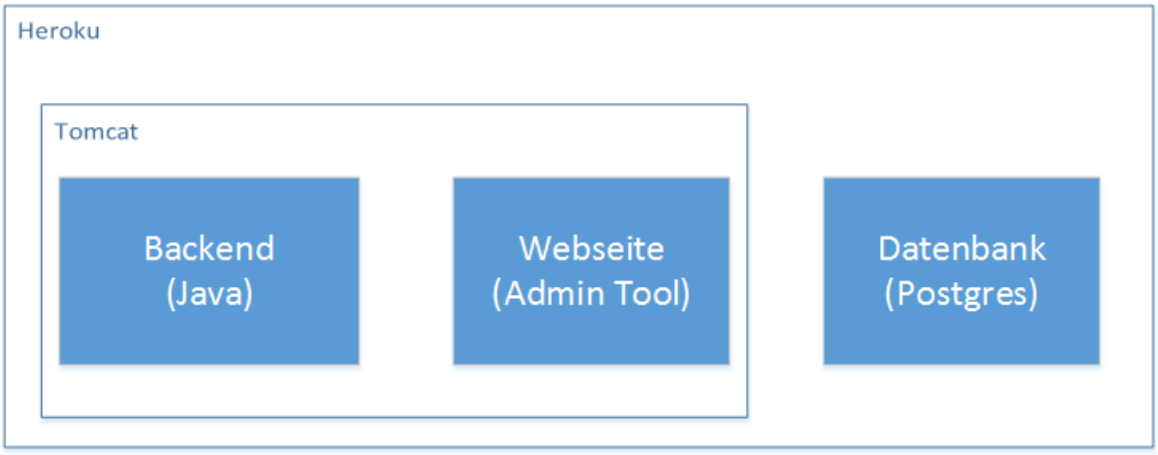


Abbildung 4: Grundstruktur der Systemkomponente

3.1.2 Backend

Wie bereits erwähnt wird das Backend in Java programmiert und soll eine durchgängige Struktur aufweisen, welche anhand der nächsten Grafik einfach verständlich ist.

Abbildung 5: Architektur: Backend

- **Models:** Dienen zur Instanzierung der Datenobjekte. In ihnen sind die Verknüpfungen von Datenbank Feldern zu Klassen Attributen definiert, welche von Hibernate [28] verwendet werden.
- **Mappers:** Greifen über Hibernate auf die Datenbank zu. Und untersützen die CRUD Operationen.
- **Managers:** Sind die zentrale Schnittstelle für alles was mit mit einem Model zu tun hat. Dazu gehören das Berechnen von Informationen oder das Speichern bzw. Laden von Daten in/aus der Datenbank.
- **Calculators:** Die Manager Klassen beinhalten keine grössere Logik deshalb werden Berechnungen über separate Calculator Klassen abgehandelt.
- **Servlets:** Sind die RESTful Schnittstellen es werden POST,PUT,GET und DELETE unterstützt. Servlets dienen zur Aufbereitung von Daten und greifen dafür auf die Manager Klassen zu.

3.1.3 Admin Tool

Das Admin Tool stellt eine Single Page Application dar, dies bietet dem Benutzer einen flüssigeren Ablauf und dadurch eine bessere Erfahrung beim Benutzen der Seite. Mit Hilfe vom Javascript Framework AngularJS lies sich dieses Verhalt gut umsetzten.

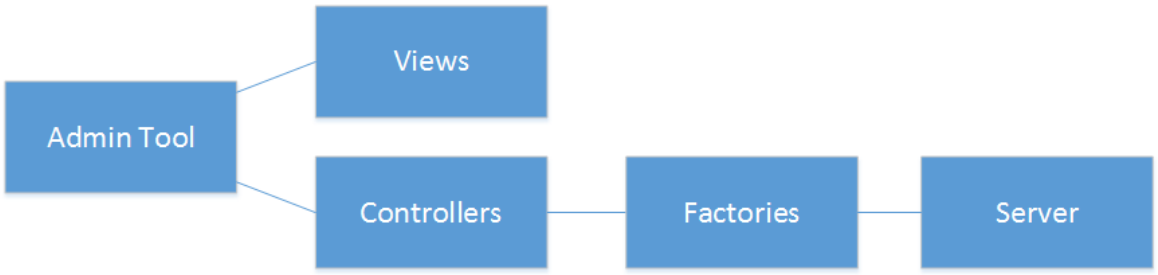


Abbildung 6: Architektur: Admin Tool

- **Views:** Das Admin Tool ist eine Single Page Application, dynamisch werden Teilelemente, Views, in die Seite geladen. Die Views definieren wie die Daten im Admin Tool dargestellt werden. Die Daten erhalten sie von den Controllern.
- **Controllers:** Die Controller beinhalten die ganze Funktionalität des Admin Tools. Sie bereiten die Daten für die Views auf und tauschen mittels den Factories Daten zwischen dem Backend und der Webseite aus.
- **Factories:** Die Factories stellen die Verbinndung zwischen Backend und Admin Tool dar. Sie greifen also auf die REST Schnittstellen des Backends zu. Sie werden auschliesslich von den Controllern genutzt.

3.1.3.1 Admin Tool: Ordnerstruktur

Die Abbildung 7 zeigt die Ordnerstruktur im Admin Tool, während die Tabelle 7 deren Inhalte beschreibt.

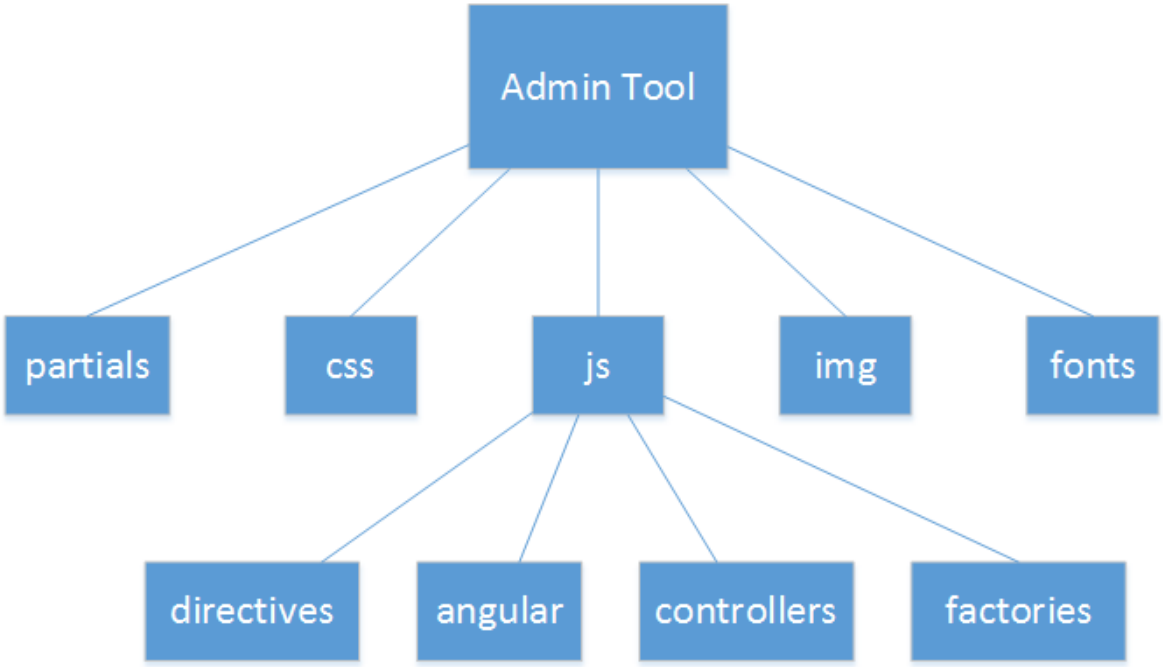


Abbildung 7: Ordnerstruktur: Admin Tool

partials	Beinhaltet die Views, welche von AngularJS dynamisch ins Admin Tool geladen werden
css	Alle benötigten CSS Dateien sind in diesem Ordner enthalten
js	Beinhaltet alle Javascript Dateien. Dazu gehören controllers, factories, directives, Bootstrap [26] und natürlich alle Dateien des AngularJS Frameworks.
img	Loader GIF-Datei
fonts	Für Bootstrap benötigte fonts

Tabelle 7: Ordnerstruktur: Admin Tool

3.2 Kommunikationsdiagramm

Die Kommuniktionsdiagramme zeigen auf welche Komponenten bei verschiedenen Vorgänge im System gebraucht werden und welche Schnittstellen dafür angesprochen werden.

Da einige Kommunikationsdiagramme gleich aufgebaut sind werden hier nur die wichtigsten beschrieben. Der Vollständigkeitshalber sind alle Kommunikationsdiagramme im Anhang zu finden.

3.2.1 Diagnose Verwalten (CRUDs)

Beim der Verwaltung von Diagnosen können diese hinzugefügt, gelesen, verändert und gelöscht werden, auf Englisch create, read, update, delete (CRUD). Das Lesen und Erstellen der Diagnose wird im folgenden Kommunikationsdiagramm beschrieben. Das Verändern und Löschen einer Diagnose funktionieren gleich wie das Lesen einer Diagnose. Die Verwaltung von anderen Entitäten wie Fragen oder Handlungsempfehlungen funktioniert genau gleich.

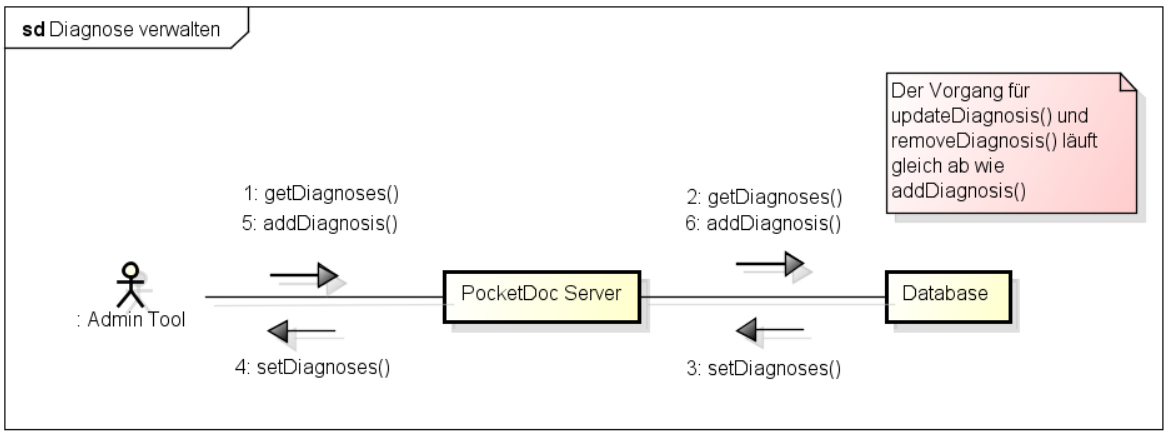


Abbildung 8: Kommunikationsdiagramm: Diagnose Verwalten

1. **Read:** Vom Admin Tool wird dem Backend ein GET Request geschickt.
2. Das PocketDoc Backend führt ein Select durch um die Daten in der Datenbank zu holen.
3. Die Datenbank antwortet dem Backend mit den gewünschten Daten
4. Das Backend antwortet dem Admin Tool mit den gewünschten Daten
5. **Create:** Vom Admin Tool wird dem Backend ein POST Request geschickt.
6. Das PocketDoc Backend führt ein Insert durch um die Daten in die Datenbank zu schreiben.

3.2.2 Regeln Überprüfen

Die Regelüberprüfung ist der Vorgang bei dem anhand von typischen Antworten überprüft wird ob dabei die richtige Diagnose herausspringt.

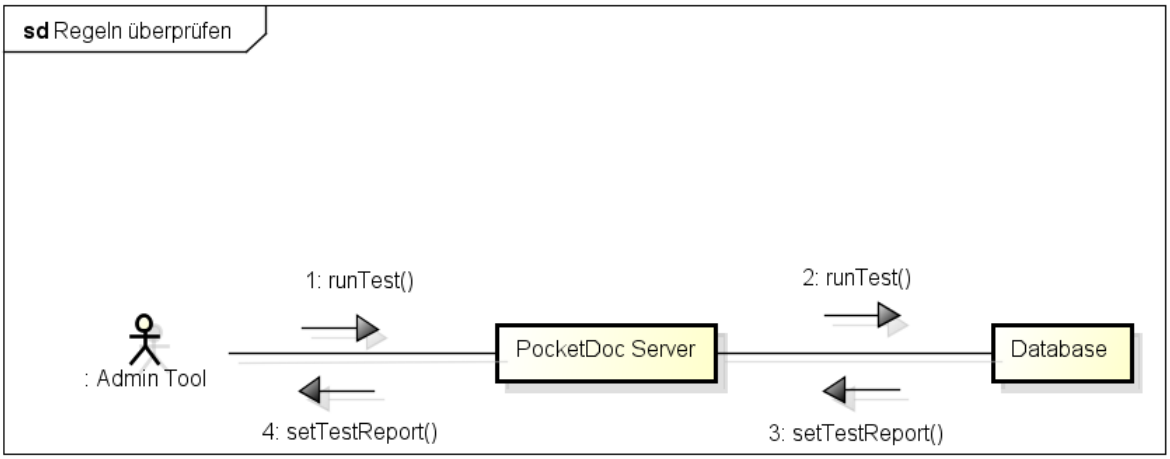


Abbildung 9: Kommunikationsdiagramm: Diagnose Testlauf

1. **Read:** Vom Admin Tool wird dem Backend ein GET Request geschickt um die Resultate des Testdurchlaufes zu holen.
2. Das PocketDoc Backend führt ein Select auf die gewünschte Diagnose aus, um die Testdaten zu holen.
3. Die Datenbank antwortet dem Backend mit den gewünschten Daten
4. Das Backend führt den Test durch und antwortet dem Admin Tool mit den resultierenden Daten

3.2.3 Testlauf

Der Testlauf ist einer der wichtigsten Vorgänge des Systems. Auf die im Testlauf verwendeten Schnittstellen soll in Zukunft das PocketDoc App zugreifen können.

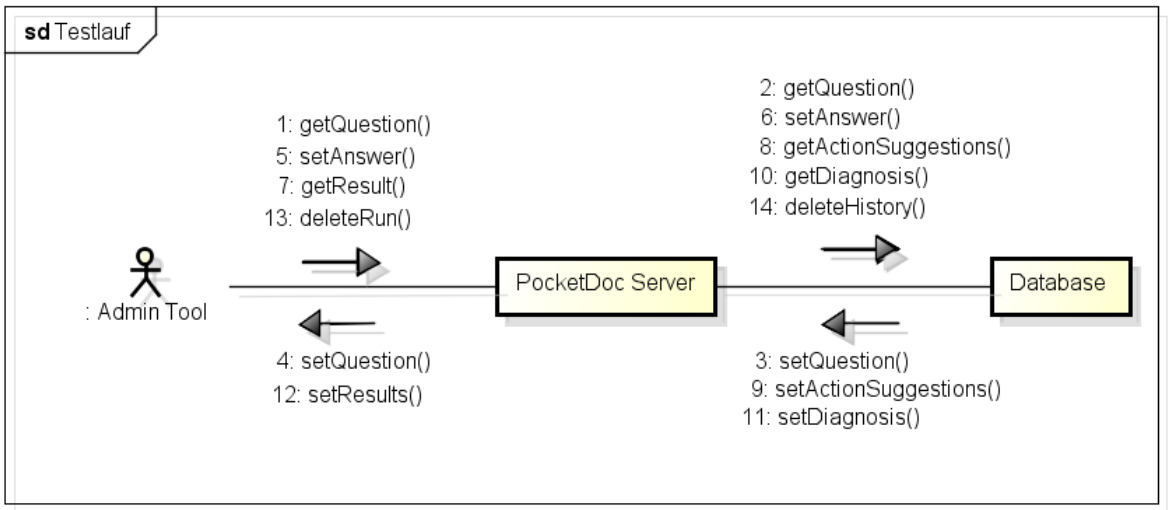


Abbildung 10: Kommunikationsdiagramm: Diagnose Testlauf

1. **Question:** Vom Admin Tool wird dem Backend ein GET Request geschickt, um die vielversprechendste nächste Frage zu holen.
2. Das PocketDoc Backend führt ein Select durch, um alle Fragen in der Datenbank zu holen.
3. Die Datenbank antwortet dem Admin Tool mit den gewünschten Daten.
4. Das Backend berechnet die vielversprechendste nächste Frage und antwortet dem Backend mit den resultierenden Daten.
5. **Antwort setzen:** Vom Admin Tool wird dem Backend ein PUT Request geschickt, um eine abgegebene Antwort zu speichern.
6. Das PocketDoc Backend führt ein Insert durch um die Antwort in der Datenbank zu schreiben.
7. **Result:** Vom Admin Tool wird dem Backend ein GET Request geschickt um die Resultate des Testdurchlaufs zu holen.
8. Das PocketDoc Backend führt ein Select durch, um alle Handlungsempfehlungen in der Datenbank zu holen.
9. Die Datenbank antwortet dem Admin Tool mit den gewünschten Daten.
10. Das PocketDoc Backend führt ein Select durch, um alle Diagnosen in der Datenbank zu holen.
11. Die Datenbank antwortet dem Admin Tool mit den gewünschten Daten.
12. Das Backend berechnet das Resultat des Testdurchlaufs und antwortet dem Admin Tool mit den resultierenden Daten.
13. **Finish:** Vom Admin Tool wird dem Backend ein DELETE Request geschickt um den Verlauf des Testdurchlaufs zu löschen.
14. Das PocketDoc Backend führt ein Delete durch um den Verlauf in der Datenbank zu löschen.

3.3 ERM

Abbildung 11 zeigt die Datenbankstruktur, welche aus dem vorgegebenen Excel File extrahiert wurde.

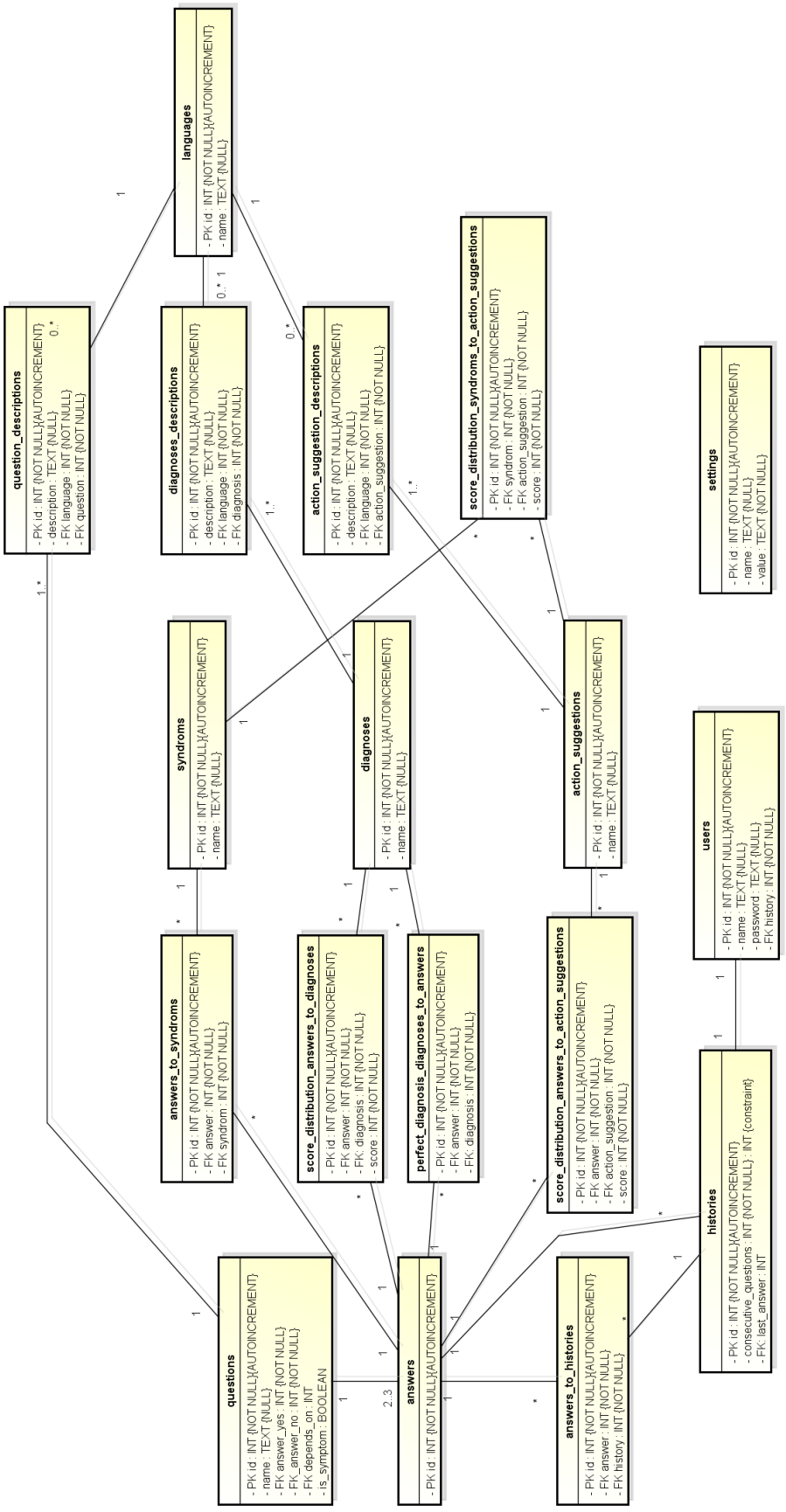


Abbildung 11: Question Beispiel aus Klassendiagramm

- **settings:** In dieser Tabelle werden Einstellungen nach dem Prinzip „Name-Value“ gespeichert.
- **users:** Die User die später dazukommen könnten werden in dieser Tabelle gespeichert. Im Moment wird nur der Administrator(admin) gespeichert. Ein User hat zudem eine History, die Referenz darauf wird ebenfalls in dieser Tabelle abgelegt. Wenn das System mit einem Vergleichsystem ausgebaut werden soll können hier zwei Histories pro Benutzer gespeichert werden.
- **histories:** Die Historie Tabelle enthält Informationen zum Fragedurchlauf wie zum Beispiel die letzte Gestellte Frage. Dazu ist in dieser Tabelle ein Counter enthalten der Zählt wie oft eine bestimmte Diagnose mit einem eigestellten Abstand an erster Stelle des Rankings ist.
- **answers_to_histories:** Diese Tabelle enthält die aktuellen Antworten eines Users. Aus den Einträgen in dieser Tabelle werden später Diagnose und Handlungsempfehlungen berechnet. Diese Tabelle referenziert auf Antworten und auf Histories.
- **questions:** Die an den User gestellten Fragen werden in dieser Tabelle abgespeichert.
- **answers:** Pro Frage gibt es zwei Antworten, diese sind in dieser Tabelle enthalten.
- **syndroms:** Diese Tabelle speichert die Syndrome.
- **answers_to_syndroms** Hier wird abgespeichert welches Syndrom aus welchen Symptomen und Informationen besteht.
- **diagnoses:** Die verschiedenen Diagnosen werden in dieser Tabelle abgespeichert.
- **perfect_diagnosis_diagnoses_to_answers:** Hier wird vermerkt, welches die typischen Antworten sind womit man eine Diagnose erreichen sollte.
- **action_suggestions** In dieser Tabelle werden die Handlungsempfehlungen gesichert.
- **languages:** Da das System mehrsprachig laufen soll sind in dieser Tabelle die verschiedenen Sprachen enthalten.
- **Zwischentabellen: score_distribution:** Diese Zwischentabellen zeigen die Punkteverteilung zwischen Syndrome, Symptome, Diagnosen und Handlungsempfehlungen
- **Zwischentabellen: descriptions:** Diese Zwischentabellen enthalten alle verschiedenen mehrsprachigen Beschreibungen.

