Hochschule für Angewandte Wissenschaften Hamburg
*Hamburg University of Applied Sciences*

# Bachelorarbeit

**Dr. Sebastian Diedrich**

## Konzeption und prototypische Realisierung einer Microservice Architektur zur graph-basierten Speicherung von medizinischen Gesundheits- und Abrechnungsdaten zwecks Analyse und Auswertung

*Fakultät Technik und Informatik*
*Studiendepartment Informatik*

*Faculty of Engineering and Computer Science*
*Department of Computer Science*

Dr. Sebastian Diedrich

# Konzeption und prototypische Realisierung einer Microservice Architektur zur graph-basierten Speicherung von medizinischen Gesundheits- und Abrechnungsdaten zwecks Analyse und Auswertung

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr.-Ing. Olaf Zukunft

Eingereicht am: 15. März 2018

**Dr. Sebastian Diedrich**

**Thema der Arbeit**
Konzeption und prototypische Realisierung einer Microservice Architektur zur graph-basierten Speicherung von medizinischen Gesundheits- und Abrechnungsdaten zwecks Analyse und Auswertung

**Stichworte**
Microservice, E-Health, Docker, Neo4J, EPA, graph-basiert, Rechnungsverfolgung, PKV

**Kurzzusammenfassung**
Das Ziel dieser Bachelorarbeit war die prototypische Konzeption und Realisierung einer Applikation, die es Patienten ermöglicht, ihre elektronischen Gesundheits- und Abrechnungsdaten zu verwalten. Die Microservice Architektur diente dabei als Grundlage, die Speicherung der Daten erfolgt graph-basiert in einer *Neo4j* Datenbank.
Im Laufe der Arbeit werden die Vor- und Nachteile des gewählten Designs erläutert und der entstandene Prototyp mit anderen Gesundheits-Apps verglichen. Die Möglichkeit der graph-basierten Präsentation der Daten stellt dabei im Vergleich zu bestehenden Produkten einen interessanten neuen Aspekt dar.

**Dr. Sebastian Diedrich**

**Title of the paper**
Conception and Prototypical Realization of a Microservice Architecture for graph-based Storage of medical Health- and Invoice Data for Analysis and Evaluation

**Keywords**
Microservice, E-Health, Docker, Neo4J, Personal Health Record, graph-based, Invoice Tracking

**Abstract**
The aim of this bachelor thesis was the prototypical conception and realization of an application that allows patients to manage their electronic health and invoice data. The microservice architecture served as the basis, the storage of the data is graph-based in a Neo4j Database.
In the course of this thesis the advantages and disadvantages of the selected design are explained and the resulting prototype will be compared with other health apps. The possibility of graph-based presentation of the data represents an interesting new aspect compared to existing products.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# 1 Einleitung