3.4 Klassendiagramm

Die Abbildung 12 zeigt nur einen Auschnitt aus unserem Klassendiagramm welches im Anhang in der Abbildung 34 vollständigkeitshalber zu finden ist. Dieses Beispiel zeigt die Struktur unserers Backends, welche sich wiederholt für Diagnosen, Syndrome, usw.

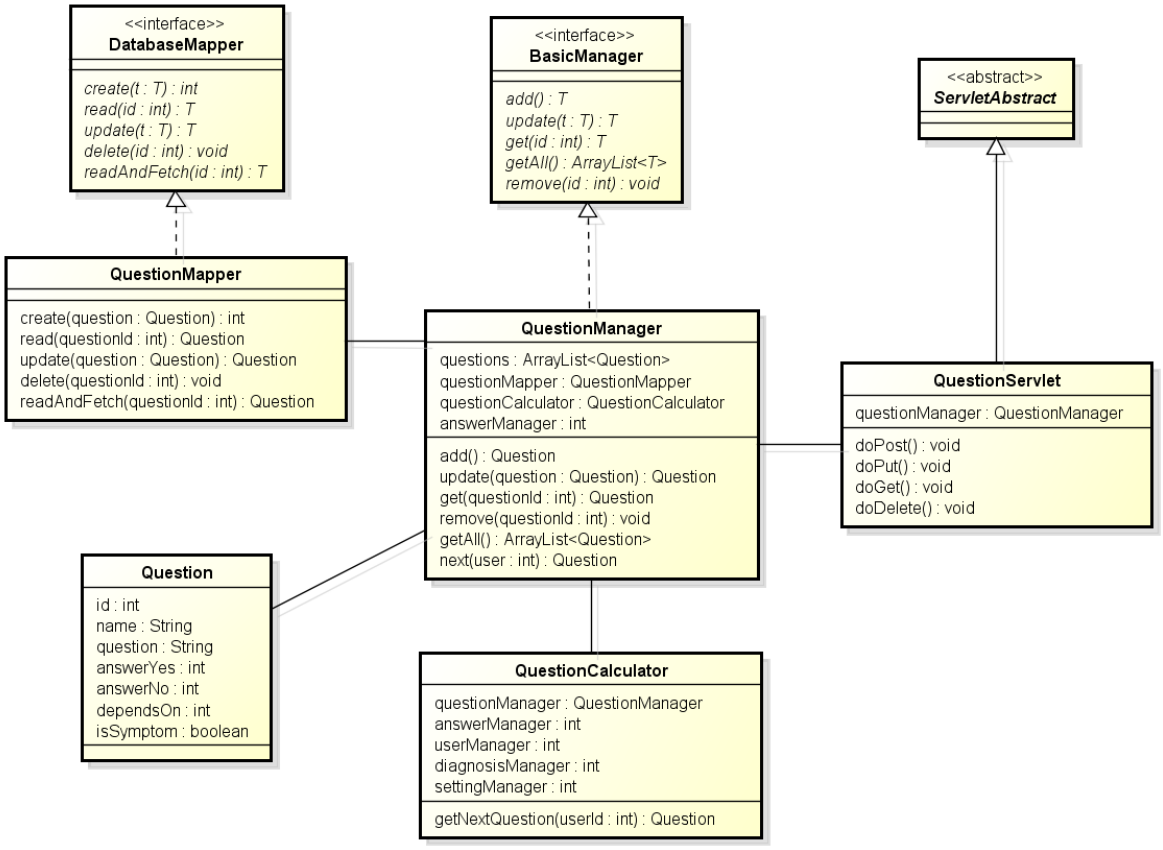


Abbildung 12: Question Beispiel aus Klassendiagramm

- **Question:** In diesem Beipsiel ist Question das Datenmodel es bestimmt den Aufbau einer Frage und in diesem werden die Verknüpfungen von Datenbankfeldern zu Klassenattributen definiert.
- **DatabaseMapper Interface:** Diese Interface bestimmt die Methoden, welche von den Mapper Klassen implementiert werden müssen.
- **QuestionMapper** Greift über Hibernate auf die Datenbank zu. Und untersützt die CRUD Operationen, welche durch das DatabaseMapper Interface vorgegeben sind.
- **BasicManager Interface:** Dieses Interface bestimmt die Methoden, welche von den Manager Klassen implementiert werden müssen.
- **QuestionManager:** Ist die zentrale Schnittstelle für alles was mit mit Question zu tun hat. Sei es das Persistieren in der Datenbank oder das Bestimmen der nächsten vielversprechendsten Frage.
- **QuestionCalculator:** Die Manager Klassen beinhalten keine grössere Logik deshalb werden Berechnungen über separate Calculator Klassen abgehandelt. Der QuestionCalculator zum Beispiel berechnet die nächste vielversprechendste Frage.
- **ServletAbstract:** Wird in den Servlets verwendet um mit Request und Responses umzugehen. Hier werden Requests in Json umgewandelt, Strings als Response geschickt und IDs aus den URLs geparst.
- **QuestionServlet:** Das QuestionServlet ist die RESTful Schnittstelle es werden POST,PUT,GET und DELETE unterstützt. Servlets dienen zur Aufbereitung von Daten und greifen dafür auf die Manager Klassen zu.

3.5 Sequenzdiagramm

3.5.1 CRUDs

Der Umfang bei den Sequenzdiagrammen von CRUDs wurde auf das Hinzufügen einer neuen Diagnose beschränkt. Da alle CRUDs nach dem folgenden Prinzip funktionieren:

Das Admin Tool ruft die Schnittstelle nach REST auf. Die Schnittstelle ruft den Manager auf, der die Anfrage bearbeitet und leitet diese weiter an den Mapper.

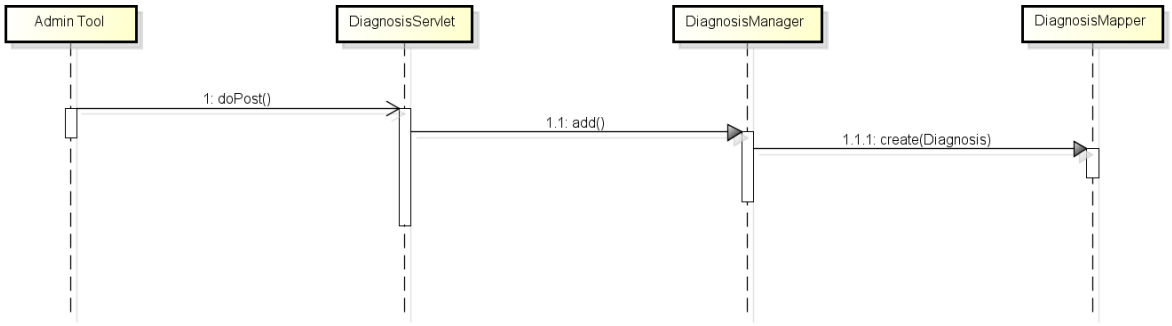


Abbildung 13: Sequenzdiagramm: Erstellung einer Diagnose

In diesem Fall wird die POST Schnittstelle aufgerufen, die Servlet Klasse ruft die add()-Methode auf und diese die create()-Methode des DiagnosisMapper. Die einzige Ausnahme ist das Lesen eines Objektes. Dort bekommt das Admin Tool noch eine Antwort vom Mapper.

3.5.2 Testlauf: Nächste Frage abfragen

Die nächste Frage wird abgefragt wenn ein Testlauf durchgeführt wird. Diese wird im Backend geholt damit sie dem Kunden gestellt werden kann.

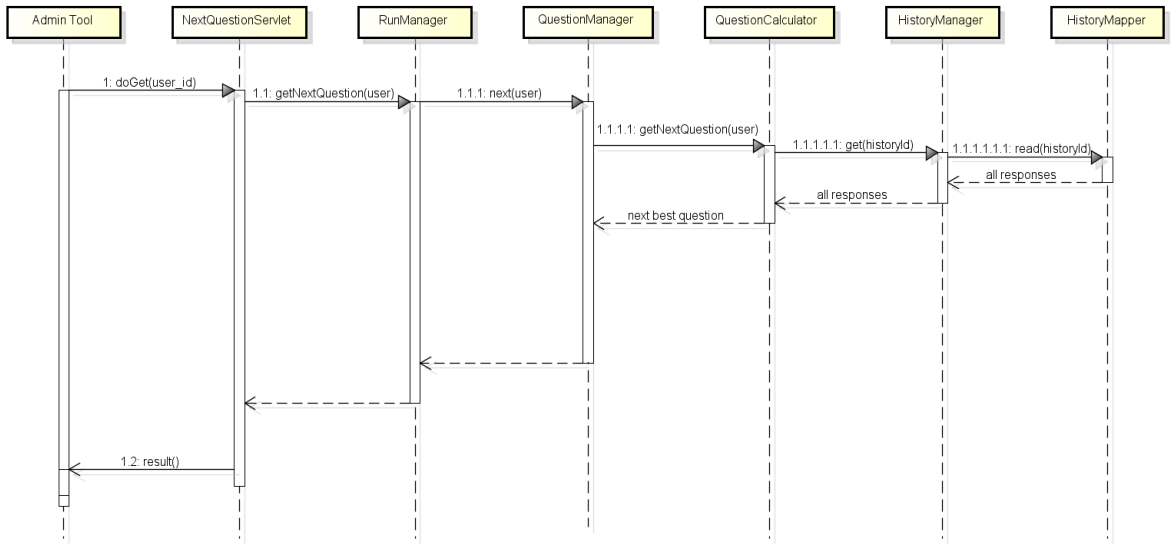


Abbildung 14: Sequenzdiagramm: Nächste Frage abfragen

Durch ein GET auf das NextQuestionServlet wird das System informiert, dass die nächste Frage abgerufen werden soll. Die Nachricht wird wie üblich an den RunManager weitergeleitet. Dieser ruft den QuestionManager auf, der dem QuestionCalculator den Auftrag gibt die nächste Frage zu berechnen. Der QuestionCalculator holt zuerst alle abgegebenen Antworten beim beim HistoryManager, somit

kann er bestimmen, welche Fragen bereits gestellt wurden. Nun kann er die nächste Frage ausrechnen und diese als Resultat zurückschicken.

3.5.3 Testlauf: Antwort abgeben

Während dem Testlauf hat der Benutzer die Möglichkeit auf die zuvor abgeholte Frage eine Antwort abzugeben. Diese Antwort wird ebenfalls dem Backend geschickt.

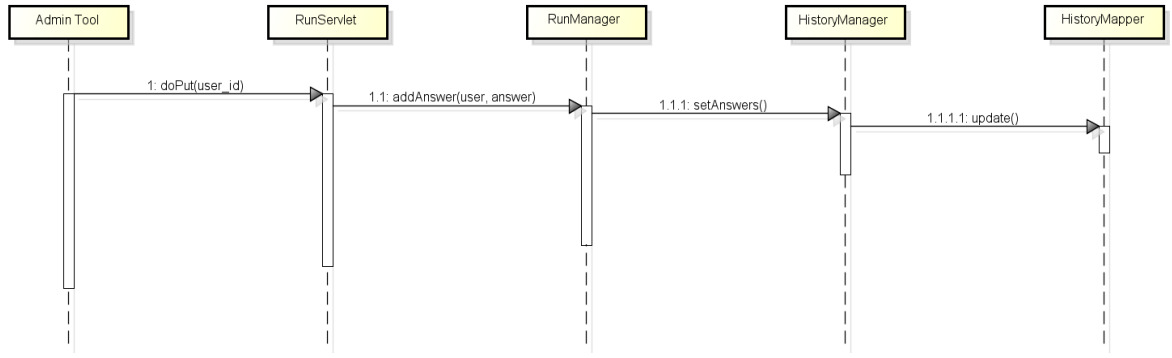


Abbildung 15: Sequenzdiagramm: Antwort abgeben

Das Setzen einer Antwort in den History geschieht durch einen PUT im RunServlet. Wieder wird hier die Nachricht an den Manager weiter geschickt. Dafür Zuständig ist der RunManager. Schliesslich wird die Antwort via HistoryManager und HistoryMapper in der Datenbank gespeichert.

3.5.4 Testlauf: Resultate abfragen

Sind keine Fragen mehr vorhanden, wurde eine Diagnose gefunden oder will der Benutzer den Fragedurchlauf beenden, so müssen die Resultate des Testlaufs abgeholt werden.

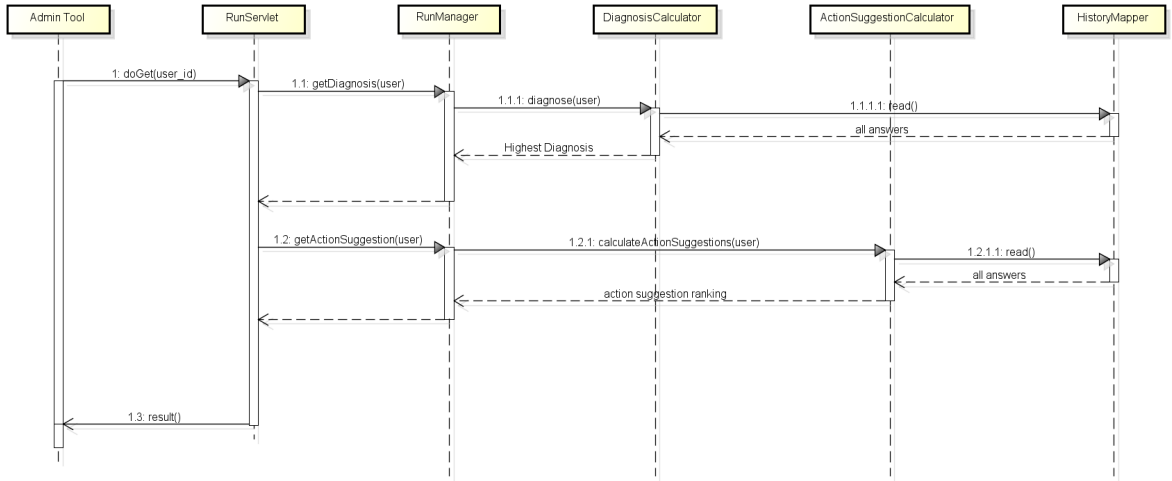


Abbildung 16: Sequenzdiagramm: Resultate abfragen

Um die Resultate abzuholen wird ein GET Request an das RunServlet gesendet.

Zuerst wird im RunManager getDiagnosis() aufgerufen. Der Auftrag wird dort via DiagnosisManager am DiagnosisCalculater weitergeleitet, der die nötigen Daten im HistoryManager holt und aus diesen die Diagnose berechnet.

Danach ruft der RunManager die getActionSuggestion-Methode im RunManager auf. Via ActionSuggestionManager wird die calculateActionSuggestion-Methode im Calculator aufgerufen. Wieder werden zuerst die vom Calculator benötigten Daten im HistoryManager abgeholt. Danach berechnet der Calculator die Handlungsempfehlungen und gibt sie zurück. Diese Resultate werden im Servlet zusammen dem Benutzer zurückgegeben.

3.5.5 Perfekte Diagnose testen

Beim Bearbeiten von Diagnosen hat der Kunde die Möglichkeit zu testen ob sich seine Diagnose mit eingestellten Standartantworten immernoch an höchster Stelle befinden würden.

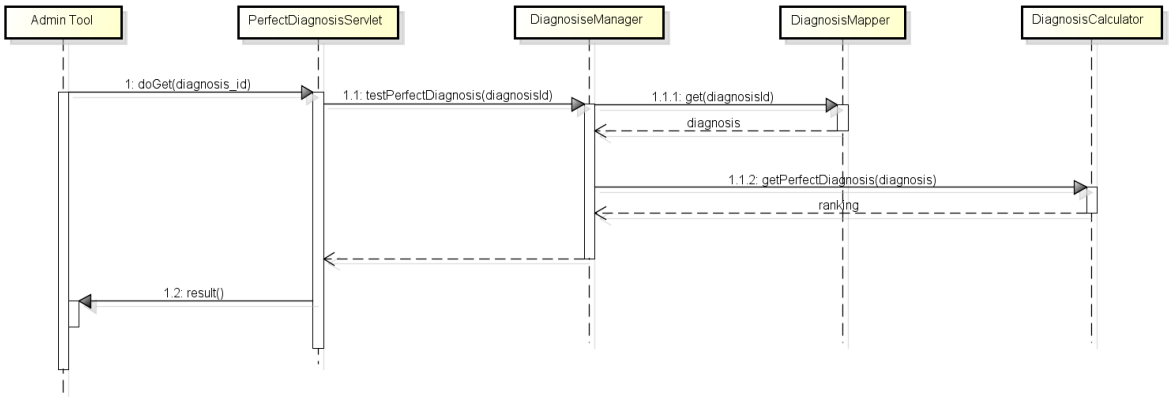


Abbildung 17: Sequenzdiagramm: Perfekte Diagnose testen

Die Überprüfung der perfekten Diagnose geht durch das PerfectDiagnosisServlet. Die Nachricht wird dann dem DiagnosisManager geschickt, der mithilfe des DiagnosisCalculator ein Test durchführt. Als Antwort wird die Rangliste, die dazugehörigen Total Scores und ein Boolean geschickt.

3.6 Wireframes

Das ganze Design des Admintools ist flach geplant. Das heisst innerhalb von wenig Klicks kommt man überall hin. Die Navigation wird dabei am oberen Rand dargestellt. Dort kann man zwischen den verschiedenen Bearbeitungsinstanzen wechseln und auch Tests starten. In folgenden Kapiteln werden die einzelnen Seiten beschrieben.

3.6.1 Diagnosen

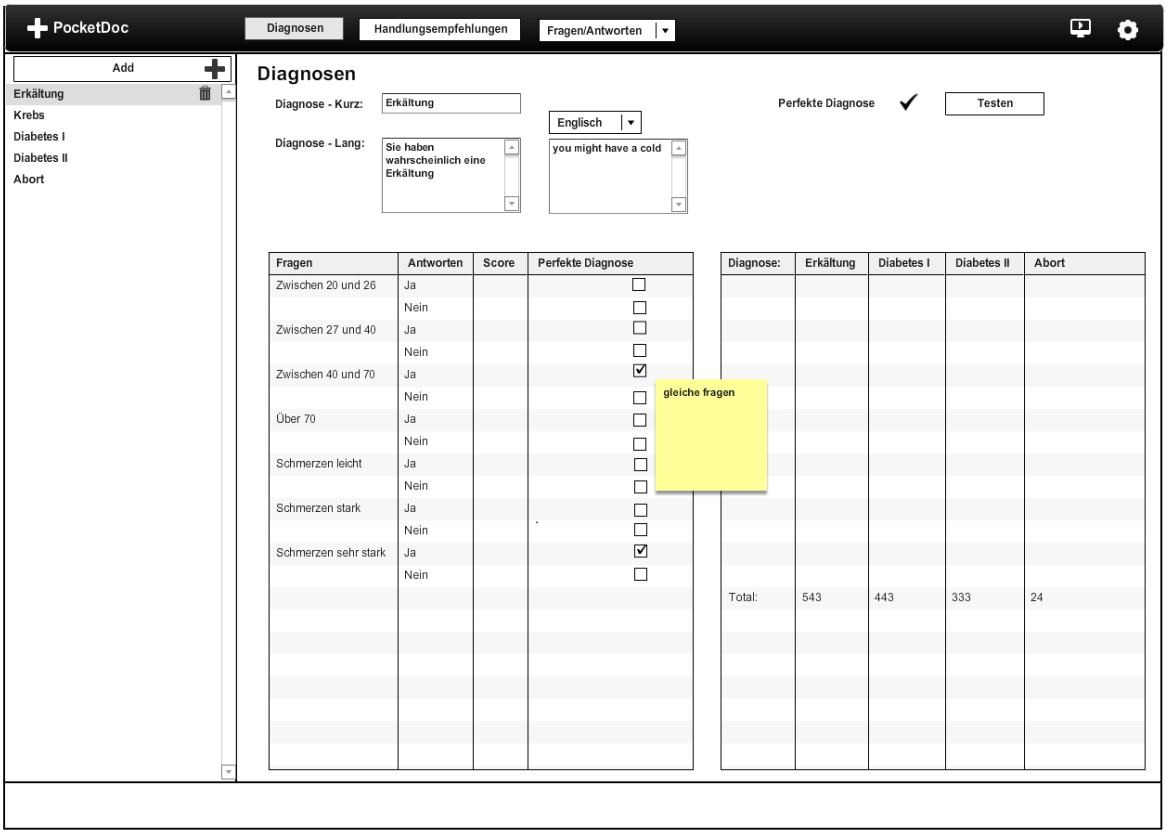


Abbildung 18: Wireframe: Diagnosen

Auf der Diagnoseseite sieht man zuerst links eine Übersicht mit allen Diagnosen. Oben befindet sich der „Add“-Knopf. Klickt man darauf wird eine neue Diagnose hinzugefügt. Unterhalb kann man auf jede einzelne Diagnose klicken dann wird diese direkt geladen und angezeigt. Dieser Bereich wiederholt sich auch auf anderen Seiten des Admin Tools.

In der Mitte befindet sich der Bereich wo die Diagnosen bearbeitet werden. Zuerst kann die Kurzbeschreibung, dann die Langbeschreibung auf Deutsch und in anderen Sprachen bearbeitet werden. Auf der rechten Seiten kann getestet werden ob die Diagnose mit der perfekten Zusammensetzung von Symptomen erreicht werden kann. Unten links ist eine Tabelle wo die Scoreverteilung und die perfekte Diagnose eingestellt werden können. Die rechte Tabelle zeigt das detaillierte Resultat des Tests. Diese zwei Tabellen sind so plazziert, dass die Reihen auf die gleichen Fragen/Antworten verweisen.

3.6.2 Fragen

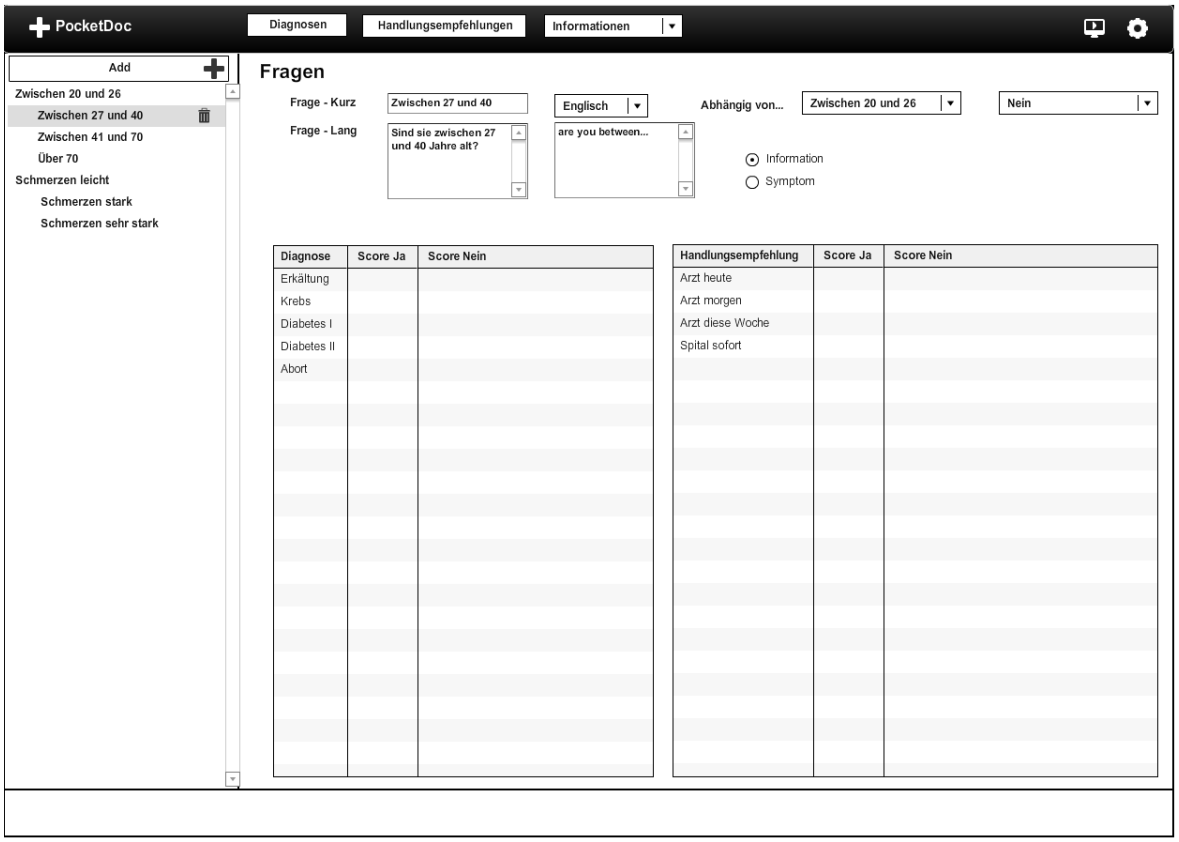


Abbildung 19: Wireframe: Fragen

Die Fragenseite sieht der Diagnoseseite sehr ähnlich. Auf der linken Seite ist wieder die Übersicht der Fragen angezeigt

Auch die rechte Seite sieht fast wie die Diagnoseseite aus. Wieder sind Kurzbeschreibung und Langbeschreibung in mehreren Sprachen zu sehen. Die Fragenseite hat nun aber keine Tests, dafür zwei Scoreverteilungstabellen. Die eine Tabellen ist für die Verteilung zu Diagnosen gedacht, die andere für die Handlungsempfehlungen. Oben rechts noch einstellbar ist die Abhängigkeit. Damit kann eine Frage von einer anderen Frage abhängig sein. Dies ist wichtig beim Herausfinden einer nächsten Frage. Darunter kann noch eingestellt werden ob es sich bei der Frage um eine Information oder ein Symptom handelt.

3.6.3 Handlungsempfehlungen

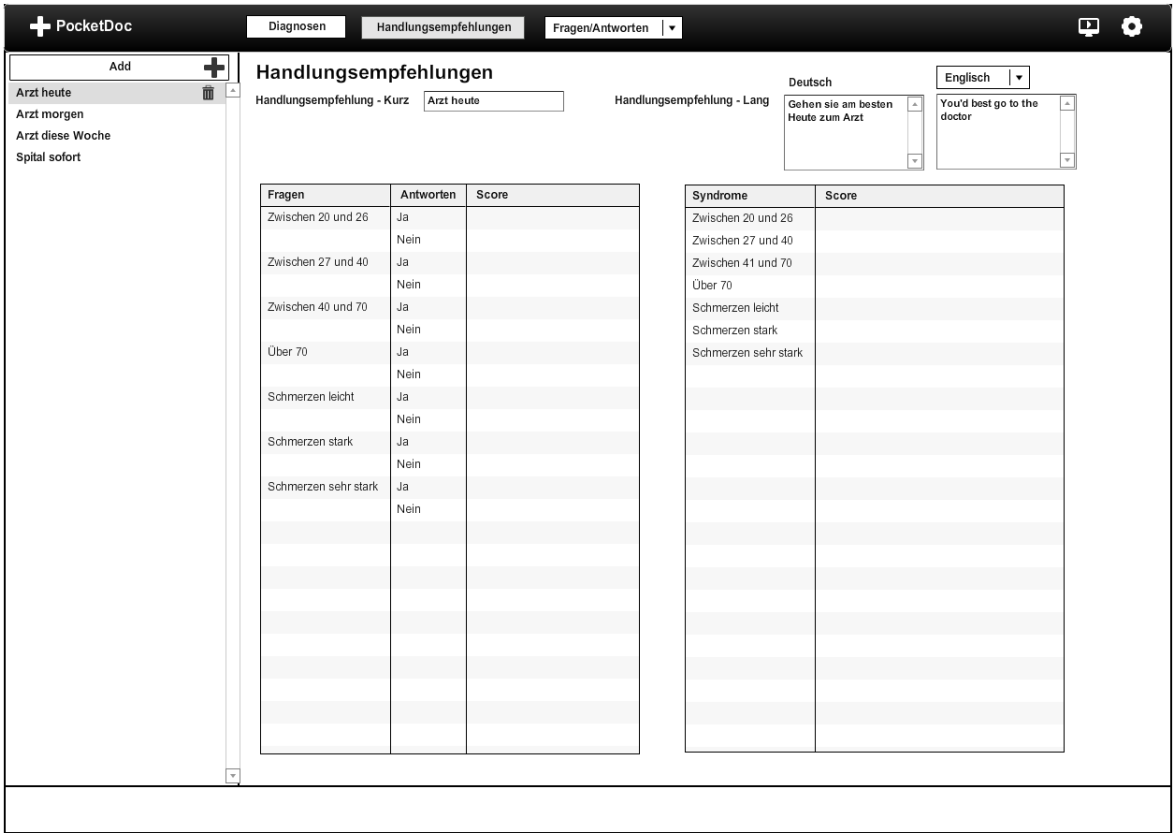


Abbildung 20: Wireframe: Handlungsempfehlungen

Auch die Handlungsempfehlungsseite ist ähnlich wie die letzten zwei Seiten aufgebaut. Die linke Seite kümmert sich um eine Übersicht der Handlungsempfehlungen, wieder mit einem Button, um eine neue Handlungsempfehlung hinzuzufügen.

Der rechte Teil hat weniger Ihnalt als die Diagnose- und Fragenseiten. Oben sind Kurz- und Langbeschreibungen einstellbar. Unten sind zwei Scoreverteilungstabellen, einmal für Fragen und einmal für Syndrome.

3.6.4 Syndrome

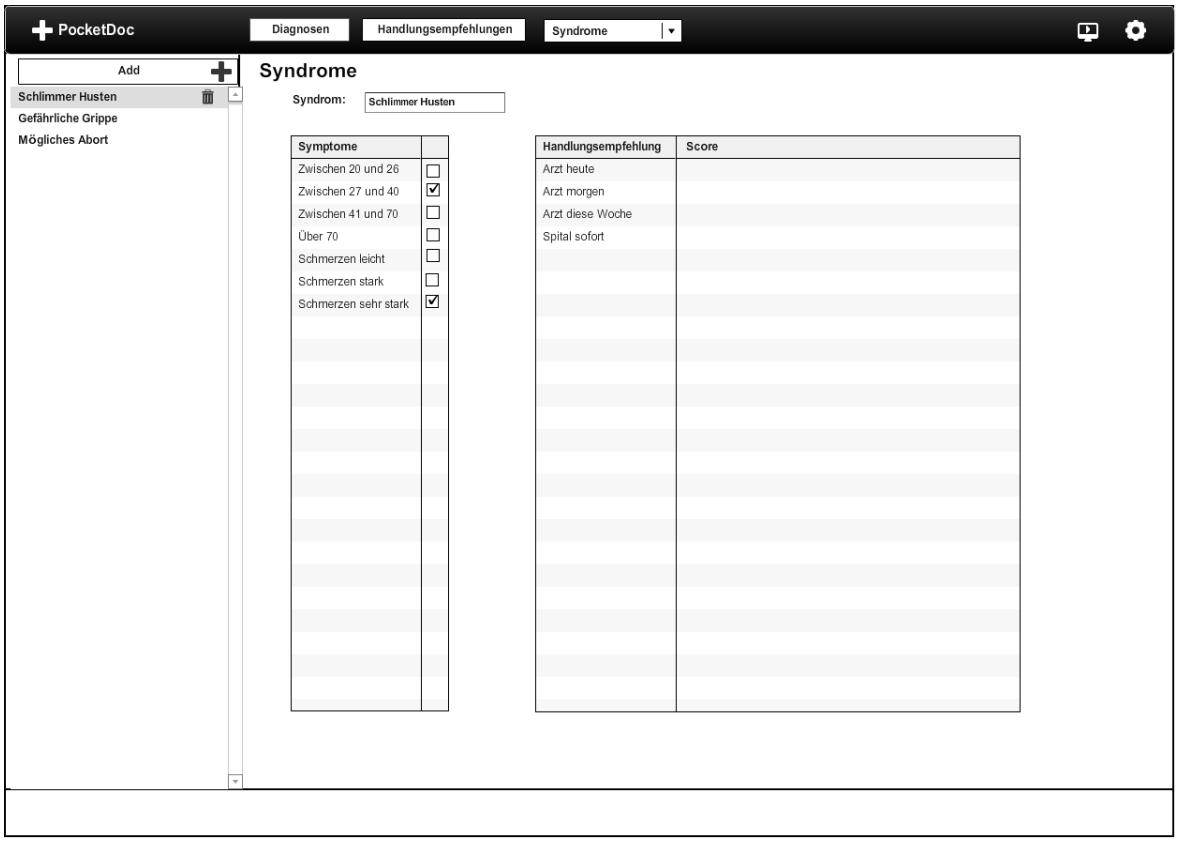


Abbildung 21: Wireframe: Syndrome

Die Syndromseite hat noch weniger Inhalt als die letzten paar Seiten. Auf der link Seite befindet sich die übliche Navigation.

Auf der rechten Seite findet man hier speziell nur eine Kurzbeschreibung. Darunter auf der rechten Seite eine Scoreverteilungstabelle. Links daneben kann man hier einstellen aus welchen Symptome sich das Syndrom zusammensetzt.

3.6.5 Testlauf

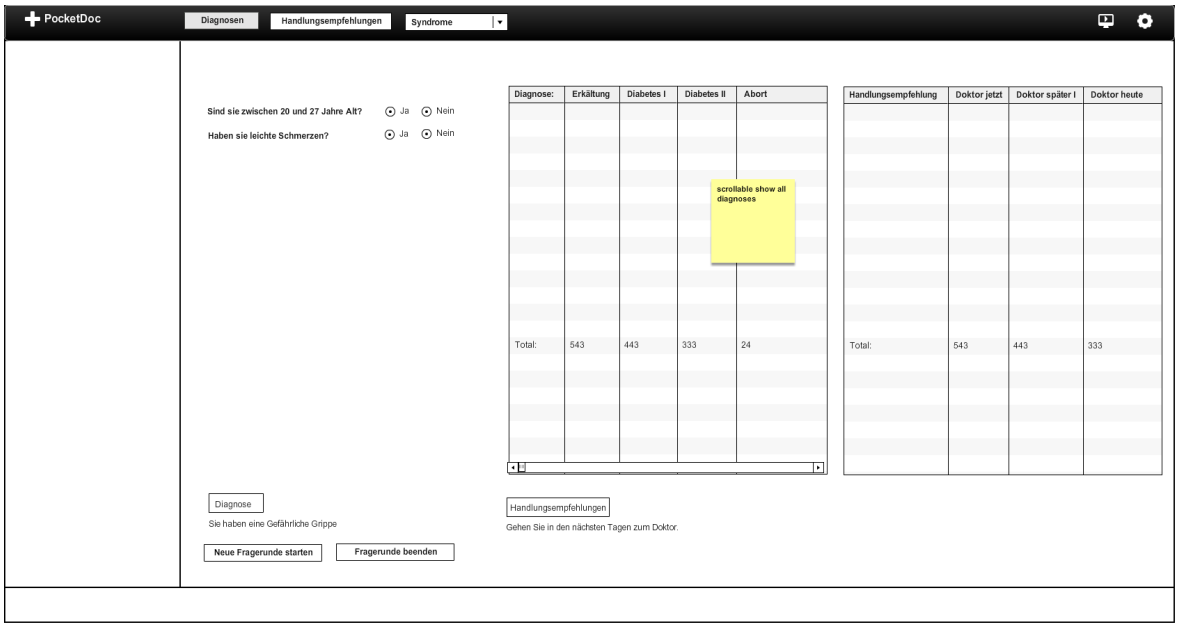


Abbildung 22: Wireframe: Testlauf

Die Testlauf Seite bietet eine optimale Nachvollziehung wie das System im Hintergrund rechnet und welche Änderungen welchen Einfluss auf die Diagnose und die Handlungsempfehlungen haben. Auf der Linken Seite werden die Fragen nacheinander gestellt, dabei gehen wird auch gleich die Funktionalität der nächsten vielversprechendsten Frage getestet. In der mittleren Tabelle sehen wir was die Antwort für einen Einfluss auf die Diagnosen hat und auf der rechten Seite den Einfluss der Antwort auf die Handlungsempfehlungen

Der Button Neue Fragerunde starten ermöglichen es erneut zu beginnen. Der Button Fragerunde beendenßorgt für eine frühzeitige Beendung der Fragerunde. Nach der Beendung werden unten links die Diagnosen angezeigt und unten rechts die Handlungsempfehlungen. Man hat die Möglichkeit die Fragerunde wieder aufzunehmen oder eine neue Runde zu starten.

3.6.6 Einstellungen

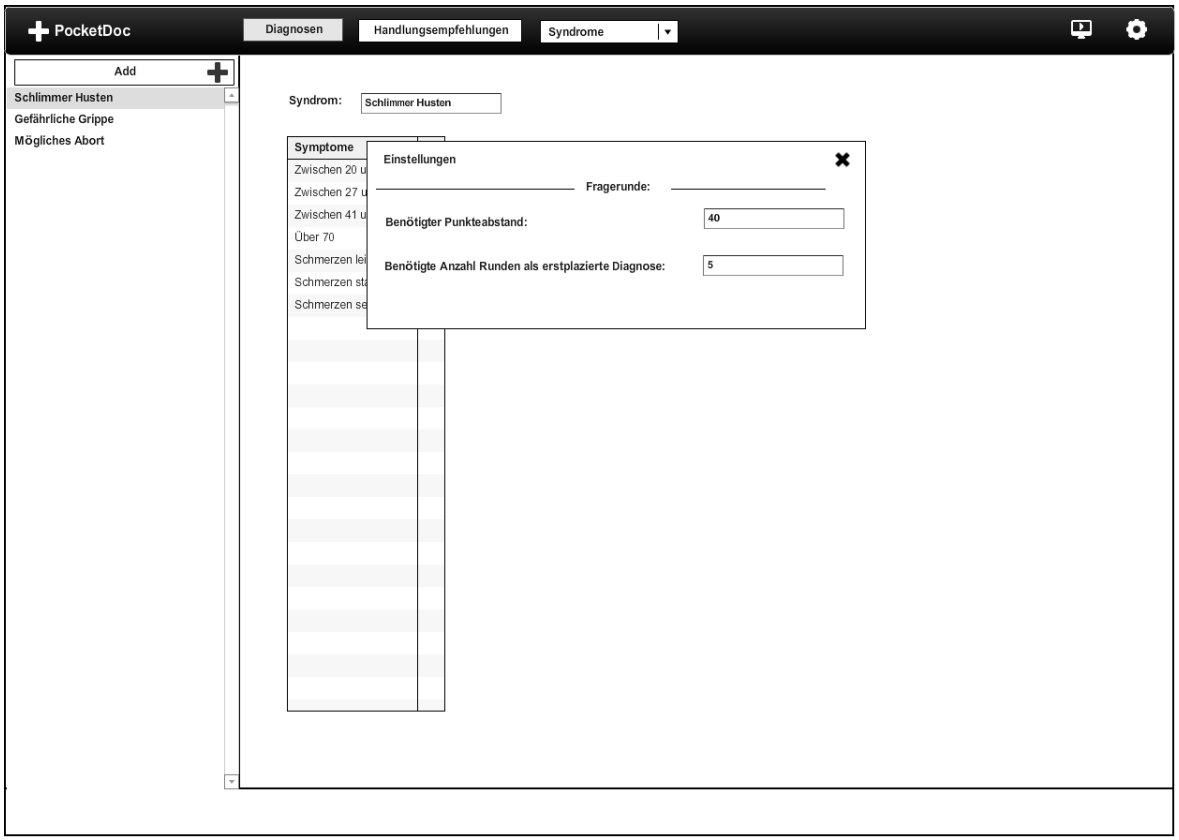


Abbildung 23: Wireframe: Einstellungen

Das Einstellungsfenster öffnet man mit dem Zahnrad auf der rechten Seite der Navigation. Das Fenster öffnet sich dann über die Seite die gerade offen ist. Darauf können verschiedene Einstellungen bearbeitet werden.

4 Umsetzung

Während der Umsetzung wurden verschiedene Arten von Klassen geschrieben. Es wurden Calculators Manager, Mappers, Servlet, Serializer und Tests geschrieben. In den nächsten Kapiteln werden die wichtigsten Klassen genauer beschrieben. Mehr Informationen zu den einzelnen Methoden in den Klassen wurden mit Javadoc [30] im Code dokumentiert.

4.1 Backend

4.1.1 Calculators

Die Calculators beinhalten alle Rechenaufgaben, die im System notwendig sind. Die Berechnung der Ranglisten sowie die Berechnung der nächsten Frage.

4.1.1.1 ActionSuggestionCalculator

Im ActionSuggestionCalculator wird die Rangliste der Handlungsempfehlungen anhand der bisher beantworteten Fragen berechnet. Dafür werden zuerst alle Antworten im History Objekt abgeholt. Durch diese Antworten wird dann durchiteriert und jeweils die Scoreverteilung geholt.

Die Scoreverteilung wird durch Hibernate der Answer-Entity automatisch zugewiesen. Sie wird als HashSet gespeichert und kann gelesen werden, wenn zuerst im Mapper die readAndFetch Methode aufgerufen wird. Damit sorgen wird dafür, dass alle one to many Referenzen im Objekt gelesen werden.

Diese Scoreverteilung enthält ein Score und die Handlungsempfehlung die den entsprechenden Score erhalten wird. Der Calculator iteriert dann durch diese Scoreverteilung und verteilt mithilfe einer Map den Score den entsprechenden Handlungsempfehlungen.

Der Handlungsempfehlung Score wird aber noch von Syndromen beinflusst. Dazu werden alle vorhandene Syndrome geholt weil zuerst überprüft werden muss ob diese zutreffen. Es wird also überprüft welche Syndrome mit den bereits abgegebenden Antworten erfüllt werden. Ist dies der Fall wird dort ebenfalls die Scoreverteilung geholt und dem Ranking hinzugefügt.

Am Schluss wird dieses Ranking in eine Treemap kopiert, dem Score nach geordnet und zurückgegeben.

4.1.1.2 DiagnosisCalculator

Der DiagnosisCalculator funktioniert nach dem genau gleichen Prinzip wie der ActionSuggestionCalculator. Dieser beinhaltet aber mehr Methoden. Zum einen gibt es eine Methode die wie bei den Handlungsempfehlungen eine Rangliste berechnet. Dannach gibt es eine Methode, die nur die Top Diagnose berechnet und eine Methode die ein Ranking berechnet und dazu eine zusätzliche Antwort dem Ranking hinzufügt. Diese wird dann im QuestionCalculator gebraucht.

Bei der Berechnung des Ranking wird hier genau gleich vorgegangen wie bei der Berechnung des Handlungsempfehlung-Ranking. Zuerst werden alle Antworten geladen, dann ihre Scoreverteilung und abschliessend wird diese Scoreverteilung gebraucht um ein Ranking aufzubauen. Da Syndrome bei der Berechnung der Diagnosen keine Rolle spielen wird hier auf diesen Schritt verzichtet.

Dieses Ranking wird ebenfalls in eine Treemap kopiert, dem Score nach geordnet und zurückgegeben.

Dieses vorgehen wird eins zu eins für das Berechnen der Top Diagnose gebraucht. Dort wird aber einfach das höchste Element der Treemap zurückgegeben. Diese Methode wird dann für den eigentlichen Fragedurchlauf gebraucht, da dort die ganze Liste nicht wichtig ist sondern nur die oberste Diagnose gebraucht wird.

Die dritte Methode enthält als zusätzliches Parameter eine Zusatzantwort. Bei dieser Methode wird einfach am Schluss die Scoreverteilung noch zusätzlich dem Ranking zugeteilt.

4.1.1.3 QuestionCalculator

Der QuestionCalculator hat eine Aufgabe und darum auch nur eine public Methode. Diese Method ist dafür verantwortlich die nächst Frage zu berechnen.

Das Berechnen der nächste Frage muss grundsätzlich überprüfen, welche Frage die grösste Differenz zwischen erst- und zweitplatzierter Diagnose im Diagnoseranking erzielen würde, jedoch gibt es Fragen

die bevorzugt werden. Als erstes wird überprüft ob eine Frage auf die zuletzt gestellte Frage abhängig ist. Diese Frage würde dann bevorzugt werden. Aus diesem Grund wird dies zuerst überprüft.

Dannach müssen alle Fragen die im System gespeichert sind geholt werden und „gesäubert“ werden. Es werden zuerst alle schon gestellten Fragen von dieser Liste gelöscht, sowie alle Fragen die von diesen Fragen abhängig sind, da diese so oder so nicht mehr gestellt werden müssen.

Aus dieser Liste von Fragen wird mit Hilfe des DiagnosisCalculator für jede mögliche Antwort eine Rangliste erstellt. Dafür wird die Methode gebraucht bei der eine Zusatzantwort angegeben werden kann. Aus diesen Ranglisten wird jeweils der Scoreunterschied der erst- und zweitplatzierten Diagnose angeschaut. Die Frage die dabei den grössten Scoreunterschied bewirkt ist somit die nächste Frage.

Als letzter Berechnungsschritt wird überprüft, ob diese Frage von einer anderen Frage abhängig ist. Diese andere Frage müsste dann zuerst gestellt werden. Dieser Vorgang wird rekursiv durchgeführt um an die letzte Frage der Abhängigkeitskette zu gelangen. Dieser Schritt ist wegen Loops gefährlich. Aus diesem Grund wird beim Speichern von Fragen und Antworten immer zuerst nach Loops gesucht.

4.1.2 Mapper

Die Mapper sind die letzte Klassen bevor die Daten in der Datenbank gespeichert werden. Diese Klasse enthält verschiedene Lese-, Schreib-, Lösch- und Updatemethoden die für diesen Zweck die Hibernate Library verwenden. Für jede Tabelle in der Datenbank existiert eine Model-Klasse, ausser für Zwischentabellen die keine Zusatzattribute haben.

4.1.2.1 Fetching Strategie

Bei der Umsetzung dieser Klassen trat das Problem auf, dass wenn die Entity geholt werden die Referenzen nicht mitgeladen werden. Für diesen Zweck können Referenzen im Code so deklariert werden, dass diese immer geladen werden. Dies geht gut wenn man wenig Referenzen hat. Da in diesem System alle Tabellen irgendwie verbunden sind ist diese Strategie des „Eager Fetching“ keine gute Strategie. Dies würde bedeuten, dass wenn eine Tabelle geladen wird, alle Tabellen der ganzen Datenbank geladen werden.

Aus diesem Grund wurde die Strategie angewendet, dass nur one to one, und many to one Referenzen direkt geladen werden. Many to many und many to one Referenzen müssen hingegen im nachhinein geladen werden. Aus diesem Grund wurde in den Mappern eine Methode eingebaut, welche die Entitäten lädt und ihre Referenzen direkt „fetcht“. Wird bei einer Entität eine solche Referenz gebraucht, dann muss diese Methode dafür gebraucht werden.