Im 19. Jahrhundert war die Welt noch größtenteils unerforscht und Gelehrte wie Charles Darwin und Alexander von Humboldt sammelten während ihrer Forschungsreisen unzählige Daten über die Natur, um Rückschlüsse auf deren Entstehung und Fortbestand ziehen zu können.
Zwei Jahrhunderte später liegt bei vielen Menschen der Fokus beim Sammeln von naturwissenschaftlichen Daten eher auf der eigenen Person. Zahlreiche Geräte erlauben es unzählige Gesundheitsparameter zu sammeln. Die moderne Medizin produziert jährlich Billionen von Datensätzen, die fast ausschließlich elektronisch gespeichert werden und damit für eine spätere Datenverarbeitung zur Verfügung stehen.
Werden diese Daten kombiniert, ergibt sich ein sehr genaues Bild vom aktuellen Gesundheitszustand eines Menschen. Diese Entwicklung wird in unserer Gesellschaft sehr kontrovers diskutiert. Befürworteter vernetzten sich in Bewegungen wie *Quantified Self*, in der Hoffnung das eigene dauerhafte Vermessen und Teilen dieser Daten führe zu einem besseren und gesünderen Leben. Zahlreiche Software-Programme ermöglichen die langfristige Speicherung und Verwaltung dieser Daten. Zusätzlich kann der Nutzer noch weitere medizinisch-relevante Gesundheitsdaten wie z.B. Diagnosen, Medikamentenpläne oder Ernährungsgewohnheiten in das Programm einpflegen.
Genau dieser Punkt verursacht bei den Gegnern der Bewegung große Sorgen, da die Speicherung dieser Gesundheitsdaten häufig auf zentralen Servern großer Unternehmen stattfindet. Was genau mit diesen Daten passiert, ist ungewiss und möglicherweise nicht immer im Sinne der Nutzer.
Begriffe wie *Big Data* und das damit häufig verbundene *Data Mining* sind in der aktuellen IT- und Geschäftswelt allgegenwärtig. Inwieweit die Auswertung der begehrten Daten über den gesundheitlichen Zustand einer Person oder ganzer Bevölkerungsgruppen möglich ist, hängt allerdings stark vom Verhalten der Menschen und Institutionen ab, die diese Daten generieren und speichern. Die Entscheidung welche Software für diesen Zweck genutzt wird, bestimmt maßgeblich die Möglichkeiten, ob und wie diese Daten gebraucht bzw. missbraucht werden.

## 1.1 Motivation

Das Gesundheitssystem in Deutschland ist eines der besten und teuersten der Welt. Jeder Bürger hat die freie Arztwahl, ab einem bestimmten Einkommen bzw. Anstellungsverhältnis auch zwischen einer gesetzlichen (GKV) oder privaten Krankenversicherung (PKV). Als Selbständiger oder als Beamter ist eine Mitgliedschaft in einer PKV verpflichtend. So werden über Jahre viele Diagnosen, Untersuchungen, Medikamentenverschreibungen und Labordaten durch zahlreiche medizinische Einrichtungen (Fachärzte, Krankenhäuser, Ambulanzen etc.) generiert.
Diese Daten liegen dann meist lokal bei den Einrichtungen selbst, oder werden von den Patienten in einem großen Aktenordner abgelegt. Verbindungen zwischen Beschwerden, Diagnosen und erhobenen Gesundheitsdaten sind ab einer bestimmten Komplexität nur noch schwer nachzuvollziehen.
Hinzu kommt, dass privatversicherte Personen sich um die Rückerstattung der entstandenen Kosten durch die PKV / Beihilfe selber kümmern müssen. Gerade bei der Kostenerstattung durch die Beihilfe kann es zu Verzögerungen kommen, da häufig zunächst ein Mindestbetrag für die Erstattung erreicht werden muss, bevor ein Antrag gestellt werden kann.
Ich selber habe während meiner ärztlichen Tätigkeit diese Erfahrung gemacht: Besonders chronisch kranke Patienten haben mehrere Aktenordner voller medizinischer Daten, deren Durchsicht Stunden in Anspruch nimmt - trotzdem fällt es manchmal schwer, Zusammenhänge zu erfassen.
Eine Speicherung all dieser medizinischen (Rechnungs-)Daten in einer Graph-Datenbank könnte es ermöglichen, Zusammenhänge besser zu erkennen, Doppeluntersuchungen zu vermeiden und ggf. auch weitere hilfreiche Analysen und Trackings bezüglich Kosten (Erstattung der Rechnungen durch Private Krankenversicherungen / Beihilfe) ermöglichen.

## 1.2 Zielsetzung

Ziel dieser Arbeit ist es einen lauffähigen Software-Prototypen zu erstellen, der das Speichern von medizinischen Daten und die damit verbundenen Abrechnungskosten ermöglicht. Grundlage ist eine Microservice Architektur, das Deployment der einzelnen Services erfolgt mittels Docker-Container. Durch den Einsatz einer Graph-Datenbank können später die Zusammenhänge mittels eines Standard-Browsers graphisch dargestellt werden.
Zusätzlich sollen andere Softwareprodukte zum Sammeln von medizinischen Daten mit diesem Prototypen verglichen und bezüglich bestimmter Kriterien (Nutzen, Bedienkomfort, Datenschutz etc.) analysiert werden.