Da bei diesen Entitäten die Referenzen nicht immer gebraucht werden wird so sehr viel Platz gespart. [24]

4.1.3 Servlets

Die Servlet sind die erste Klassen die von der Servlet Library aufgerufen werden wenn ein HTTP Aufruf entgegen kommt. Das System wurde hauptsächlich ressourcen-basiert aufgebaut. Das heisst jede Ressource kriegt ihr Manager, ihr Mapper, ihre Entität und ihr eigenes Servlet. Einzig Ressourcen die beim Testlauf gebraucht werden, haben wenn möglich ein zusammengefasstes Servlet.

4.1.3.1 Intermediate Servlets

Für die Behandlung von Zwischentabellen gab es zwei verschiedene Ansätze. Entweder man behandelt sie in den Servlets die für die Tabellen verantwortlich sind, die in den Zwischentabellen referenziert werden. Oder sie werden in eigenen Servlets behandelt.

Es wurde entschieden, dass Zwischentabellen welche von beide Referenzen aus gebraucht werden in einem eigenen Servlet behandelt werden. Zum Beispiel eine Scoreverteilung welche eine Referenz auf Diagnose und Antwort hat muss bei der Bearbeitung von Diagnosen und bei der Bearbeitung von Antworten geändert werden können.

Zwischentabellen welche nur von einer Referenz gebraucht werden, werden hingegen direkt im jeweiligen Servlet behandelt. Die lange Beschreibung einer Diagnose die zugleich auf Sprachen referenziert

muss zum Beispiel nur bei der Bearbeitung von Diagnosen verändert werden. Trotzdem kriegen diese Tabellen auch eigene Servlets.

4.1.4 Serializer

Die Serializer bzw auch die Deserializer wurden geschrieben um Json Nachrichten direkt zu Entitäten umzuwandeln und umgekehrt. Somit können die Nachrichten die von Admin Tool kommen ohne grossen Aufwand als Java Klassen interpretiert werden. Auch die Rückgaben die meistens aus Klassen bestehen können so mit ein paar wenigen Methodenaufrufen als Json geschickt werden. [10]

4.1.5 Schnittstellenbeschreibung

Die Schnittstellenbeschreibung sind wie Json Nachrichten aufgebaut. Somit sieht man was Zuweisungen sind, was Arrays sind, was ein Objekt ist und was für ein Wert erwartet wird. Am Anfang der jeweilige Beschreibung ist der Pfad angezeigt. Mit dem Pfad kann auf die Schnittstelle zugegriffen werden. Wichtig dabei ist, dass bei „:id“ eine bestimmte Id angegeben werden kann.

Weiter unten ist als Beispiel die Schnittstellenbeschreibung eines GET Aufruf um eine Diagnose zu lesen. Die Restlichen Schnittstellenbeschreibungen sind im Anghang unter Schnittstellenbeschreibung zu finden

```
Pfad: /diagnosis/:id
Request:
{}

Response:
{
        diagnosis_id: <Number>,
        name: <String>,
        descriptions:
        [{
                description_id: <Number>,
                language_id:    <Number>,
                langage_name: <String>,
                description: <String>
        }],
        perfect_diagnosis:
        [{
                answer_id: <Number>
        }
]}
```

4.1.6 Tests

Damit die Stabilität des System gewärleistet ist wurden Tests geplannt. Da das System mehrere wenig komplexe Klassen enthält und mehrere Librarys benutzt wurden entschieden, dass nicht alles getestet werden muss. Getestet werden müssen nur die Klassen welche komplexe Vorgänge beinhalten wie die Calculators und damit das Verhalten kritisch beinträchtigen könnten.

4.1.6.1 Calculators

Die Calculators werden so getestet, dass sie alle Fälle abdecken. Damit die Tests nicht auf Datenbankeinträge zugreifen werden ManagerFakes erstellt, die diese gewünschten Einträge zurückgeben. Die wichtigsten Einträge sind dabei Diagnosen, Handlungsempfehlungen, Syndrome, Antworten und alle Scoreverteilungen. All diese müssen vor den Tests instanziert und mit den FakeManager injiziert werden.

Damit die Tests Funktionieren müssen Scoreverteilungen eingestellt werden. Diese werden nach folgendem Schema eingestellt, wobei Objekt 1-3 Handlungsempfehlungen oder Diagnosen sind.

Antwort	Objekt 1	Objekt 2	Objekt 3
1	10	124	1234
2	144	3	71
3	23	1432	4

Tabelle 8: Antworten zu Hanldungsempfehlungen/Diagnosen Scoreverteilung

4.1.6.1.1 ActionSuggestionCalculator

Die Komplixität des ActionSuggestionCalculator ist ziemlich gering. Aus diesem Grund müssen nur die Einträge der Rangliste nach ihrer Berechnung getestet werden. Als abgegebene Antworten gelten die Antworten 1, 2 und 3.

Nr.	Testbezeichnung	Erwartungswert
1	Erste Handlungsempfehlung in Ranking testen	Der erste Eintrag entspricht der geplanten Handlungsempfehlung (Handlungsempfehlung 2). Zudem stimmt der Score mit der Summe der Scoreverteilungen überein.
2	Zweite Handlungsempfehlung in Ranking testen	Der zweite Eintrag entspricht der geplanten Handlungsempfehlung (Handlungsempfehlung 3). Zudem stimmt der Score mit der Summe der Scoreverteilungen überein.
3	Letzte Handlungsempfehlung in Ranking testen	Der letzte Eintrag entspricht der geplanten Handlungsempfehlung (Handlungsempfehlung 1). Zudem stimmt der Score mit der Summe der Scoreverteilungen überein.
4	Handlungsempfehlung in Ranking mit Syndrom testen. Das Syndrom verteilt einen hohen Score auf Handlungsempfehlung 1	Handlungsempfehlung 1 ist der erste Eintrag in ranking.

Tabelle 9: ActionSuggestionCalculator Tests

4.1.6.1.2 DiagnosisCalculator

Der DiagnosisCalculator hat mehr Funktionalität. Aus diesem Grund müssen mehr Tests gemacht werden. Bei der Berechnung einer Zusatzantwort und der Perfekten Diagnose müssen aber nicht mehr alle Einträge des Rankings getestet werden. Als abgegebene Antworten gelten die Antworten 1, 2 und 3.

Nr.	Testbezeichnung	Erwartungswert
1	Methode GetDiagnosis testen	Die Diagnose entspricht der geplante erste Diagnose (Diagnose 2). Zudem stimmt der Score mit der Summe der Scoreverteilungen überein.
2	Erste Diagnose in Ranking testen	Die erste Diagnose entspricht wie vorhin der geplante erste Diagnose (Diagnose 2). Zudem stimmt der Score mit der Summe der Scoreverteilungen überein.
3	Zweite Diagnose in Ranking testen	Die zweite Diagnose entspricht der geplante zweite Diagnose (Diagnose 3). Zudem stimmt der Score mit der Summe der Scoreverteilungen überein.
4	Letze Diagnose in Ranking testen	Die erste Diagnose entspricht der geplante letze Diagnose (Diagnose 1). Zudem stimmt der Score mit der Summe der Scoreverteilungen überein.
5	Diagnose mit Zusatzantwort testen. Dieser wird ein hoher Zusatzscore verteilt.	Die erste Diagnose entspricht der Diagnose die von der Zusatzantwort eine zusätzliche Scoreverteiliung bekommt. Zudem stimmt der Score mit der Summe der Scoreverteilungen überein.
6	Die Antworten werden als Perfekte Diagnose einer Diagnose deklariert. Danach wird diese getestet.	Die erste Diagnose entspricht der geplante erste Diagnose (Diagnose 2). Zudem stimmt der Score mit der Summe der Scoreverteilungen überein.

Tabelle 10: DiagnosisCalculator Tests

4.1.6.1.3 QuestionCalculator

Die einzige Funktion des QuestionCalculator, der nextQuestionCalculator, braucht nicht mehr nur Antworten sondern auch dazugehörende Fragen. Zudem ist die Rechnung komplexer. Aus diesem Grund wird in diesem Calculator eine neue Scoreverteilung definiert.

Frage	Abhängigkeit	Antwort	Diagnose 1	Diagnose 2	Diagnose 3
1		1	10	124	1234
		2	144	3	71
2		3	23	1432	4
		4	1	1	1
3		5	4000	1	1
		6	1	1	1
4		7	1	1	1
		8	1	300	1
5	Antwort 8	9	1	1	1
		10	1	1	1

Tabelle 11: Antworten zu Diagnosen Scoreverteilung

Aus dieser neuen Scoreverteilungen werden folgende Fälle getestet:

Nr.	Testbezeichnung	Erwartungswert
1	Nächste Frage abfragen. Vorhin wurde Antwort 3 abgegeben.	Die nächste Frage ist Frage 3, da diese den grössten Einfluss auf Diagnose 1 hat.
2	Nächste Frage abfragen. Vorhin wurde Antwort 6 abgegeben.	Die nächste Frage ist Frage 2, da diese den grössten Einfluss auf Diagnose 2 hat.
3	Nächste Frage abfragen. Vorhin wurde Antwort 8 abgegeben. Auf diese Antwort ist Frage 5 Abhängig.	Die nächste Frage ist Frage 5, da sie auf Antwort 8 abhängig ist.
4	Nächste Frage abfragen. Vorher wurde die Antwort mit der grössten Scoreverteilung abgegeben.	Die nächste Frage ist nicht die Frage dessen Antwort schon abgegeben wurde.
5	Nächste Frage abfrage. Vorher wurde eine Antwort der Frage 5 so abgeändert, dass sie die grösste Scoreverteilung hat.	Die nächste Frage ist Frage 4, da Frage 5 von Frage 4 abhängig ist.
6	Nächste Frage abfragen. Vorhin wurde eine Antwort der Frage 5 so abgeändert, dass sie die grösste Scoreverteilung hat. Zudem wurden zwei Antworten abgegeben. Die erste davon gehört der Frage 4.	Die nächste Frage ist nicht Frage 5 die die grösste Scoreverteilung hat, denn Frage 5 ist von Frage 4 abhängig und diese wurde zuvor schon beantwortet.

Tabelle 12: QuestionCalculator Tests

4.2 Admin Tool

Das Admin Tool wurde wie geplannt mit HTML und AngularJS umgesetzt. Es stellt eine Single Page Application dar und ist zusammen mit dem Backend auf Heroku gehostet. Da die Funktionalität des Admin Tool schon sehr gut in den Wireframes beschrieben ist wird in diesem Kapitel nur noch auf die Probleme bei der Umsetzung und auf die Änderungen gegenüber den Wireframes eingeangen.

4.2.1 Einarbeitung AngularJS

Das Framework AngularJS erleichtert das Verwalten und Darstellen von Daten extrem. Auf übersichtliche Art können so auch komplexe Strukturen abgebildet werden. Die Konzepte von AngularJS zu verstehen und richtig anzuwenden ist nicht ganz einfach und beruht wohl auf langer Erfahrung. In kurzer Zeit musste von den Teammitgliedern also möglichst viel gelernt und aufgefasst werden, damit die Entwicklung des Admin Tools effizient von statten gehen konnte. Diese Umstände haben den Zeitplan ein wenig beeinträchtigt, jedoch noch in einem verkraftbaren Rahmen.

4.2.2 Reaktion auf Änderungen

Das Admin Tool bietet die Möglichkeit an verschiedenen Orten die gleichen Daten zu bearbeiten. Dabei reagiert es auf Input des Benutzers und synchronisiert die Änderungen sofort. AngularJS bietet unterschiedliche Ansätze um Änderungen zu erkennen.

Zum einen gibt es die Möglichkeit ein Objekt oder sogar eine ganze Liste von Objekten mit $scope.$watch(watchExpression, listener, [objectEquality]); zu beobachten. Bei einer Änderung kann man jeweils die alte und die neue Version abfragen um einen Vergleich zu machen. [3]

Zum anderen kann man die ng-change Direktive im HTML auf das Input Feld setzten und bei jeder Änderung eine angegebene Funktion aufrufen. [2]

Während des Projekts wurden beide Methoden parallel verwendet, um zu schauen, welche handlicher ist. In einem Refactor Schritt sollte dann alles einheitlich umgesetzt werden.

Das Beobachten von mehreren Objekten in einer Liste mit einem Watcher ist sehr unbrauchbar. An einem Beispiel lässt sich die Problematik gut erklären. Alle Diagnosen sind in einer Liste diagnoses abgespeichert und es gibt einen Watcher auf diese Liste. Jedes Mal wenn eine Änderung an einer Diagnose geschieht, muss nun geschachtelt über die alte und die neue Liste iteriert werden um die betreffende Diagnose zu finden.

Die ng-change Direktive dagegen ist sehr praktisch, da man das geänderte Objekt direkt der Funktion übergeben kann. Ein Nachteil davon ist, dass man keinen Zugriff auf den vorherigen Wert hat.

Es hat sich also herausgestellt, dass es viel einfacher ist mit ng-change auf Änderungen zu reagieren. Auf Grund von Zeitmangel konnte leider das Refactoring nicht mehr umgesetzt werden.

4.2.3 Authentifizierung

Eine Authentifizierung wurde vorerst nicht geplant. Ist jedoch ein wichtiger Faktor um das Admin Tool von unerlaubten Zugriffen zu schützen. Deshalb wurde eine schnelle Lösung gesucht um diese Funktionalität zu unterstützen. Via Anmeldeformular, welches die Informationen mittels HTTP Request an das Backend leitet, geschieht die Anmeldung beim Service. In der Antwort erhält das Admin Tool den Login Status und kann auf Grund diesem den boolean isLogged setzten. Welcher dann von AngularJS genutzt wird um die Inhalte darzustellen oder zu vestecken. Der Benutzer wird beim Anmelden in einem Cookie gespeichert. Beim Abmelden erfolgt auch ein Request an das Backend und das Cookie wird wieder gelöscht. Das Cookie wird bei jedem Refresh der Seite überprüft somit behält der Benutzer seinen Login Status auch beim Page Refresh. Da eine clientseitige Authentifizierung noch nicht genug sicher ist. Haben wir auch serverseitig (also im Backend) eine Validierung eingefügt. Es wird dort mit HTTP Session gearbeitet. In der Session steht eine UserID, welche bei jedem Request auf Existenz überprüft wird. Falls der User nicht existiert Antwortet das Backend mit „You are not authorized to view this data.“

4.2.4 Änderungen gegenüber Wireframes

Während der Entwicklung des Admin Tools haben sich noch einige Punkte ergeben, welche von den entworfenen Wireframes abweichen. In diesem Kapitel wird kurz auf die Änderungen eingegangen.

4.2.4.1 Anmeldung

Authentifizierung wurde zuerst nicht eingeplant deshalb haben wir die benötigten Elemente dazu noch eingefügt. Das Kapitel 4.2.3 beschreibt das Verhalten der Authentifizierung.

4.2.4.2 Layout: Fragen Zum einen sind die Funktionalitäten nun komplett spaltenweise gegliedert. Die Auswahl ob eine Frage ein Symptom oder eine Information ist befindet sich neu neben der Abhängigkeit und die beiden Elemente der Abhängigkeit sind untereinander dargestellt.

Abbildung 24: Frageansicht: Funktionalität spaltenweise

Zum anderen gab es Änderungen in der Darstellung der Navigation der Fragen. Ursprünglich geplant war es die abhängigen Fragen eingerückt unter der Top Frage darzustellen. Dies wurde allerdings nicht so umgesetzt, sondern die Fragen sind auf Wunsch des Kunden einfach alphabetisch geordnet.

4.2.4.3 Einstellungen

Die Einstellungen wurden auf einer eigenen Seite und nicht wie geplant als „Popup-Fenster“ realisiert.

5 Abschluss

5.1 Limitierungen und Verbesserungsvorschläge

Das PocketDoc System ist noch erweiterbar und weisst zum Teil Limitierungen auf. Deshalb dient dieses Kapitel dazu diese Punkte noch zusammengefasst zu präsentieren.

5.1.1 Verbesserungen

5.1.1.1 Syndrome

Syndrome bestehen sowohl aus Symptomen und Informationen. Im PocketDoc System sind dies also alle bestehenden Fragen. In der Syndrom Ansicht kann der Benutzer auswählen welche dieser Fragen dazu gehört und welche nicht. Im „Morbi cognitio“-Algorithmus wird bei der Berechnung überprüft ob ein Syndrom auftritt oder nicht dabei weiss der Algorithmus aus welchen Antworten ein Syndrom besteht und er überprüft, ob alle darin enthaltenen Antworten auftauchen. Das Backend ist also ausgereifter als das Admin Tool. Es fehlt nämlich die Möglichkeit auszuwählen ob die Ja oder die Nein-Antwort zum Syndrom gehört. Momentan wird beim klicken der Checkbox in der Syndrom Ansicht die Ja-Antwort ausgewählt.

5.1.1.2 Testen aller Diagnose auf deren Erreichbarkeit mit typischen Symptomen

Um zu überprüfen ob eine Diagnose mittels typischen Symptomen auftritt kann auf der Diagnose Ansicht der „Test Current“-Button gedrückt werden. Dabei resultiert eine Rangreihenfolge, wo man überprüfen kann ob die aktuelle Diagnose wirklich auch an erster Stelle steht und somit erfolgreich ist. Bei 100 ist die Überprüfung ob alle Diagnosen erfolgreich ereicht werden eine Fleissarbeit. Deshalb gibt es die Möglichkeit alle auf einen Knopf zu überprüfen. Dafür gibt es den „Test All“-Button, dieser öffnet einen neuen Tab und zeigt eine Liste der fehlgeschlagenen Diagnosen. Diese Funktionalität ist insofwern wichtig, weil eine Anpassung der Punkteverteilung einer Diagnose, auf alle anderen Diagnosen einen Einfluss hat.

Die Auswertung ob alle Diagnosen erfolgreich erreicht werden, könnte besser gestaltet werden dazu bietet sich an genau die gleiche Darstellung wie auch bei „Test Current“ zu verwenden dort jedoch nur die Diagnosen, welche fehlgeschlagen sind anzuzeigen.

5.1.1.3 Ng-Change Directive anstatt Watcher

In Kapitel 4.2.2 ist beschrieben, warum eine Verwendung von ng-change die bessere Variante ist. Leider konnte aus Zeitgründen das Refactoring nicht mehr erledigt werden. Diese Verbesserung ist eine dringende Empfehlung.

5.1.1.4 Loop bei Abhängigkeit

Eine Frage kann von einer Antwort einer andern Frage abhängig sein. Dabei können Loops entstehen. Beipiel: Frage 2 ist abhängig von Antwort Ja von Frage 1. Frage 1 ist abhängig von Antwort Ja von Frage 2. Dies wird im Backend bereits erfolgreicht überprüft. Jedoch ist es im Admin Tool noch möglich die Abhängigkeit zu setzen. Es resuliert in einem fehlschlagenden Request.

5.1.1.5 Loader

Um dem Benutzer anzuzeigen, dass das Admin Tool noch auf die Antwort des Backends wartet, wurde ein Loader Icon integriert. Dieses sieht man vor allem gut wenn man die Diagnose Seite neu lädt. Es wurde vergessen beim Testlauf diesen Loader zu integrieren.

5.1.2 Limitationen

5.1.2.1 Int Zahlenfelder wie ID oder Score werden im Java Backend mit einem normaler 32-Bit int initialisiert. Der maximale int Wert liegt bei 2^{31} (=2147483648). [21] Dies Betrifft folgende Elemente:

- Maximale Anzahl Einträge in der Datenbank
- Die Berechnung der Total Scores bei Diagnose und Handlungsempfehlung
- Die einzelnen Scores
- Die Einstellungen

5.1.2.2 Heroku Limitationen

Da das Projekt auf Wunsch des Kunden auf Heroku laufen soll und kein Geld investiert wurde, gelten gewisse Limitationen welche von Heroku vorgeschrieben sind. Für die Postgres Datenbank gibt es mit dem Gratis Plan die Limitation, dass bis zu 10'000 Zeilen gespeichert werden können. Die genauen Pläne und deren Einschränkungen [20], sowie weitere Limitierungen wie RAM, Traffic und so weiter [12] können auf die Heroku Webseite gefunden werden.

Abbildungsverzeichnis

Literatur

[1] *AngularJs*. URL: https://angularjs.org/.

[2] *AngularJs Api - ngChange*. URL: https://docs.angularjs.org/api/ng/directive/ngChange.

[3] *AngularJs Api - Watch*. URL: https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch.

[4] *Astah*. URL: http://astah.net/de.

[5] *Bootstrap*. URL: http://getbootstrap.com/.

[6] *Forventis GmbH*. URL: http://www.forventis.ch/.

[7] *Git*. URL: http://git-scm.com/.

[8] *Google Drive*. URL: https://www.google.com/drive/.

[9] *Google Java Styleguidelines*. URL: https://google-styleguide.googlecode.com/svn/trunk/javaguide.html.

[10] *Gson User Guide - Serialize and Deserialize*. URL: https://sites.google.com/site/gson/gson-user-guide#TOC-Custom-Serialization-and-Deserialization.

[11] *Heroku*. URL: https://www.heroku.com.

[12] *Heroku Limitierungen*. URL: https://www.heroku.com/pricing.

[13] *Hibernate*. URL: http://hibernate.org/.

[14] Samuel Huber. *Excel File zur Selbstdiagnose*. Eine Kopie des Excel Files ist auf der CD zu finden.

[15] *IntelliJ*. URL: https://www.jetbrains.com/idea/.

[16] *Microsoft Visio*. URL: http://office.microsoft.com/de-ch/visio/.

[17] *Morbi Cognitio*. URL: http://de.pons.com/%C3%83%C2%BCbersetzung/deutsch-latein/Diagnose.

[18] *Panter AG*. URL: http://www.panter.ch/.

[19] *PocketDoc Prototyp*. URL: http://quiet-samurai-3688.herokuapp.com/.

[20] *PSQL Limitierungen*. URL: https://addons.heroku.com/heroku-postgresql?utm_campaign=category&utm_medium=dashboard&utm_source=addons.

[21] *PSQL - Numeric Types*. URL: http://www.postgresql.org/docs/9.1/static/datatype-numeric.html.

[22] *Redmine*. URL: http://www.redmine.org/.

[23] *Share Latex*. URL: https://de.sharelatex.com/project.

[24] uaihebert. *Four solutions to the LazyInitializationException*. Mai 2012. URL: http://uaihebert.com/four-solutions-to-the-lazyinitializationexception/.

[25] Wikipedia. *AngularJS — Wikipedia, The Free Encyclopedia*. 2014. URL: http://en.wikipedia.org/w/index.php?title=AngularJS&oldid=638343216.

[26] Wikipedia. *Bootstrap (front-end framework) — Wikipedia, The Free Encyclopedia*. 2014. URL: http://en.wikipedia.org/w/index.php?title=Bootstrap_(front-end_framework)&oldid=637483184.

[27] Wikipedia. *Google Hangouts — Wikipedia, The Free Encyclopedia*. 2014. URL: http://en.wikipedia.org/w/index.php?title=Google_Hangouts&oldid=636202872.

[28] Wikipedia. *Hibernate (Java) — Wikipedia, The Free Encyclopedia*. 2014. URL: http://en.wikipedia.org/w/index.php?title=Hibernate_(Java)&oldid=635754056.

[29] Wikipedia. *Java Database Connectivity — Wikipedia, The Free Encyclopedia*. 2014. URL: http://en.wikipedia.org/w/index.php?title=Java_Database_Connectivity&oldid=631458684.

[30] Wikipedia. *Javadoc — Wikipedia, The Free Encyclopedia*. 2014. URL: http://en.wikipedia.org/w/index.php?title=Javadoc&oldid=627808885.

[31] Wikipedia. *PostgreSQL — Wikipedia, The Free Encyclopedia*. 2014. URL: http://en.wikipedia.org/w/index.php?title=PostgreSQL&oldid=637627549.

[32] Wikipedia. *Rational Unified Process — Wikipedia, The Free Encyclopedia*. 2014. URL: http://en.wikipedia.org/w/index.php?title=Rational_Unified_Process&oldid=638478318.

[33] Wikipedia. *Representational state transfer — Wikipedia, The Free Encyclopedia*. 2014. URL: http://en.wikipedia.org/w/index.php?title=Representational_state_transfer&oldid=638358596.

Teil II

Anhang

6 Projektabwicklung

6.1 Projektplan

6.1.1 Einführung

6.1.1.1 Zweck
Dieses Dokument dient der Beschreibung des Umfangs und der Planung der Entwicklung eines Admin Tools und einer REST Schnittstelle. [33]

6.1.1.2 Gültigkeitsbereich
Dieses Dokument gilt als Grundlage für das ganze Projekt „PocketDoc“ und hat deshalb Gültigkeit über die gesamte Projektdauer.

6.1.2 Projekt Übersicht

6.1.2.1 Anmerkung
Jegliche Nachfolgende Beschreibungen stellen eine erste Vision dar und sind für das Endprodukt nicht repräsentativ.

6.1.2.2 Vision
PocketDoc ist eine Smartphone-Applikation, die es dem Benutzer ermöglicht sich selbst zu diagnostizieren und somit entscheiden zu können, ob ein Besuch beim Doktor notwendig ist, oder ob es sich nur um eine Bagatellkrankheit handelt. Der Benutzer beantwortet einfache Ja-/Nein-Fragen, welche dann analysiert werden. Der Benutzer soll am Schluss informiert werden, welche Krankheiten in Frage kommen, ob der Gang zum Doktor notwendig ist und welche Handlungsempfehlungen für den Patienten wichtig sind.

Sowohl die Fragen wie auch die schlussendliche Diagnosen und die Handelsempfehlungen können mithilfe eines Admin-Tool bearbeitet werden. Mit dem Admin-Tool kann auch überprüft werden ob mit den richtigen Antworten die richtige Diagnosen und Handelsempfehlungen auftauchen.

6.1.2.3 Aktueller Stand
Momentan besteht ein Prototyp von PocketDoc [19], welcher mit Javascript und HTML umgesetzt ist. Zudem gibt es eine Excel Datei [14], welche mittels eines Algorithmus und vorgegebenen Antworten eine Diagnose stellt. Die Logik dieses Algorithmus ist also bereits vorhanden und funktioniert für die vorhandenen Diagnosen. Im aktuellen Prototyp ist die Fragenreihenfolge noch nicht ganz ausgereift der Kunde hat jedoch eine Idee, wie wir den Verlauf effizienter gestalten könnten.

6.1.2.4 Projektumfang
Unsere Aufgabe ist es nun einen Teilbereich der Vision umzusetzen. Da die Excel Tabelle nur schwer erweiterbar ist sollen die wichtigen Entitäten und die Logik extrahiert und danach gut strukturiert werden. Um das Projekt der Industriepartner möglichst effizient weiterzuführen ist es wichtig ein gutes Grundgerüst zu legen. In unserem Projekt PocketDoc geht es also nicht darum eine Smartphone-App zu entwickeln sondern um eine Umstrukturierung der bereits vorhandenen Daten/Logik in eine bessere Form. Am Schluss soll ein Admin Tool resultieren mit dem der Kunde Fragen, Antworten, Diagnosen, Handlungempfehlungen und Symptome erfassen und anpassen kann. Damit getestet werden kann, ob bei gegebenen Antworten die richtige Diagnose gestellt wird, kann via des Admin Tool entweder eine Testrunde gestartet werden, oder man kann in einer formularartigen Übersicht die die Antworten zu allen Fragen abgeben.

6.1.2.4.1 Datenbank
Die Daten die bisher im Excel abgespeichert wurden, wie Symptome, Fragen, Antworten, Handlungsempfehlungen und Diagnosen werden in einer Datenbank abgespeichert.

6.1.2.4.2 Business Logik
Ein Teil der Businesslogik stellt der „Morbi cognitio“-Algorithmus dar, welcher die Antworten des Benutzers auswertet. Der Algorithmus berechnet ähnlich wie ein Scoring System, ausgehend von einer Punktetabelle, die Diagnose. Ein anderer Teil ist die Koordinierung und Persistierung der Fragen und Antworten. Welche an den Benutzer gestellt bzw. vom Benutzer empfangen werden. Die Business Logik kann über eine Schnittstelle angesprochen werden und greift auf die Datenbank zu. Diese Schnittstelle wird später - nicht mehr in unserem Projekt - vom App verwendet.

6.1.2.4.3 Admin Tool
Das Admin Tool stellt eine Applikation dar, welche es dem Admin erlaubt mittels der oben erwähnten Schnittstelle auf die Datenbank zuzugreifen, Daten anzupassen und neu zu erfassen. Zudem soll man einen Testdurchlauf starten können, um sicherzustellen, ob die Daten in der Datenbank auch zu den gewünschten Diagnosen/Handlungsempfehlungen führen. Für diese Tests verwendet das Admin Tool die Business Logik. Auch für die Auswertung der Diagnose mittels des Formulars wird die Business Logik verwendet.

6.1.2.5 Lieferumfang
Das Back-End sowie das Admin-Tool und den ganzen Quellcode werden auf einem Heroku-Server deployt. Der Zugriff auf dem Server wird nach dem Projekt dem Kunden übergeben.

6.1.3 Projektorganisation

Wir sind beide auf der gleichen hierarchischen Stufe eingegliedert. Beide übernehmen ihre Kompetenzen und Erfahrungen enstrepchend ihrer Verantwortlichkeiten. Zum Entwicklungsteam kommt Prof. Dr. Eduard Glatz hinzu, welcher eine beratende Funktion einnimmt.

6.1.3.1 Organisationsstruktur
Die Organisationsstruktur wird aufgrund der geringe Anzahl Projektmitglieder sehr dynamisch geführt. Konkret heisst das, dass jeder Aufgaben im Zuständigkeitsbereich des anderes übernehmen kann.

Name	Zuständigkeit
Nathan Bourquin	Backend
Oliver Frischknecht	Admin Tool

6.1.3.2 Externe Schnittstellen
Name	Zuständigkeit
Dr. Eduard Glatz	Beratung während der Planung und Umsetzung
Dr. med. Samuel Huber	Praxispartner und Kunde
Flavio Trolese	Praxispartner und Kunde

6.1.4 Managementabläufe

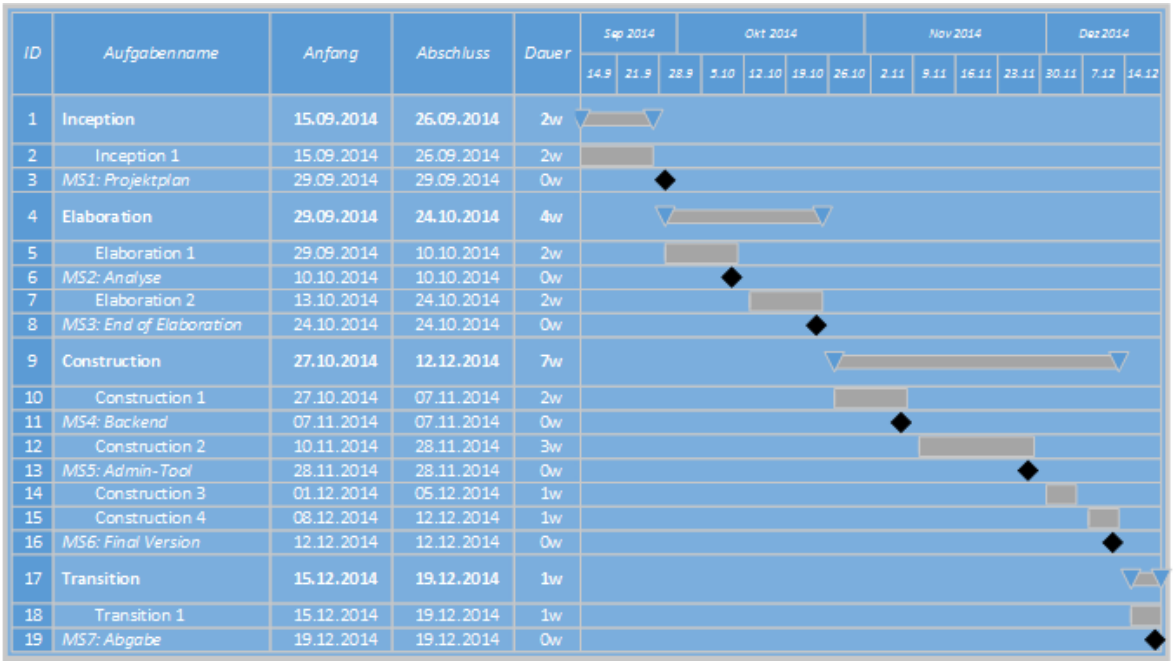
6.1.4.1 Kostenvoranschlag
Die Studienarbeit bringt 8 ECTS Punkte ein. Ein ECTS-Punkt stellt 30 Arbeitsstunden dar. Da wir die Arbeit zu zweit bewältigen, rechnen wir mit 16 ECTS Punkten. Der zeitliche Aufwand für die Studienarbeit beläuft sich somit auf 480 Arbeitsstunden.

Die Studienarbeit erstreckt sich über folgende Zeitspanne von 96 Tagen:
Starttermin: Montag, 15. September 2014
Abgabetermin: Freitag, 19. Dezember 2014

Aus der Zeitspanne ergibt sich die Projektdauer von 14 Wochen. Jedes Teammitglied sollte deshalb pro Woche 17 Stunden in diese Arbeit investieren, damit das Total von 480 Arbeitsstunden erreicht wird.

6.1.4.2 Zeitliche Planung

Folgende Abbildung zeigt einen groben Projektverlauf. Spezifische Aufgaben werden nur in der Construction Phase mit Redmine verwaltet, da dort wichtig ist das die Teammitglieder individuell arbeiten können und wissen bis wann was erledigt sein muss. Die groben Ziele der einzelnen Phasen sind im nächsten Kapitel ersichtlich.



ID	Aufgabenname	Anfang	Abschluss	Dauer
1	**Inception**	**15.09.2014**	**26.09.2014**	**2w**
2	Inception 1	15.09.2014	26.09.2014	2w
3	*MS1: Projektplan*	29.09.2014	29.09.2014	0w
4	**Elaboration**	**29.09.2014**	**24.10.2014**	**4w**
5	Elaboration 1	29.09.2014	10.10.2014	2w
6	*MS2: Analyse*	10.10.2014	10.10.2014	0w
7	Elaboration 2	13.10.2014	24.10.2014	2w
8	*MS3: End of Elaboration*	24.10.2014	24.10.2014	0w
9	**Construction**	**27.10.2014**	**12.12.2014**	**7w**
10	Construction 1	27.10.2014	07.11.2014	2w
11	*MS4: Backend*	07.11.2014	07.11.2014	0w
12	Construction 2	10.11.2014	28.11.2014	3w
13	*MS5: Admin-Tool*	28.11.2014	28.11.2014	0w
14	Construction 3	01.12.2014	05.12.2014	1w
15	Construction 4	08.12.2014	12.12.2014	1w
16	*MS6: Final Version*	12.12.2014	12.12.2014	0w
17	**Transition**	**15.12.2014**	**19.12.2014**	**1w**
18	Transition 1	15.12.2014	19.12.2014	1w
19	*MS7: Abgabe*	19.12.2014	19.12.2014	0w

Abbildung 25: Soll-Zeitplan

6.1.4.3 Phasen/Iterationen

6.1.4.3.1 Inception 1

In der Inception Phase wird der Projektplan erstellt. Dabei wird die Aufgabenstellung grob definiert, der grobbe Zeitplan bestimmt. Zudem wird in dierser Phase die administrative und organisatorische Struktur definiert.

6.1.4.3.2 Elaboration 1

In dieser Phase wird die Aufgabenstellung genau analysiert und definiert. Dazu wereden Use Cases erstellt, welche in einem Use Case Diagramm visualisiert werden. Nur umfassende und komplexe Use Cases werden im Fully Dressed Format beschrieben. Alle andern werden nur brief beschrieben. Zudem wird das Domain Model und eine Risiko Liste erstellt. Die Non Functional Requirements werden beschrieben. Desweiteren wird die Infrastruktur eingerichtet und es wird testweise die Funktionalität der einzelnen Systemkomponente überprüft.

6.1.4.3.3 Elaboration 2

Die uns unbekannten Technologien werden analysiert. Es werden Kommunikationsdiagramme und Wireframes angefertigt. Es werden für umfassenden und komplexen Use Cases, Sequenzdiagramme erstellt und so die systeminterne Kommunikation inklusive der Benutzerinteraktion visualisiert. Ausgehend vom Domain Modell wird das Klassendiagramm erstellt. Aufgrund dessen wird die logische Codestrukturierung ersichtlich. Nach Fertigstellung des Klassendiagramms wird ein Entity Relationship Modell als Diagramm für die Datenhaltung erstellt. Mit dem Abschluss dieser Iteration ist MS3 erstritten.

6.1.4.3.4 Construction 1

Auf Grund der in der Elaboration Phase gefällten Design und Architektur Entscheide, werden in den Constructionphasen das System in kleinen Schritten implementiert und getestet.

In dieser Phase wird das Backend mit dem „Morbi Cognitio“-Algorithmus und dem Zugriff auf die Datenbank entwickelt. Zu Beginn wird die Datenbank, nach ERM Diagramm eingerichtet danach wird vom einen Teammitglied die Implementierung der Servlets und der Calculator gemacht, paralell dazu fertigt das andere Teammitglied die Mapper und Manager Klassen an.

6.1.4.3.5 Construction 2

In dieser Phase wird das Admin Tool mit HTML und AngularJS umgesetzt. Da der Aufbau all unserer Seiten (siehe Kapitel 3.6) sehr ähnlich ist und die Konzepte von AngularJS für uns Neuland sind konzentriert sich ein Teammitglied zuerst auf die Realisierung einer Seite und achtet dafür auf eine gute Ordnerstruktur. Wenn die erste Seite erfolgreich zu Ende gestellt ist, werden die anderen Seiten angefertigt. Da die Testrun Seite die komplexeste ist wird sich ein Teammitglied vollständig auf dessen Korrektheit konzentrieren.

6.1.4.3.6 Construction 3

Diese Phase wird zum Refactoring der anderen Phasen benutzt.

6.1.4.3.7 Construction 4

Diese Constructionphase dient als Zeitpuffer, falls eine der anderen Phasen länger dauern sollte.

6.1.4.3.8 Transition 1

In dieser Phase werden noch die letzten Änderungen an der Dokumentation gemacht. Zudem müssen alle Unterlagen für die Abgabe fertiggestellt werden. Dazu gehören die Dokumentation, der Programmcode, ein Abstract und ein Poster.

6.1.4.4 Meilensteine

6.1.4.4.1 MS1: Projektplan

Folgende Dokumente sind beim erreichen des Meilensteins vorhanden:

- Projektplan

6.1.4.4.2 MS2: Analyse

Folgende Dokumente sind beim erreichen des Meilensteins vorhanden:

- Use Case Diagramm
- Domain Model
- Non Functional Requirements
- Risiko Liste
- Analyse von unbekannten Technologien

Ausserdem ist zu diesem Zeitpunkt die Infrastruktur eingerichtet und die Funktionalität der einzelnen Systemkomponente überprüft worden.

6.1.4.4.3 MS3: End of Elaboration

Folgende Dokumente sind beim erreichen des Meilensteins vorhanden:

- Wireframes
- Kommunikationsdiagramme
- Sequenzdiagramme
- Klassendiagramm
- Entity Relationsship Modell

6.1.4.4.4 MS4: Backend
Das Backend ist funktionsfähig implementiert und die Schnittstellen sind klar definiert.

6.1.4.4.5 MS5: Admin-Tool
Das Admin-Tool ist funktionsfähig implementiert und mit dem Backend zusammengehängt

6.1.4.4.6 MS6: Final Version
Sowohl Backend und Admin-Tool sind abgabebereit. Der Code soll hier gut strukturiert und wartbar sein.

6.1.4.4.7 MS7: Abgabe
Die Dokumentation ist fertiggestellt.

6.1.4.5 Besprechungen
Wöchentlich am Mittwoch findet eine Projektbesprechung statt. Bei unvorhergesehenen Problemen oder zu frühem erreichen eines Meilensteines können auch ausserterminliche Treffen vereinbart werden. Die Treffen dienen der Fortschrittsanalyse der letzten Woche, die aufgetretenen Probleme werden nach Priorität eingestuft und das weitere Vorgehen wird festgelegt.

6.1.5 Risikomanagement

6.1.5.1 Projektrisiken

Nr	Titel	Beschreibung	max. Schaden[h]	Eintrittswahrsch.	Gew. Schaden
1	Überraschungen bei der Verwendung von LaTeX	Bei der Verwendung von LaTeXkönnte man Zeit verlieren wenn sich die Dokumentation nicht so schnell schreiben lässt wie nötig.	10	30%	3
2	Überraschungen bei der Verwendung von Heroku	Bei der Verwendung von Heroku läuft etwas nicht wie geplant. Dadurch geht Zeit verloren.	16	10%	1.6
3	Überraschungen bei der Verwendung von AngularJs	Bei der Verwendung der unbekannten Technologie AngularJs können bestimmte Funktionen nicht so eingesetzt werden wie geplant. Dadurch wird Zeit verloren.	32	10%	3.2
4	Heroku Begrenzungen	Heroku hat verschiedene Begränzungen für Free User, wie zum Beispiel, dass eine Datenbank maximal 10'000 Rows haben darf.	24	25%	6
5	Man hat zu wenig Zeit für das Projekt	Während der Woche nehmen andere Module zu viel Zeit in Anspruch und so kommt es zu Verspätungen im Projekt.	8	25%	2

6.1.5.2 Allgemeine Risiken

Nr	Titel	Beschreibung	max. Schaden[h]	Eintrittswahrsch.	Gew. Schaden
1	Ausfall des Heroku Server	Der Heroku Server ist über den Zeitraum von mehr als einem Tag nicht verfügbar.	8	1%	0.08
2	Ausfall des Arbeitsplatzes	Ein Arbeitsplatz ist über den Zeitraum von einer Wochen nicht verfügbar.	4	4%	0.16
3	Server erleidet ein Totalschaden, alle Daten gehen verloren	Die Daten auf einem der Datenserver (Heroku, Google Drive, writeLateX) sind gelöscht und nicht mehr wiederherstellbar.	48	0.0001%	0.000048
4	Architektur wird verändert	In der Construction Phase will man die Architektur verändern.	64	5%	3.2
5	Teammember ist unmotiviert	Ein Teammember findet keine Motivation und macht seine Arbeit nicht.	4	50%	2
6	Kommunikation mit Auftraggeber ist schlecht	Auftraggeber und Entwickler kommunizieren nicht mehr miteinander. So kommt dazu, dass etwas unerwünschtes entwickelt wird.	24	2%	0.48
7	Anforderungen sind Unklar	Bestimmte Anforderungen werden nicht mitgeteilt (siehe 8) oder zu ungenau erklärt. Es müssen so spät anpassungen gemacht werden	24	8%	1.92

6.1.5.3 Umgang mit Projektrisiken

Nr	Titel	Vorbeugung	Verhalten beim Eintreten
1	Überraschungen bei der Verwendung von LaTeX	Bei der Planung wurde ein Zeitpuffer eingeplant. Zudem setzt sich jeder Entwickler während dem Semester mit LaTeXauseinander.	Im Internet nach Lösungen suchen, im Notfall Freunde mit Erfahrung fragen. Die verlorene Zeit wird im Zeitpuffer eingetragen.
2	Überraschungen bei der Verwendung von Heroku	Bei der Planung wurde ein Zeitpuffer eingeplant. Zudem setzen sich die Entwickler während der Constructionphase mit Heroku auseinander.	Das Lösungsfinden wird Priorisiert bis eine Lösung gefunden wurde. Die verlorene Zeit wird im Zeitpuffer eingetragen.
3	Überraschungen bei der Verwendung von AngularJs	Bei der Planung wurde ein Zeitpuffer eingeplant. Zudem setzt sich jeder Entwickler vor der Construction Phase mit unbekannten Technologien auseinander.	Es wird eine andere Lösung für das Problem gesucht. Die verlorene Zeit wird im Zeitpuffer eingetragen.
4	Heroku Begrenzungen	Die Arbeit mit Heroku und die Datenbankarchitektur wird so geplant, dass es nicht so weit kommt.	Bestimmte Automatisierungen müssen eine Zeit lang manuell ausgeführt werden. Architekturen müssen neu geplant werden.
5	Man hat zu wenig Zeit für das Projekt	Das Projekt erhält erste Priorität, andere Arbeiten müssen im Notfall Zuhause gemacht werden.	Arbeit die nicht gemacht wird während der Woche wird am Wochenende nachgeholt.

6.1.5.4 Umgang mit allgemeinen Risiken

Nr	Titel	Vorbeugung	Verhalten beim Eintreten
1	Ausfall des Heroku Server	Durch die Arbeit mit Git und regelmässigen Pulls ist immer ein aktueller Stand auf den Lokalen Arbeitsplätzen	Solange der Server nicht erreichbar ist wird lokal gearbeitet.
2	Ausfall des Arbeitsplatzes	Es sind immer mindestens 2 Arbeitsplätze pro Person verfügbar.	Es wird dann auf dem anderen Arbeitsplatz gearbeitet.
3	Server erleidet ein Totalschaden, alle Daten gehen verloren	Wichtige Daten sind auf mehreren Platformen vorhanden.	Die Daten die wichtig sind werden wieder zurückkopiert und die jeweilige Umgebung wird wieder aufgebaut.
4	Architektur wird verändert	Es wird genug Zeit in der Elaboration Phase inverstiert um dafür zu sorgen, dass die Architektur nicht mehr verändert werden muss.	Die Architektur wird angepasst wenn dafür Zeit übrig bleibt und die dafür verwendete Zeit wir im Zeitpuffer eingetragen.
5	Teammember ist unmotiviert	Es werden bestimmte „SA“ - Zeiten eingeplannt wo man sich zusammenreisst und für die SA Arbeitet	Arbeit die nicht gemacht wird während der Woche wird am Wochenende nachgeholt.
6	Kommunikation mit Auftraggeber ist schlecht	Es werden mindestens alle zwei Wochen Meetings durchgeführt, um den aktuellen Stand zu besprechen.	Der Auftraggeber wird kontaktiert um dies sobald wie möglich wieder in Ordnung zu bringen.
7	Anforderungen sind Unklar	Die Anforderungen werden bei jedem Meeting durch den Aktuellen stand der Arbeit wieder neu angeschaut. Offene Fragen werden dann gestellt.	Man konntaktiert den Auftraggeber und behebt gemeinsam das Prob

6.1.6 Arbeitspakete

Die Arbeitspakete wurden in Redmine erfasst und sind in Kapitel 32 nochmals dokumentiert.

6.1.7 Infrastruktur

6.1.7.1 Hardware

Für die Software Entwicklung und das Dokumentieren benötigt jedes Teammitglied seinen eigenen Laptop. Für die Datensicherung, die Testumgebung sowie das automatische „Building“ unseres Programmes verwenden wir Heroku.