Das Speichern und Verarbeiten von Gesundheitsparametern mittels „Fitness-Gadgets“ soll nicht im Fokus dieser Arbeit stehen.

## 1.3 Aufbau der Arbeit

### Kapitel 1: Einleitung

Im ersten Kapitel werden Motivation und Zielsetzung für diese Arbeit erläutert. Es wird kurz die Aktualität des gewählten Themas dargelegt und dass dieses für viele Unternehmen einen wichtigen Zukunftsmarkt darstellt. Auch der persönliche Bezug zum Thema und die möglichen Gefahren beim falschen Umgang mit sensiblen Gesundheitsdaten werden beschrieben.

### Kapitel 2: Grundlagen

Im Grundlagenkapitel werden wichtige Begriffe und Technologien erläutert. Im Fokus steht dabei die gewählte Architekturform und aktuelle Möglichkeiten der Virtualisierung. Zusätzlich werden grundlegende Modelle zur Datenspeicherung aufgezeigt.

### Kapitel 3: Anforderungsanalyse

In diesem Kapitel wird der Anwendungsschwerpunkt definiert und welche Benutzergruppen mit dem Programm interagieren. Zusätzlich werden verschiedene Anwendungsfälle beschrieben und in der Spezifikation werden Stakeholder, das FDM und verschiedene Wireframes erläutert.

### Kapitel 4: Konzeption

Das Kapitel Konzeption erläutert die Architektur des Prototypens anhand von verschiedenen Sichten. Zusätzlich werden alle *Use Cases* mittels Sequenzdiagramm genauer erläutert und Design-Entscheidungen diskutiert.

### Kapitel 5: Realisierung

Dieses Kapitel beschäftigt sich mit der letztendlichen Umsetzung, der vorher skizzierten Architektur. Es beinhaltet zahlreiche Code-Beispiele und technische Erläuterungen der genutzten Technologien und Sprachen.

### Kapitel 6: Evaluierung

Das vorletzte Kapitel beurteilt, in wieweit die vorher erarbeitete Spezifikation erfüllt wurde und welche Bedeutung das Testen bei der Entwicklung des Prototypens hatte.

### Kapitel 7: Fazit und Ausblick

Abschließend wird noch einmal die gesamte Arbeit zusammenfassend bewertet und ein kurzer Ausblick für zukünftige Erweiterungen bzw. Verbesserungen des entstandenen Prototypens gegeben.