6.1.7.2 Software

Während dem Projekt wir folgende Software für die Programmierung und Dokumentierung verwendet:

- Git [7]
- Heroku [11]
- Intellij (Backend) [15]
- Hibernate (Backend) [13]
- Bootstrap (Admin Tool) [5]
- Angular JS (Admin Tool) [1]
- Google Drive [8]
- ShareLaTeX [23]
- Redmine [22]
- Visio [16]
- Astah [4]

6.1.7.3 Kommunikation

Die Projektinternet Kommunikation wird in der Regel mündlich erfolgen. Falls dies nicht möglich sein sollte stehen E-Mail und Google Hangouts [27] zur Verfügung. Die Besprechungen mit den Industriepartnern werden über Google Hangouts abgehalten.

6.1.8 Qualitätsmassnahmen

Massnahme	Beschreibung	Zeitraum	Ziel
Erfassung der Arbeitspackete	Es werden alle Arbeitspakete während den Construction Phasen in Redmine erfasst	Gesamtes Projekt	Übersichtliches Changemanagement
Zeiterfassung	Jedes Teammitglied trägt seinen Zeitaufwand in Redmine ein.	Gesamtes Projekt	Einhalten des Zeitplans
Meeting	Wöchentlich sitzt das Team zusammen analysiert den Fortschritt und plant das weitere Vorgehen	Gesamtes Projekt	Klar definierte Ziele innerhalb einzelner Iterationen
Code Style Guidelines	Es werden bewährte Code Style Guidelines verwendet. [9]	Construction	Einheitliche Terminologie gute Lesbarkeit des Programmcodes für alle Teammitglieder
Code-Reviews	Es wird gegenseitig sporadisch Feedback zum Progammierstil gegeben.	Construction	Qualitativ hochstehender Code
Testcoverage	Die Tests sollen den gesammten Logischen Teil abdecken.	Construction	Stabiles Gesamtsystem
Dokumentation	Design-entscheidungen und Implementation werden dokumentiert	Gesamtes Projekt	Vollständige Systemdokumentation

6.1.8.1 Dokumentation

Die Dokumentation wird mit LaTeX auf www.sharelatex.com erstellt. So können alle Teammitglieder gleichzeitig an der Dokumentation arbeiten. Vor einer Abgabe wird das jeweilig einzureichende Dokument in Teilen auf die Teammitglieder verteilt.

6.1.8.2 Projektmanagement

Da Redmine bereits sämtliche Funktionalität anbietet, um das Projektmanagement übersichtlich und flexibel zu gestalten, wird unser Projektmanagement während den Construction Phasen vollständig in Redmine ausgeführt. Da die SA ein Projekt mit fixem Anfang und Ende ist finden wir es sinnvoll nach RUP [32] vorzugehen, dabei ist es wichtig die Anforderungen gut zu erfassen und dannach diese nicht mehr gross abzuändern. Wir möchten aber noch zwei kleine Anpassung machen nämlich führen wir nach jeder Iteration eine Art „Sprint"-Meeting ein. Zudem wollen wir agil vorgehen.

Die Entscheidung RUP einzusetzen wurde einstimmig gefällt. Wir wollten vor allem etwas einsetzen was uns bekannt war. RUP wurde uns in SE2 beigebracht und musste im SE2-Projekt eingesetze werden. Die Aufteilung der Iterationen gefällt uns zudem sehr. Was uns aber noch fehlte war die Möglichkeit agil vorzugehen, was wir kurzerhand mit Zweiwochen-Meetings mit dem Kunde einbauten.

6.1.8.2.1 Arbeitspakete

Es wird pro Arbeitspaket ein Ticket erstellt. Beim Erstellen wird dabei pro Ticket die geplante Zeitspanne angegeben, einen Verantwortlichen referenziert und die passende Iteration gewählt.

6.1.8.2.2 Bugs

Beim Auftreten von Bugs wird umgehend ein neues Ticket in Redmine erstellt mit der Zugehörigkeit des Verursachers. Bugs die das Weiterarbeiten am Projekt hindern werden mit höchster Priorität markiert. Bugs die kaum auffallen bekommen eine kleinere Priorität.

6.1.8.3 Entwicklung

Die Industriepartner wünschen, dass das Entwicklungsteam Heroku verwenden soll, um den Programmcode zu verwalten. Der Code wird also wie gewünscht auf dem Heroku Gitserver verwaltet und somit kontinuierlich gesichert.

6.1.8.3.1 Vorgehen

Bei der Entwicklung verfolgen wir den Test-Driven Development Ansatz, welchen wir im SE2 Unterricht erlernt haben. Änderungen am Code (Changes) werden so häufig wie möglich lokal committed. Somit ist es möglich das Auftreten von Bugs schnell auf einen Change zurückzuführen. Beim Abschliessen eines Issues wird die lokale Version auf den Git Server gepusht. Vor Beginn jedes Arbeitspakets muss die aktuellste Version des Programmcodes aus dem Git gepulled werden.

6.1.8.3.2 Code Reviews

Code Reviews geschehen sporadisch jedoch ca. einmal pro Woche. Es wird dabei jeweils der Code des andern Entwicklers angeschaut und Feedback gegeben. Diese Reviews sollen nicht länger als eine Halbstunde dauern.

6.1.8.3.3 Code Style Guidelines

Das Entwicklungsteam hat sich auf folgende Code Style Guidelines geeinigt: Google Styleguide [9]

6.1.8.3.4 Testing

Das Backend wird mithilfe von JUnit tests getestet. Damit werden alle Algorithmen und die wichtigsten Klassen des Backends getestet.
Das Admin-Tool wird von Hand getestet.

6.1.9 Plan und Ist-Auswertung

Mit 466.25 Stunden, welches das Team in diese Projekt investiert hat, wurde der von der SA vorgesehene Rahmen von 448 Stunden gut eingehalten. Dabei haben beide Teammitglieder ähnlich viel Zeit investiert für Nathan Bourquin 235.5 Stunden, für Oliver Frischknecht 230.75 Stunden.

Folgende Grafik zeigt die IST-Zeitauswertung des Projekts.

	Inception 1	Elaboration 1	Elaboration 2	Construction 1	Construction 2	Construction 3	Construction 4	Transition 1	Total
Nathan Bourquin	10.25	22	17.5	47.75	55.75	21.5	36.5	24.25	**235.5**
Oliver Frischknecht	12	22.5	16.5	41	58.5	24.75	34.5	21	**230.75**

Abbildung 26: Zeitplanung: IST

Folgende Grafik zeigt die geplante für die Phasen geplante Zeit des Projekts.

	Inception 1	Elaboration 1	Elaboration 2	Construction 1	Construction 2	Construction 3	Construction 4	Transition 1	Total
Team	46	55	38	68	102	38	38	38	**423**

Abbildung 27: Zeitplanung: Plan

Folgende Grafik zeigt die IST-Auswertung des Projekts auf Themen verteilt.

Design	63
Entwicklung	229.5
Dokumentation	93
Sitzungen	46
Selfstudy	34.75
Total	**466.25**

Abbildung 28: Zeitplanung: Themen

Das ganze noch in einem Diagramm.

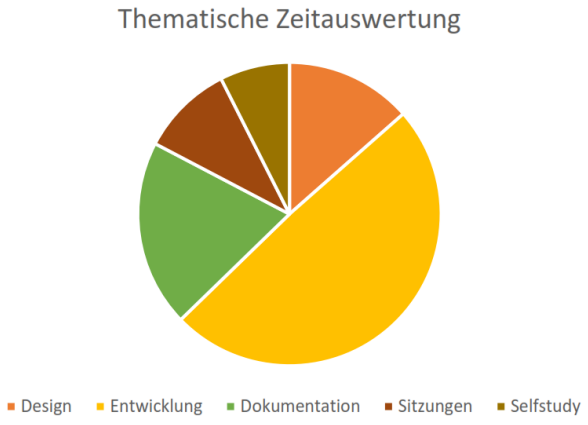
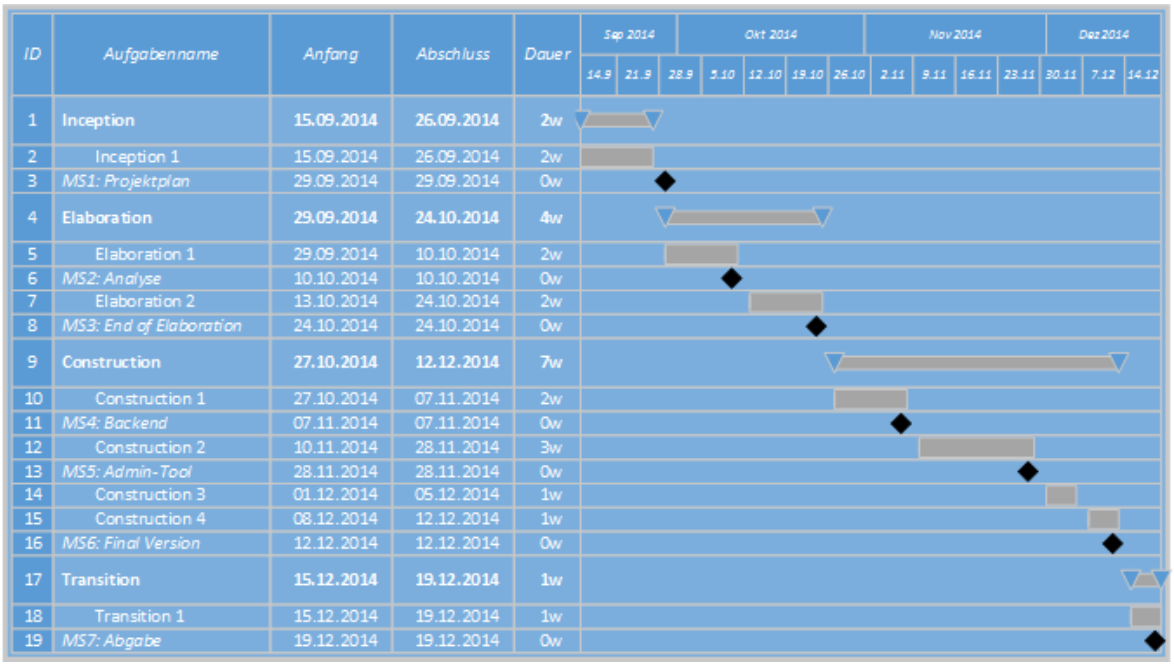


Abbildung 29: Zeitplanung Diagramm: Themen

6.1.10 Vergleich Zeitplan

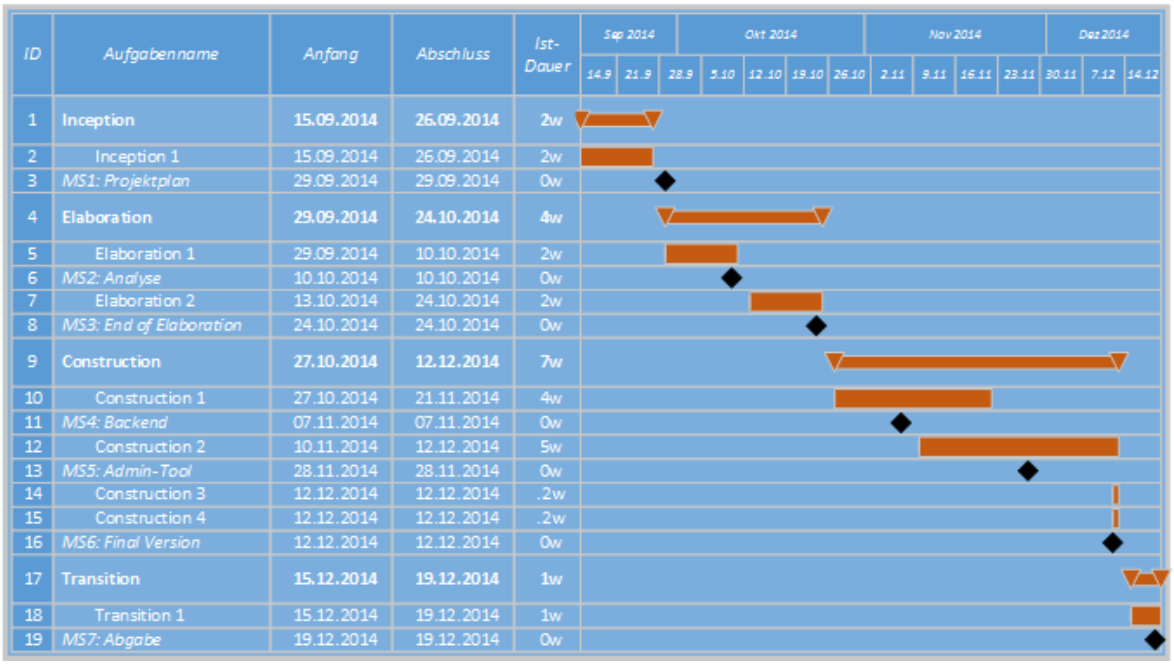
Folgende Abbildung zeigt den geplanten Zeitplan, welcher in Abbildung 25 schon dargestellt wurde. Hier wird er zum besseren Vergleich noch einmal angezeigt.



ID	Aufgabenname	Anfang	Abschluss	Dauer
1	**Inception**	**15.09.2014**	**26.09.2014**	**2w**
2	Inception 1	15.09.2014	26.09.2014	2w
3	*MS1: Projektplan*	29.09.2014	29.09.2014	0w
4	**Elaboration**	**29.09.2014**	**24.10.2014**	**4w**
5	Elaboration 1	29.09.2014	10.10.2014	2w
6	*MS2: Analyse*	10.10.2014	10.10.2014	0w
7	Elaboration 2	13.10.2014	24.10.2014	2w
8	*MS3: End of Elaboration*	24.10.2014	24.10.2014	0w
9	**Construction**	**27.10.2014**	**12.12.2014**	**7w**
10	Construction 1	27.10.2014	07.11.2014	2w
11	*MS4: Backend*	07.11.2014	07.11.2014	0w
12	Construction 2	10.11.2014	28.11.2014	3w
13	*MS5: Admin-Tool*	28.11.2014	28.11.2014	0w
14	Construction 3	01.12.2014	05.12.2014	1w
15	Construction 4	08.12.2014	12.12.2014	1w
16	*MS6: Final Version*	12.12.2014	12.12.2014	0w
17	**Transition**	**15.12.2014**	**19.12.2014**	**1w**
18	Transition 1	15.12.2014	19.12.2014	1w
19	*MS7: Abgabe*	19.12.2014	19.12.2014	0w

Abbildung 30: Soll-Zeitplan

Beim Ist-Zeitplan sieht man, dass fast alles eingehalten werden konnte. Man kann feststellen, dass die Constructionphasen anders aussehen. Die Construction 1 und 2 dauern länger und überlappen sich. Der Grund dafür war, dass ein Entwickler mit dem Frontend anfing während der andere Entwickler das Backend anpasste. Als dies Fertig war musste das Frontend fertig gemacht werden, was länger dauerte. Aus diesem Grund mussten Zeitpuffer benutzt werden und auf Refactoring verzichtet werden.



ID	Aufgabenname	Anfang	Abschluss	Ist-Dauer
1	**Inception**	**15.09.2014**	**26.09.2014**	**2w**
2	Inception 1	15.09.2014	26.09.2014	2w
3	*MS1: Projektplan*	29.09.2014	29.09.2014	0w
4	**Elaboration**	**29.09.2014**	**24.10.2014**	**4w**
5	Elaboration 1	29.09.2014	10.10.2014	2w
6	*MS2: Analyse*	10.10.2014	10.10.2014	0w
7	Elaboration 2	13.10.2014	24.10.2014	2w
8	*MS3: End of Elaboration*	24.10.2014	24.10.2014	0w
9	**Construction**	**27.10.2014**	**12.12.2014**	**7w**
10	Construction 1	27.10.2014	21.11.2014	4w
11	*MS4: Backend*	07.11.2014	07.11.2014	0w
12	Construction 2	10.11.2014	12.12.2014	5w
13	*MS5: Admin-Tool*	28.11.2014	28.11.2014	0w
14	Construction 3	12.12.2014	12.12.2014	.2w
15	Construction 4	12.12.2014	12.12.2014	.2w
16	*MS6: Final Version*	12.12.2014	12.12.2014	0w
17	**Transition**	**15.12.2014**	**19.12.2014**	**1w**
18	Transition 1	15.12.2014	19.12.2014	1w
19	*MS7: Abgabe*	19.12.2014	19.12.2014	0w

Abbildung 31: Ist-Zeitplan

6.1.11 Arbeitspakete

In der Abbildung 32 sieht man die während den Construction-Phasen erstellten Arbeitspakete. Hier sieht man, dass aus Zeitgründen die Refactorings nicht mehr gemacht werden konnten.

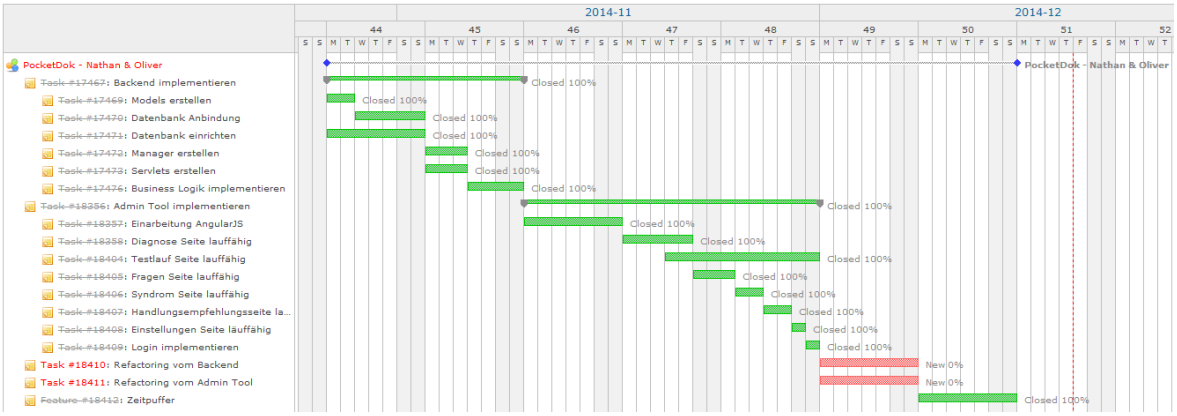


Abbildung 32: Arbeitspakete

6.1.12 Codestatistik

In dieser Arbeit wurden verschiedene Programmiersprachen verwendet, im Backend wurde mit Java programmiert, während im Admin Tool mit Javascript und HTML gearbeitet wurde. In der Abbildung 33 sieht man, dass für also das fürs Backend 9766 und fürs Admin Tool 3062 Zeilen Code geschrieben wurden.

Extension	Count	Lines	Lines MIN	Lines MAX	Lines AVG
css *(CSS files)*	1x	61	61	61	61
html *(HTML files)*	7x	741	27	144	105
java *(Java classes)*	104x	9714	9	413	93
js *(JS files)*	27x	2270	7	501	84
xml *(XML configuration file)*	1x	52	52	52	52
Total:	**140x**	**12838**	**156**	**1171**	**395**

Abbildung 33: Codestatistik

6.2 Testprotokolle

6.2.1 ActionSuggestionCalculator

Nr.	Testbezeichnung	Testresultat
1	Erste Handlungsempfehlung in Ranking testen	Der erste Eintrag war wie geplant Handlungsempfehlung 2. Der Score entsprach ebenfalls der Summe der Scoreverteilungen.
2	Zweite Handlungsempfehlung in Ranking testen	Der zweite Eintrag war wie geplant Handlungsempfehlung 3. Der Score entsprach ebenfalls der Summe der Scoreverteilungen.
3	Letzte Handlungsempfehlung in Ranking testen	Der letze Eintrag war wie geplant Handlungsempfehlung 1. Der Score entsprach ebenfalls der Summe der Scoreverteilungen.
4	Handlungsempfehlung in Ranking mit Syndrom testen. Das Syndrom verteilt einen hohen Score auf Handlungsempfehlung 1	Der erste Eintrag im Ranking war wie geplant Handlungsempfehlung 1, da dort das Syndrom einen hohen Score verteilt. Der Scoore entsprach ebenfalls der Summe der Scoreverteilungen.

Tabelle 13: ActionSuggestionCalculator Tests

6.2.2 DiagnosisCalculator

Nr.	Testbezeichnung	Testresultat
1	Methode GetDiagnosis testen	Die resultierende Diagnose war Diagnose 2 und entsprach so der geplanten ersten Diagnose.
2	Erste Diagnose in Ranking testen	Die resultierende erste Diagnose im Ranking war Diagnose 2 und entsprach so der geplanten ersten Diagnose. Die Summe der Scoreverteilungen entsprach zudem dem Score der resultierenden ersten Diagnose.
3	Zweite Diagnose in Ranking testen	Die resultierende zweite Diagnose im Ranking war Diagnose 3 und entsprach so der geplanten zweiten Diagnose. Die Summe der Scoreverteilungen entsprach zudem dem Score der resultierenden zweiten Diagnose.
4	Letze Diagnose in Ranking testen	Die resultierende letzte Diagnose im Ranking war Diagnose 1 und entsprach so der geplanten letzen Diagnose. Die Summe der Scoreverteilungen entsprach zudem dem Score der resultierenden letzen Diagnose.
5	Diagnose mit Zusatzantwort testen. Dieser wird ein hoher Zusatzscore verteilt.	Die erste Diagnose des Rankings entsprach der Diagnose die den Zusatzscore bekommt. Die Summe der Scoreverteilungen entsprach zudem dem Score der resultierenden ersten Diagnose.
6	Die Antworten werden als Perfekte Diagnose einer Diagnose deklariert. Danach wird diese getestet.	Die erste Diagnose des Rankings entsprach der geplanten ersten Diagnose. Die Summe der Scoreverteilungen entsprach zudem dem Score der resultierenden ersten Diagnose.

Tabelle 14: DiagnosisCalculator Tests

6.2.3 QuestionCalculator

Nr.	Testbezeichnung	Testresultat
1	Nächste Frage abfragen. Vorhin wurde Antwort 3 abgegeben.	Die nächste Frage war wie erwartet Frage 3.
2	Nächste Frage abfragen. Vorhin wurde Antwort 6 abgegeben.	Die nächste Frage war wie erwartet Frage 2.
3	Nächste Frage abfragen. Vorhin wurde Antwort 8 abgegeben. Auf diese Antwort ist Frage 5 Abhängig.	Die nächste Frage war wie erwartet Frage 5.
4	Nächste Frage abfragen. Vorher wurde die Antwort mit der grössten Scoreverteilung abgegeben.	Die nächste Frage war wie erwartet nicht die Frage dessen Antwort schon abgegeben wurde.
5	Nächste Frage abfrage. Vorher wurde eine Antwort der Frage 5 so abgeändert, dass sie die grösste Scoreverteilung hat.	Die nächste Frage war wie erwartet Frage 4.
6	Nächste Frage abfragen. Vorhin wurde eine Antwort der Frage 5 so abgeändert, dass sie die grösste Scoreverteilung hat. Zudem wurden zwei Antworten abgegeben. Die erste davon gehört der Frage 4.	Die nächste Frage war wie erwartet nicht Frage 5.

Tabelle 15: QuestionCalculator Tests

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

INFORMATIK

Aufgabenstellung zur Studienarbeit HS 2014

„PocketDok - App für die Selbstdiagnose“

Gruppe: Frischknecht, Oliver / Bourquin, Nathan

Vision

Wir haben die Vision, dass Menschen sich jederzeit, überall und unabhängig von ihrem Versicherungsmodell einen ersten medizinischen Rat holen können. Dabei fasziniert uns die Herausforderung, eine grundlegende ärztliche Tätigkeit zu automatisieren, sowie das sich damit eröffnende gesellschaftliche und ökonomische Potential.
PocketDok ist ein Web- und App-basiertes medizinisches Expertensystem mit dessen Hilfe sich Bagatellerkrankungen erkennen und von schwereren Erkrankungen unterscheiden lassen. Damit können User zuverlässig entscheiden, ob sie sich selber behandeln können oder einen Arzt aufsuchen sollten. Im Falle einer Bagatellkrankeit gibt das System Therapieempfehlungen ab und erlaubt dem User die Evaluation seines Heilungsverlaufs.
Unsere einmalige Chance besteht darin, als Erste ein echtes Empowerment des Nutzers in einem der wichtigsten Lebensbereiche, der Gesundheit, zu bewirken und gleichzeitig die Kosten für den Patienten und das Gesundheitswesen zu reduzieren. Unser System leistet einen entscheidenden Beitrag zum Patienten-Empowerment, indem es die Selbstbestimmung erkrankter Menschen unterstützt. Als weiterer Effekt werden teure medizinische Institutionen von Bagatellfällen entlastet, was deren Verfügbarkeit erhöht und Gesundheitskosten spart.

Stand

Wir haben einen Algorithmus entwickelt, der bisher rund fünfzig Bagatellerkrankungen anhand von Ja-/Nein Symptomabfragen erkennt. Ferner wurde ein einfacher, auf Webtechnologien basierender *Proof of Concept (Prototyp)* mit beschränktem Funktionsumfang realisiert, welcher bei ersten Tests mit medizinischen Fachexperten und Medizinlaien auf positive Resonanz gestossen ist. Der Proof of Concept steht unter http://quiet-samurai-3688.herokuapp.com/ zur Verfügung.

Aufgabe

Basierend auf dem bestehenden Algorithmus soll im Rahmen einer Studien- oder Bachelorarbeit sowohl das Backend wie auch das Frontend der Applikation neu konzipiert, designed und umgesetzt werden.
Der umzusetzende Funktionsumfang kann - falls gewünscht - bei Projektstart abschliessend in Diskussion mit dem Praxispartner definiert werden. Es gibt zurzeit keine Vorgaben betreffend die einzusetzenden Technologien. Das Frontend kann beispielsweise auch für iOs, Android

oder Windows Phone umgesetzt werden.

Allgemeine Vorgaben

Die Arbeit ist gemäss den allgemeinen Vorgaben [1] durchzuführen. Dies beinhaltet auch Vorgaben zur Berichtsgestaltung.

Termine

15. Sept. 2014	Arbeitsbeginn
19. Dez. 2014	Abgabe des Berichts an den Betreuer (bis 17:00)

Weitere Termine: siehe Terminangaben auf dem HSR-Web (intern).

Betreuung

Betreuer: Prof. Dr. Eduard Glatz, Email: eglatz@hsr.ch

Während der Durchführung der Arbeit findet nach Möglichkeit regelmässig jede Woche eine Besprechung mit dem Betreuer statt. Dazu werden entsprechende Termine bei Arbeitsbeginn festgelegt. Die Studierenden verfassen vor jder Besprechun eine Traktandenliste, die spätestens am Vortag dem Betreuer per Email zu senden ist. Jede Besprechung wird von den Studierenden in einem Protokoll dokumentiert.

Praxispartner

Forventis GmbH, Kontaktperson: Dr. med. Samuel Huber (Email: huber@forventis.ch)

Panter AG, Kontaktperson: Hr. Flavio Trolese (Email: tro@panter.ch)

Referenzen

[1] Glatz, E., „Vorgaben zur Arbeitsdurchführung“, Ausgabe des 11. September 2014.

Rapperswil, den 11. Sept. 2014

Eduard Glatz

8 Klassendiagramm

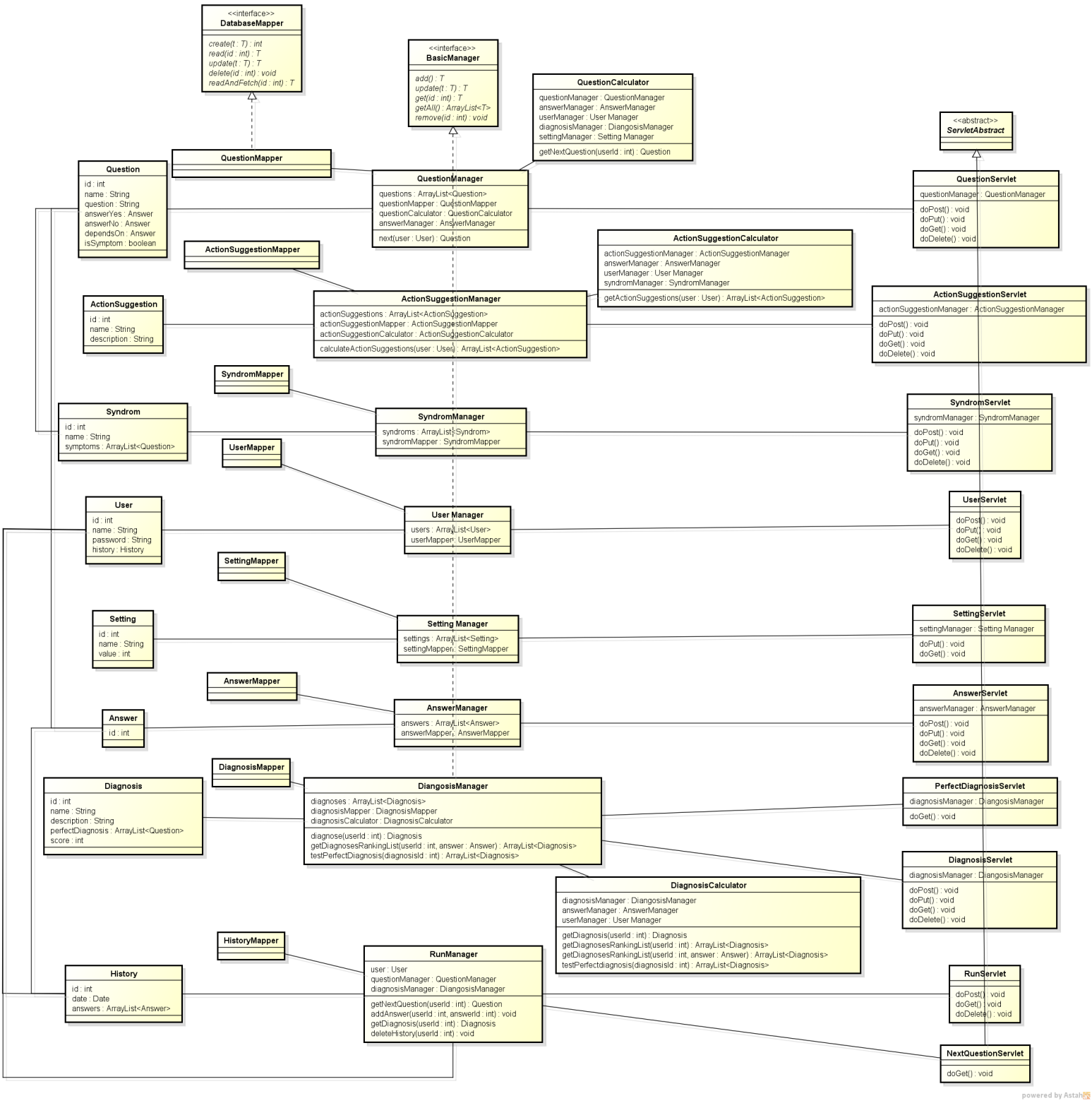


Abbildung 34: Klassendiagramm

9 Kommunikationsdiagramme

9.1 Handlungsempfehlungen verwalten

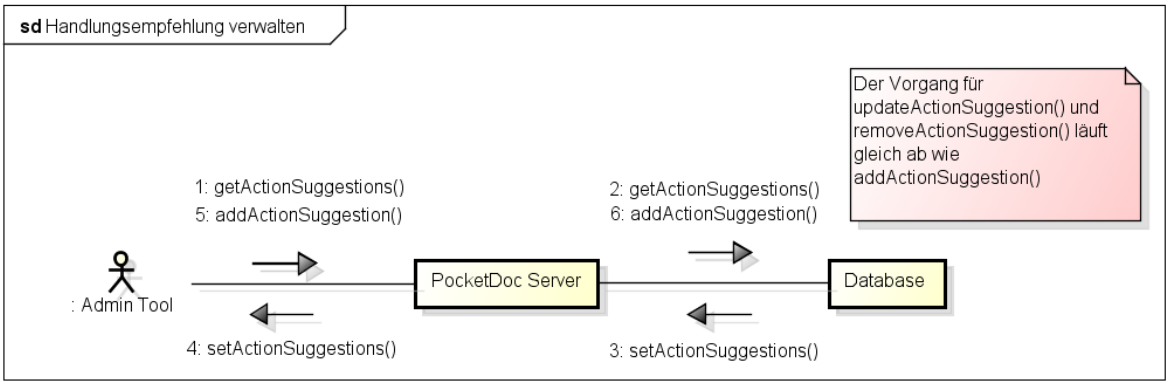


Abbildung 35: Handlungsempfehlungen verwalten

9.2 Fragen / Antworten verwalten

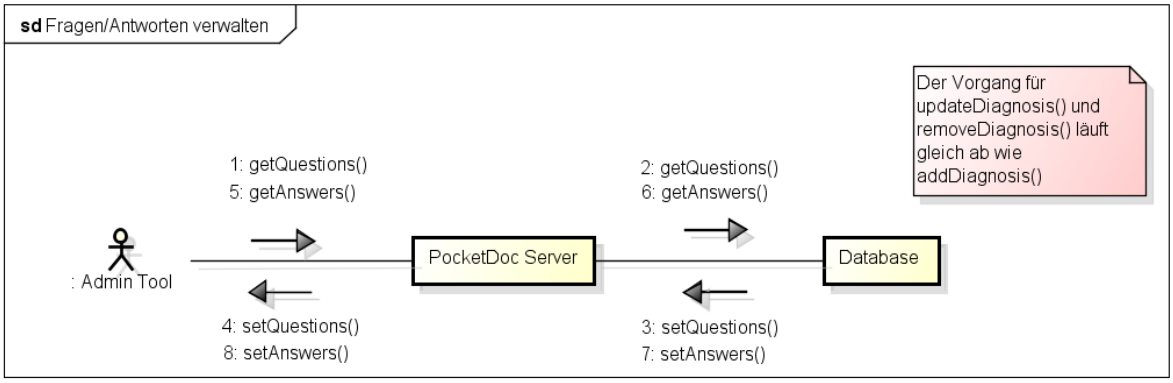


Abbildung 36: Fragen / Antworten verwalten

9.3 Syndrome verwalten

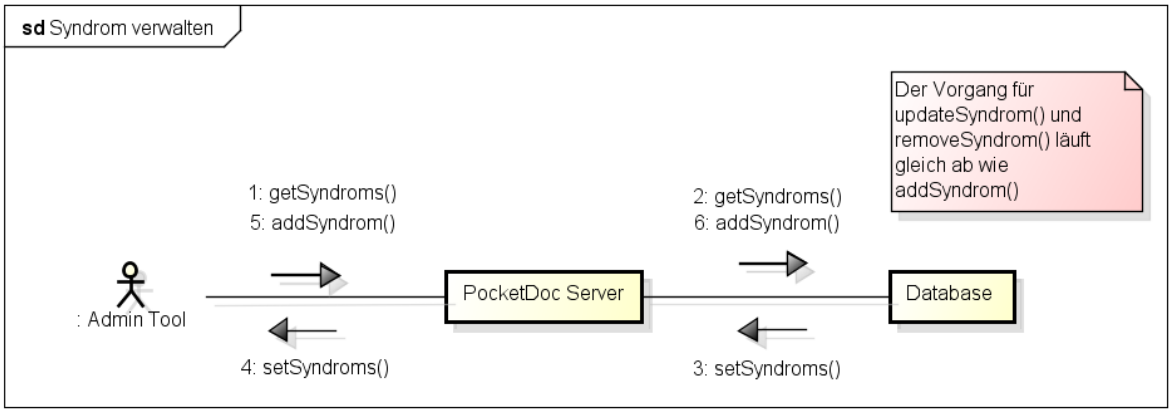


Abbildung 37: Syndrome verwalten

9.4 Testregeln verwalten

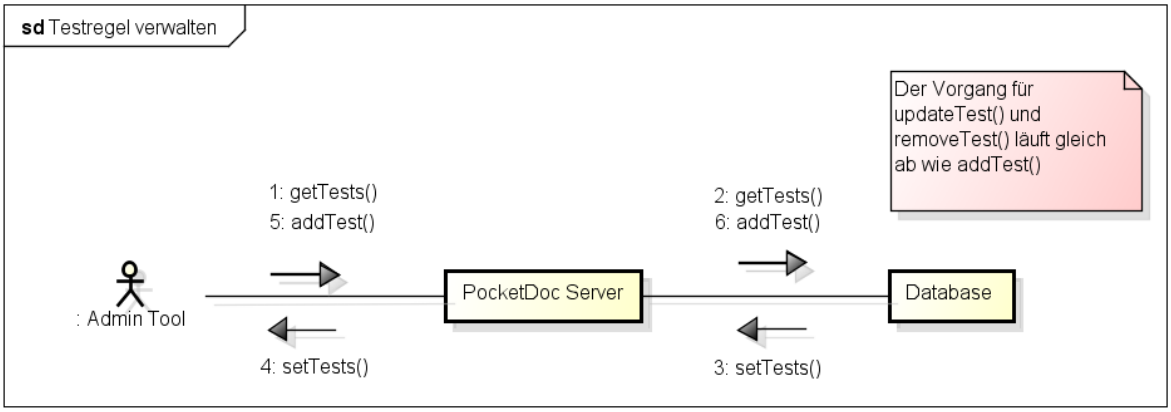


Abbildung 38: Testregeln verwalten

9.5 Einstellungen verwalten

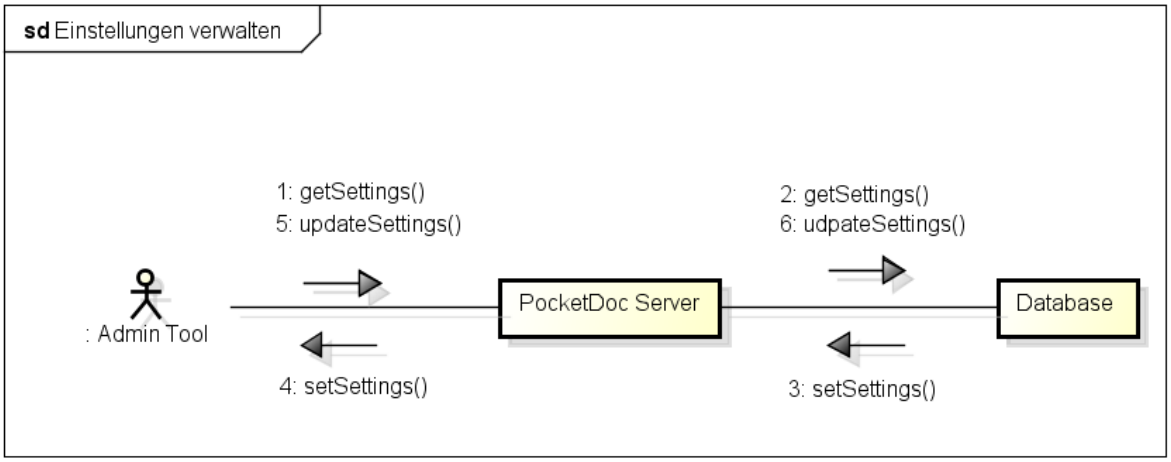


Abbildung 39: Einstellungen verwalten

10 Schnittstellenbeschreibung

10.1 ActionSuggestionServlet

10.1.1 POST

```
Pfad: /actionSuggestion
Request:
{}

Response:
{
        action_suggestion_id: <Number>,
        descriptions:
        [{
                description_id: <Number>,
                language_id: <Number,
                language_name: <String>,
                description: <String>
        }]
}
```

10.1.2 PUT

```
Pfad: /actionSuggestion
Request:
{
        action_suggestion_id: <Number>,
        name: <String>
}

Response:
{}
```

10.1.3 DELETE

```
Pfad: /actionSuggestion/:id
Request:
{}
Response:
{}
```

10.1.4 GET (All)

```
Pfad: /actionSuggestion
Request:
{}

Response:
[{
        action_suggestion_id: <Number>,
        name: <String>,
        descriptions:
        [{
                description_id: <Number>,
                language_id: <Number,
                language_name: <String>,
                description: <String>
        }]
}]
```

10.1.5 GET

```
Pfad: /actionSuggestion/:id
Request:
{}

Response:
{
        action_suggestion_id: <Number>,
        name: <String>,
        descriptions:
        [{
                description_id: <Number>,
                language_id: <Number,
                language_name: <String>,
                description: <String>
        }]
}
```

10.2 DiagnosisServlet

10.2.1 POST

```
Pfad: /diagnosis
Request
{}

Response:
{
        diagnosis_id: <Number>,
        descriptions:
        [{
                description_id: <Number>,
                language_id:    <Number>,
                langage_name: <String>,
                description: <String>
        }]
}
```

10.2.2 PUT

```
Pfad: /diagnosis
Request
{
        diagnosis_id: <Number>,
        name: <String>,
        perfect_diagnosis:
        [{
                answer_id: <Number>
        }]
}

Response:
{}
```

10.2.3 DELETE

```
Pfad: /diagnosis/:id
Request:
{}
Response:
{}
```

10.2.4 GET (All)

```
Pfad: /diagnosis
Request:
{}

Response:
[{
        diagnosis_id: <Number>,
        name: <String>,
        descriptions:
        [{
                description_id: <Number>,
                language_id:    <Number>,
                langage_name: <String>,
                description: <String>
        }],
        perfect_diagnosis:
        [{
                answer_id: <Number>
        }]
}]
```

10.2.5 GET

```
Pfad: /diagnosis/:id
Request:
{}

Response:
{
        diagnosis_id: <Number>,
        name: <String>,
        descriptions:
        [{
                description_id: <Number>,
                language_id:    <Number>,
                langage_name: <String>,
                description: <String>
        }],
        perfect_diagnosis:
        [{
                answer_id: <Number>
        }
]}
```

10.3 LoginServlet

10.3.1 POST

```
Pfad: /login
Request:
{
        name: <String>,
        password: <String>
}

Response:
{
        status : <Boolean>
}
```

10.4 LogoutServlet

10.4.1 POST

```
Pfad: /logout
Request:
{}

Response:
{
        status : <Boolean>
}
```

10.5 NextQuestionServlet

10.5.1 GET

```
Pfad: /nextQuestion/user/:id
Request:
{}

Response:
{
        question_id: <Number>,
        name: <String>,
        is_symptom: <Boolean>,
        description:
        [{
                description_id: <Number>,
                language_id: <Number>,
                language_name: <String>,
                description: <String>
        }],
        answer_no:
        {
                answer_id: <Number>,
                has_dependency: <Boolean>
        },
        answer_yes: {
                answer_id: <Number>
                has_dependency: <Boolean>
        }
}
```

10.6 PerfectDiagnosisServlet

10.6.1 GET

```
Pfad: /perfectDiagnosis/:id
Request:
{}

Response:
{
        perfect_diagnosis_result: <Boolean>,
        perfect_diagnosis_rankings:
        [{
                diagnosis_id: <Number>,
                name: <String>,
                score: <Number>,
                descriptions:
                [{
                        description_id: <Number>,
                        language_id: <Number>,
                        language_name: <String>,
                        description: <String>
                }],
                perfect_diagnosis:
                [{
                        answer:
                        {
                                answer_id: <Number>,
                                has_dependency: <Boolean>
                        }
                }]
        }]
}
```

10.7 QuestionServlet

10.7.1 POST

```
Pfad: /question
Request:
{}

Response:
{
        question_id: <Number>,
        is_symptom: <Boolean>,
        descriptions:
        [{
                description_id: <Number>,
                language_id: <Number>,
                language_name: <String>,
                description: <String>
        }],
        answer_no:
        {
                answer_id: <Number>,
                has_dependency: <Boolean>
        },
        answer_yes:
        {
                answer_id: <Number>,
                has_dependency: <Boolean>
        }
}
```

10.7.2 PUT

```
Pfad: /question
Request:
{
        question_id: <Number>,
        name: <String>,
        description: <String>,
        dependence: <Number>,
        is_symptom: <Boolean>
}

Response:
{}
```

10.7.3 DELETE

```
Pfad: /question/:id
Request:
{}

Response:
{}
```

10.7.4 GET (All)

```
Pfad: /question
Request:
{}

Response:
[{
        question_id: <Number>,
        name: <String>,
        is_symptom: <Boolean>,
        descriptions:
        [{
                description_id: <Number>,
                language_id: <Number>,
                language_name: <String>,
                description: <String>
        }],
        answer_no:
        {
                answer_id: <Number>,
                has_dependency: <Boolean>
        },
        answer_yes:
        {
                answer_id: <Number>,
                has_dependency: <Boolean>
        }
}]
```

10.7.5 GET

```
Pfad: /question/:id
Request:
{}

Response:
{
        question_id: <Number>,
        name: <String>,
        is_symptom: <Boolean>,
        descriptions:
        [{
                description_id: <Number>,
                language_id: <Number>,
                language_name: <String>,
                description: <String>
        }],
        answer_no:
        {
                answer_id: <Number>,
                has_dependency: <Boolean>
        },
        answer_yes:
        {
                answer_id: <Number>,
                has_dependency: <Boolean>
        }
}
```

10.8 RunServlet

10.8.1 PUT

```
Pfad: /run/user/:id
Request:
{
        answer_id: <Number>
}

Response:
{}
```

10.8.2 DELETE

```
Pfad: /run/user/:id
Request:
{}

Response:
{}
```

10.8.3 GET

```
Pfad: /run/user/:id
Request:
{}

Response:
{
        diagnosis:
        {
                diagnosis_id: <Number>,
                name: <String>,
                descriptions:
                [
                        (Siehe DiagnosenServlet)
                ],
                perfect_diagnosis:
                [
                        (siehe DiagnosenServlet)
                ]
        },
        action_suggestions:
        [
                (siehe ActionSuggestionServlet)
        ]
}
```

10.9 SettingServlet

10.9.1 PUT

```
Pfad: /setting
Request:
{
        setting_id: <Number>,
        name: <String>,
        value: <Number>
}

Response:
{}
```

10.9.2 GET (All)

```
Pfad: /setting
Request:
{}

Response:
[{
                setting_id: <Number>,
                name: <String>,
                value: <Number>
}]
```

10.9.3 GET

```
Pfad: /setting/:id
Request:
{}

Response:
{
        setting_id: <Number>,
        name: <String>,
        value: <Number>
}
```

10.10 SyndromServlet

10.10.1 POST

```
Pfad: /syndrom
Request:
{}

Response:
{
        syndrom_id: <Number>,
        symptoms:
        [{
                answer_id: <Number>
        }]
}
```

10.10.2 PUT

```
Pfad: /syndrom
Request:
{
        syndrom_id: <Number>,
        name: <String>,
        symptoms:
        [{
                answer_id: <Number>
        }]
}

Response:
{}
```

10.10.3 DELETE

```
Pfad: /syndrom/:id
Request:
{}

Response:
{}
```

10.10.4 GET (All)

```
Pfad: /syndrom/
Request:
{}

Response:
[{
        syndrom_id: <Number>,
        name: <String>,
        symptoms:
        [{
                answer_id: <Number>
        }]
}]
```

10.10.5 GET

```
Pfad: /syndrom/:id
Request:
{}

Response:
 {
        syndrom_id: <Number>,
        name: <String>,
        symptoms:
        [{
                answer_id: <Number>
        }]
}
```

10.11 TestRunServlet

10.11.1 GET (All)

```
Pfad: /testrun/user/:id
Request:
{}

Response:
{
        diagnoses:
        [
                (siehe DiagnosisServlet)
        ],
        action_suggestions:
        [
                (siehe ActionSuggestionServlet)
        ]
}
```

10.12 UserServlet

10.12.1 POST

```
Pfad: /user
Request:
{}

Response:
{
        user_id: <Number>,
        name: <String>,
        password: <String>,
        history_id: <Number>
}
```

10.12.2 PUT

```
Pfad: /user
Request:
{
        user_id: <Number>,
        name: <String>,
        password: <String>,
        history_id: <Number>
}

Response:
{}
```

10.12.3 DELETE

```
Pfad: /user/:Id
Request:
{}

Response:
{}
```

10.12.4 GET (All)

```
Pfad: /user
Request:
{}

Response:
[{
        user_id: <Number>,
        name: <String>,
        password: <String>,
        history_id: <Number>
}]
```

10.12.5 GET

```
Pfad: /user/:Id
Request:
{}

Response:
{
        user_id: <Number>,
        name: <String>,
        password: <String>,
        history_id: <Number>
}
```

10.13 ActionSuggestionDescriptionServlet

10.13.1 POST

```
Pfad: /actionSuggestionDescription
Request:
{}

Response:
{
        description_id: <Number>,
        language_id: <Number>,
        action_suggestion_id: <Number>,
        description: <String>
}
```

10.13.2 PUT

```
Pfad: /actionSuggestionDescription
Request:
{
        description_id: <Number>,
        language_id: <Number>,
        action_suggestion_id: <Number>,
        description: <String>
}

Response:
{}
```

10.13.3 DELETE

```
Pfad: /actionSuggestionDescription/:Id
Request:
{}

Response:
{}
```

10.13.4 GET (All)

```
Pfad: /actionSuggestionDescription
Request:
{}

Response:
[{
        description_id: <Number>,
        language_id: <Number>,
        action_suggestion_id: <Number>,
        description: <String>
}]
```

10.13.5 GET

```
Pfad: /actionSuggestionDescription/:Id
Request:
{}

Response:
{
        description_id: <Number>,
        language_id: <Number>,
        action_suggestion_id: <Number>,
        description: <String>
}
```

10.14 DiagnosisDescriptionServlet

10.14.1 POST

```
Pfad: /diagnosisDescription
Request:
{}

Response:
{
        description_id: <Number>,
        language_id: <Number>,
        diagnosis_id: <Number>,
        description: <String>
}
```

10.14.2 PUT

```
Pfad: /diagnosisDescription
Request:
{
        description_id: <Number>,
        language_id: <Number>,
        diagnosis_id: <Number>,
        description: <String>
}

Response:
{}
```

10.14.3 DELETE

```
Pfad: /diagnosisDescription/:Id
Request:
{}

Response:
{}
```

10.14.4 GET (All)

```
Pfad: /diagnosisDescription
Request:
{}

Response:
[{
        description_id: <Number>,
        language_id: <Number>,
        diagnosis_id: <Number>,
        description: <String>
}]
```

10.14.5 GET

```
Pfad: /diagnosisDescription/:Id
Request:
{}

Response:
{
        description_id: <Number>,
        language_id: <Number>,
        diagnosis_id: <Number>,
        description: <String>
}
```

10.15 QuestionDescriptionServlet

10.15.1 POST

```
Pfad: /questionDescription
Request:
{}

Response:
{
        description_id: <Number>,
        language_id: <Number>,
        question_id: <Number>,
        description: <String>
}
```

10.15.2 PUT

```
Pfad: /questionDescription
Request:
{
        description_id: <Number>,
        language_id: <Number>,
        question_id: <Number>,
        description: <String>
}

Response:
{}
```

10.15.3 DELETE

```
Pfad: /questionDescription/:Id
Request:
{}

Response:
{}
```

10.15.4 GET (All)

```
Pfad: /questionDescription
Request:
{}

Response:
[{
        description_id: <Number>,
        language_id: <Number>,
        question_id: <Number>,
        description: <String>
}]
```

10.15.5 GET

```
Pfad: /questionDescription/:Id
Request:
{}

Response:
{
        description_id: <Number>,
        language_id: <Number>,
        question_id: <Number>,
        description: <String>
}
```

10.16 AnswerToActionSuggestionScoreDistributionServlet

10.16.1 POST

```
Pfad: /answerToActionSuggestionScoreDistribution
Request:
{}

Response:
{
        distribution_id: <Number>,
        score: <Number>,
        action_suggestion_id: <Number>,
        answer_id: <Number>
}
```

10.16.2 PUT

```
Pfad: /answerToActionSuggestionScoreDistribution
Request:
{
        distribution_id: <Number>,
        score: <Number>,
        action_suggestion_id: <Number>,
        answer_id: <Number>
}

Response:
{}
```

10.16.3 DELETE

```
Pfad: /answerToActionSuggestionScoreDistribution/:Id
Request:
{}

Response:
{}
```

10.16.4 GET (All)

```
Pfad: /answerToActionSuggestionScoreDistribution
Request:
{}

Response:
[{
        distribution_id: <Number>,
        score: <Number>,
        action_suggestion_id: <Number>,
        answer_id: <Number>
}]
```

10.16.5 GET

```
Pfad: /answerToActionSuggestionScoreDistribution/:Id
Request:
{}

Response:
{
        distribution_id: <Number>,
        score: <Number>,
        action_suggestion_id: <Number>,
        answer_id: <Number>
}
```

10.17 AnswerToDiagnosisScoreDistributionServlet

10.17.1 POST

```
Pfad: /answerToDiagnosisScoreDistribution
Request:
{}

Response:
{
        distribution_id: <Number>,
        score: <Number>,
        diagnosis_id: <Number>,
        answer_id: <Number>
}
```

10.17.2 PUT

```
Pfad: /answerToDiagnosisScoreDistribution
Request:
{
        distribution_id: <Number>,
        score: <Number>,
        diagnosis_id: <Number>,
        answer_id: <Number>
}

Response:
{}
```

10.17.3 DELETE

```
Pfad: /answerToDiagnosisScoreDistribution/:Id
Request:
{}

Response:
{}
```

10.17.4 GET (All)

```
Pfad: /answerToDiagnosisScoreDistribution
Request:
{}

Response:
[{
        distribution_id: <Number>,
        score: <Number>,
        diagnosis_id: <Number>,
        answer_id: <Number>
}]
```

10.17.5 GET

```
Pfad: /answerToDiagnosisScoreDistribution/:Id
Request:
{}

Response:
{
        distribution_id: <Number>,
        score: <Number>,
        diagnosis_id: <Number>,
        answer_id: <Number>
}
```

10.18 SyndromToActionSuggestionScoreDistributionServlet

10.18.1 POST

```
Pfad: /syndromToActionSuggestionScoreDistribution
Request:
{}

Response:
{
        distribution_id: <Number>,
        score: <Number>,
        action_suggestion_id: <Number>,
        syndrom_id: <Number>
}
```

10.18.2 PUT

```
Pfad: /syndromToActionSuggestionScoreDistribution
Request:
{
        distribution_id: <Number>,
        score: <Number>,
        action_suggestion_id: <Number>,
        syndrom_id: <Number>
}

Response:
{}
```

10.18.3 DELETE

```
Pfad: /syndromToActionSuggestionScoreDistribution/:Id
Request:
{}

Response:
{}
```

10.18.4 GET (All)

```
Pfad: /syndromToActionSuggestionScoreDistribution
Request:
{}

Response:
[{
        distribution_id: <Number>,
        score: <Number>,
        action_suggestion_id: <Number>,
        syndrom_id: <Number>
}]
```

10.18.5 GET

```
Pfad: /syndromToActionSuggestionScoreDistribution/:Id
Request:
{}

Response:
{
        distribution_id: <Number>,
        score: <Number>,
        action_suggestion_id: <Number>,
        syndrom_id: <Number>
}
```

10.19 Anleitung zur Einrichtung der Entwicklungsumgebung

1. Heroku Account erstellen
2. Industriepartner muss diesen Account als Collaborator dem App zuweisen
3. Heroku Toolbelt installieren
4. Git klonen gemäss Heroku → Personal Apps → pocketdoc → Code
5. JRE 8 installieren
6. Intellij IDEA installieren
7. PostgreSQL installieren
8. Apache Tomcat installieren
9. Apache Maven installieren
10. Intellij einrichten → Maven, Tomcat
11. hibernate.cfg → es gibt eine lokale und eine remote Version
12. Im geklonten Projektordner hat es einen „db“-Ordner in diesem befinden sich die DB-Scripts welche in die PSQL datenbank eingespielt werden müssen.

11 Installationsanleitung

1. Neuste Version von Postgresql installieren (Version 9.3.5)
   `http://www.enterprisedb.com/products-services-training/pgdownload#windows`
   ACHTUNG: beim Installieren wird man nach einem Passwort aufgefordert dieses muss man sich unbedingt merken. Den Rest der Installation kann man mittels klicken auf Next vornehmen.

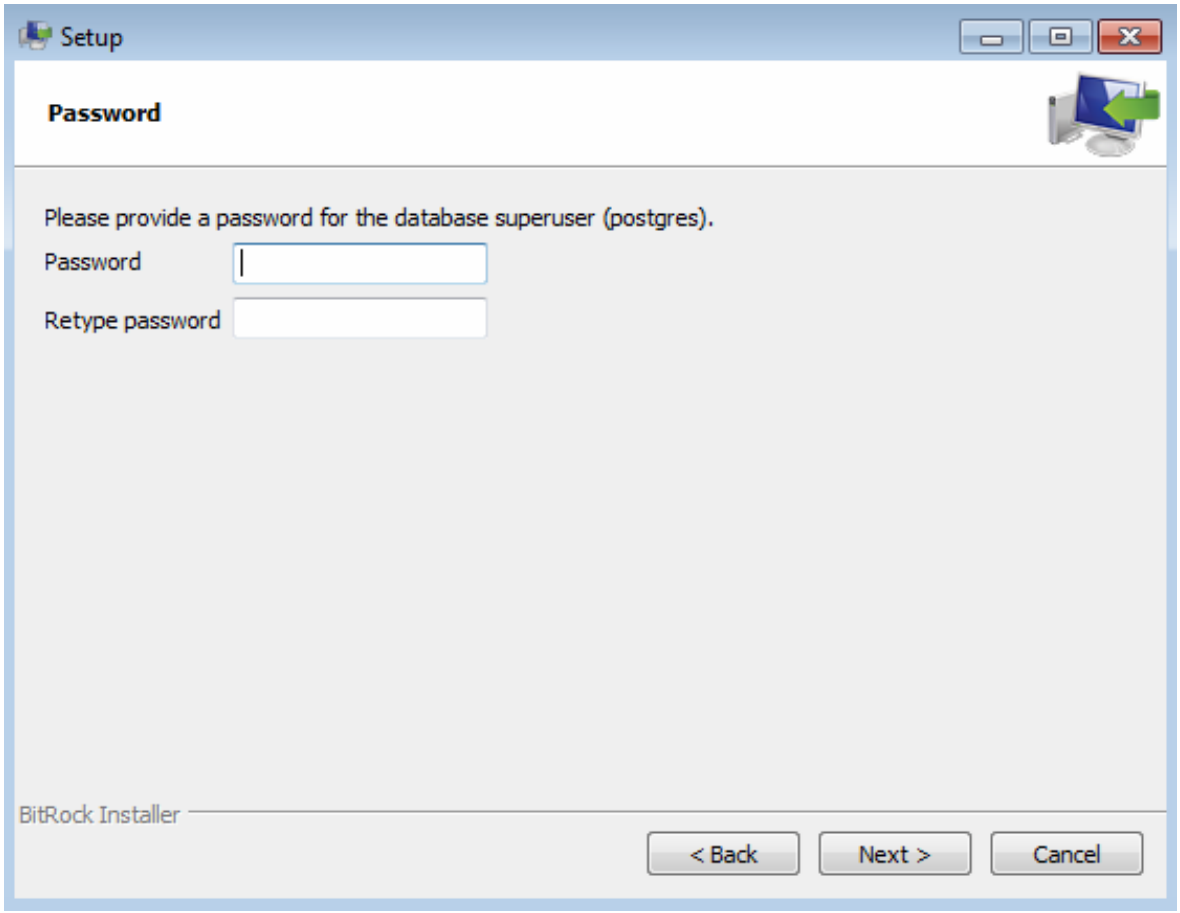


Abbildung 40: PostgreSQL Installation

2. Nun startet man die SQL-Shell (psql) einfach unter Start nach „psql“ suchen und starten. Dann muss man mit Enter bestätigen bis man zur Eingabe des Passworts kommt. Dort muss man das während der Installation eingegebene Passwort einfügen.

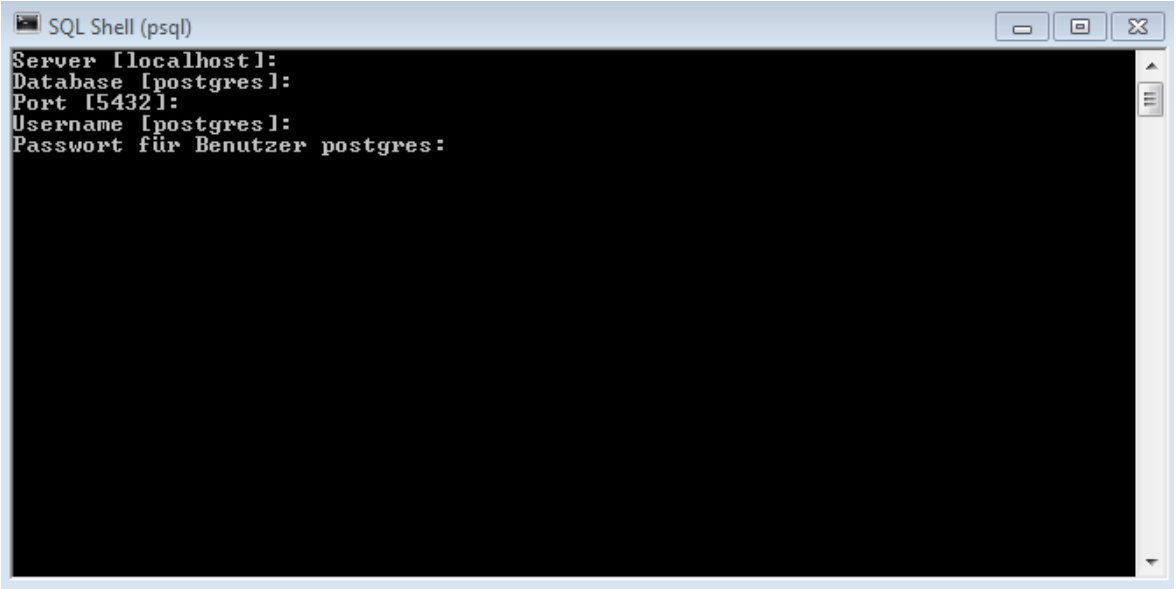


Abbildung 41: Postgre Shell

3. Nun folgende Befehle eingeben dabei auf die Unterscheidung “ und ’ achten.

```
1 CREATE DATABASE "pocketdoc" WITH ENCODING='UTF8';
2 CREATE USER pocketdoc WITH PASSWORD 'pocketdocpassword';
3 GRANT ALL ON DATABASE pocketdoc to pocketdoc;
```

4. Nun muss die Datenbank noch erstellt werden. Dafür liegt ein kleines Skript bereit. Dieses geht davon aus das die Befehle richtig ausgeführt wurden und die Namen übereinstimmen. Ausserdem muss sich die Postgresql Installation im Verzeichnis „C:\Program Files\PostgreSQL\9.3\bin“ befinden. Im „pocketdoc.zip“ hat es einen Ordner db dort drin auf „create database.bat“ klicken um die Datenbank einzurichten. Falls der Installationspfad nicht derselbe ist so muss im „create database.bat“ der Pfad mit einem Editor angepasst werden. Nicht erschrecken die ersten paar Befehle sollen Fehlschlagen, da die alten Tabellen gelöscht werden. (Beim ersten mal Ausführen gibts noch keine alten Tabellen aber bei einem Reset der datenbank wird dies benötigt)
5. Jetzt muss der Servlet Container „Tomcat 8“ installiert werden. Diesen gibts hier zum Download `http://tomcat.apache.org/download-80.cgi`. Am besten 64-bit zip version. Das zip einfach entpacken und den Pfad dorthin merken.
6. Im nun entpackten Tomcat Pfad hat es einen Ordner webapps dort drin muss man ein Verzeichnis mit dem Namen „pocketdoc“ anlegen. Und das in der Beilage angehängte „webapp.zip“ dort hineinverschieben und entpacken. Im Folgenden Bild sieht man gut den Pfad und wie die entpackten Daten aussehen sollten.

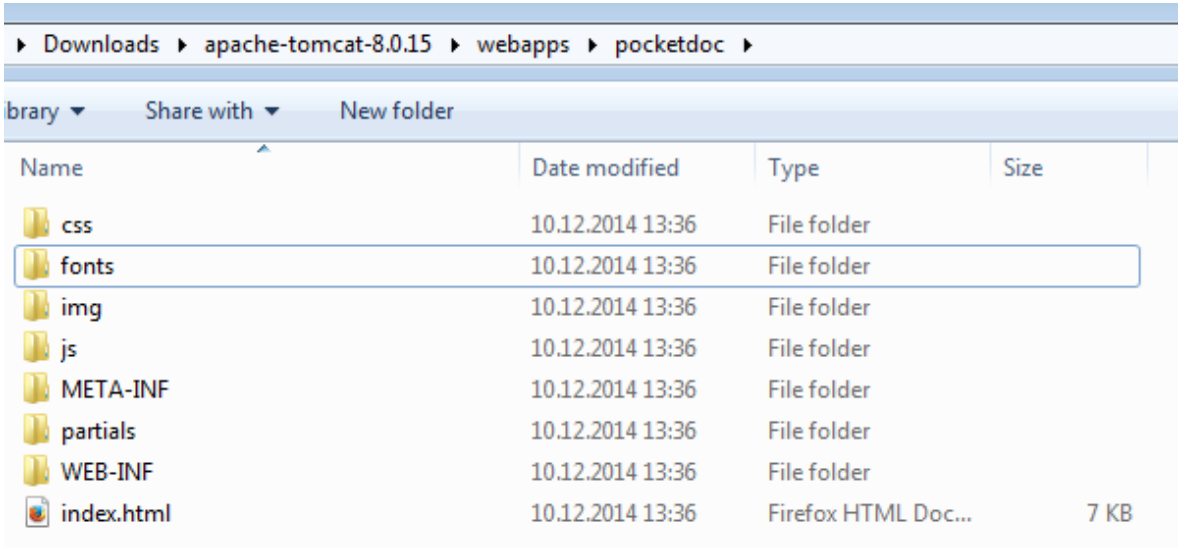


Abbildung 42: Tomcat: Verzeichnisstruktur

7. Nun muss die „hibernate.cfg.xml“ Datei, welche im „pocketdoc.zip“ enthalten ist kopiert werden und die „hibernate.cfg.xml“ im Ordner „WEB-INF/classes“ überschrieben werden.
8. Als weiterer Schritt muss die Applikation im Tomcat Root laufen deshalb muss dies im „server.xml“ von der Tomcat config angepasst werden. Dazu kann man einfach das im „pocketdoc.zip“ enthaltene „server.xml“ kopieren und das im „(TOMCAT-Verzeichnis)/conf“ Verzeichnis überschrieben werden.
9. Um den Tomcat nun laufen zu lassen brauchet es noch ein Java Runtiime Environment, unbedingt das neuste (JRE8) welches hier zum Download steht. `http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html`
10. Nach dem das JRE installiert ist sollte das JRE unter „C:\Program Files\Java\jre8“ installiert sein. Diesen Pfad sollte man schnell überprüfen und den Pfad in die Zwischenablage kopieren. Nun muss man eben die JRE_HOME Environment Variable setzten. Dazu kann man unter Start „Environment“ eingeben. Dannach auf „Edit environment variables for your account“ klicken. (Deutsch: „Umgebungsvariablen“, „Umgebungsvariablen für dieses Konto bearbeiten“)

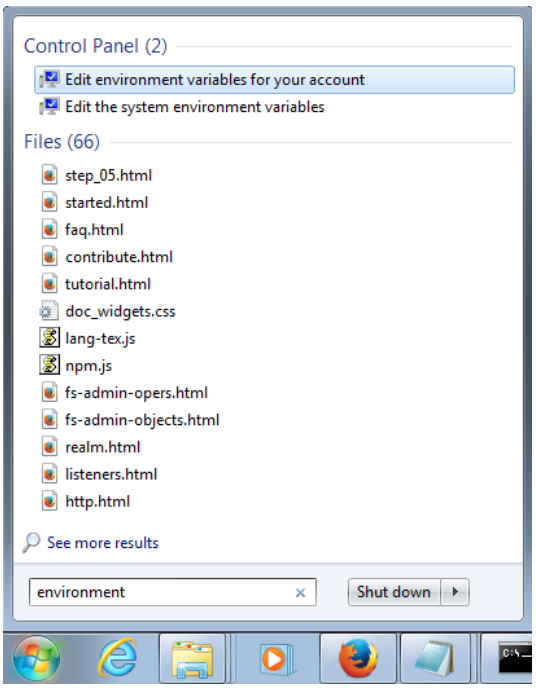


Abbildung 43: Start: Suche nach Umgebungsvariablen

11. Nun sollte sich ein neues Fenster geöffnet haben dort auf „New“ klicken. Dannach wie das folgende Bild anzeigt die Inhalte setzten. Der Angezeigte Pfad ist der in die Zwischenablage kopierte Pfad vom letzten Schritt. (Anstatt jre7, jre8)

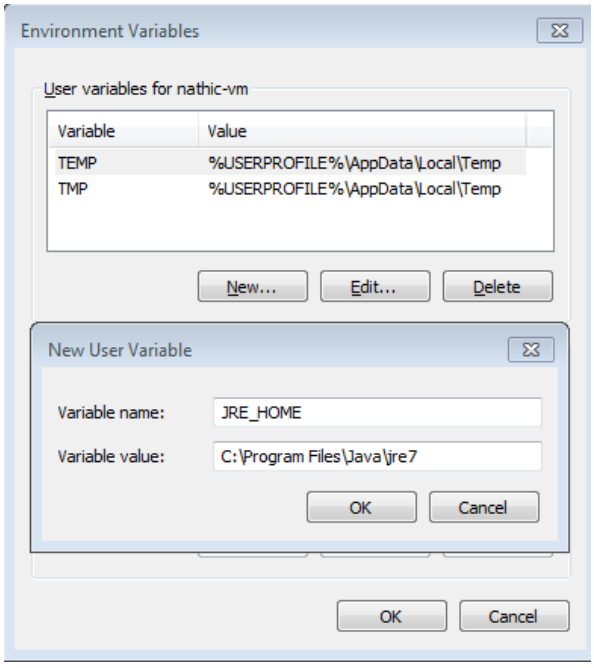


Abbildung 44: Umgebungsvariable: JRE_HOME

12. Nun im Tomcat Verzeichnis den Ordner bin öffnen und auf „startup.bat„ klicken.
13. Nun kann man via `http://localhost:8080` darauf zugreifen.
14. Den Tomcat kann man mit dem „shutdown.bat“ wieder beenden.