# 2 Grundlagen

## 2.1 Microservices vs. Monolith

Bei der Softwareentwicklung stehen zahlreiche Entwurfsprinzipien und Architekturstile zur Verfügung. In diesem Abschnitt soll vor allem der Architekturstil *Microservices* mit einer monolithischen Struktur verglichen werden.
Bereits vorab kann man sagen, dass jeder Architekturstil seine Daseinsberechtigung hat, auch wenn in den letzten Jahren zunehmend die lose gekoppelte Softwareentwicklung, die die DNA der *Microservices* darstellt, zunehmend Bedeutung im Bereich Unternehmenssoftware (enterprise software) gewinnt Knoche (2016).
Der hauptsächliche Unterschied beider Stile liegt darin, einen Geschäftsprozess mittels *einer* fest zusammenhängenden Applikation (Monolith) umzusetzen oder dieses durch eine *Suite of small Services* zu erreichen. Der Endanwender selber merkt dabei normalerweise nicht, welche Grundstruktur die genutzte Software inne hat. Für die weitere Entwicklung und Wartbarkeit der Software, hat diese Entscheidung allerdings massive Auswirkungen.
Geschäftsprozesse unterliegen in unserer heutigen Zeit einem immer schneller werdenden Wandel. Beharrt ein Unternehmen zu lange auf einem Prozess, der sich vom Kundenwunsch unterscheidet, kann dieses zu spürbaren Verlusten an Marktanteilen führen. Als Beispiele können hier z.B. Nokia oder der Einzelhandel genannt werden, die zu lange gewartet haben, auf neue Bedürfnisse der Kunden zu reagieren.
Ähnlich spürbare Prozesse passieren heute im Bereich des *Cloud Computings*. Dabei werden Hardware-Infrastrukturen, Platformen und Softwarelösungen in der *Cloud* bereitgestellt. Wo früher noch eigene Serverräume nötig waren, reicht heute ein schneller Internetzugang, um zahlreiche Geschäftsprozesse mittels IT abbilden und umsetzen zu können Rad (2017).
Einen Geschäftsprozess durch verschiedene, ineinander greifende, aber doch unabhängig von einander laufende Mircoservices umzusetzen, stellt ganz neue Herausforderungen an Entwicklung und Betreiber der Software. Ein wichtiges Charakteristikum bei der Entwicklung von Microservices stellt die Organisation an Geschäftsgrenzen dar. Dadurch werden häufig der Einsatz von „cross-functional“ Teams nötig, da jeder Service seine eigenen Schnittstellen,

Logik und Speicherung vorhalten muss. Auch der Betrieb wird zumeist durch die Teams selbst gewährleistet, nach dem Motto: „You build it, you run it!“. Im Kapitel Konzeption werden weitere Charakteristika dieses Architekturstiles erläutert.

Monolithische Strukturen sind in der Entwicklung und im Betrieb weniger komplex, da nicht viele kleine Rädchen ineinander greifen müssen und auch die Performanz kann deutlich höher sein, da Methodenaufrufe „In-Process“ passieren und keine Verzögerungen durch „Remote Procedure Calls“ auftreten.
Unternehmen sind daher auf erfahrendere Entwickler angewiesen oder müssen zuvor aufwendige Schulungen durchführen, bevor sie die Vorteile der Microservice-Architektur für sich nutzen können Takai (2017).

## 2.2 Containerisierung vs. Virtueller Server

Virtualisierung bedeutet, dass man die physikalische Hardware vom Betriebssystem abstrahiert. Die Software hierzu wird „Hypervisor“ oder „Virtual Machine Monitor“ genannt. Dieses ermöglicht es, viele unterschiedliche Betriebssysteme auf einem Server auszuführen.
Doch einen virtuellen Server zu starten benötigt einige Zeit und Ressourcen. Die Containerisierung bietet hier eine sinnvolle Alternative. Als Paul Menage 2006 den Linux Kernel um die Technologie *control groups* erweiterte, war noch nicht abzusehen, dass dieses in den nächsten Jahren einen neuen Hype namens *Containerbasierte Virtualisierung* auslösen sollte Julian (2016).
Der größte Unterschied zwischen diesen Technologien liegt darin, dass bei der Containerisierung keine Hardware-Emulation stattfindet. Dieses hat den Nachteil, dass alle laufenden Container (Prozesse) sich eine Hardware-Basis und ein Betriebssystem teilen müssen. Gleichzeitig ermöglicht es aber, hunderte bis tausende isolierte Prozesse auf einem System laufen zu lassen. Würde man das gleiche mittels Virtuellen Servern versuchen, wäre der Overhead der Hardware-Emulation extrem groß und die Performanz dadurch stark beeinträchtigt. In der Abbildung 2.1 werden noch einmal die Charakteristika beider Technologien gegenüber gestellt.
Da Microservices zur Umsetzung der prototypische Realisierung dienen, kommen verschiedene Varianten des Betriebs in Betracht. Neben den beiden eben genannten, wäre auch ein dedizierter Server als Lösung möglich. Dieser eher klassische Ansatz würde aber einige Einschränkungen bezüglich der Freiheit in Technologieentscheidungen (Dateisystem, Programmiersprachen

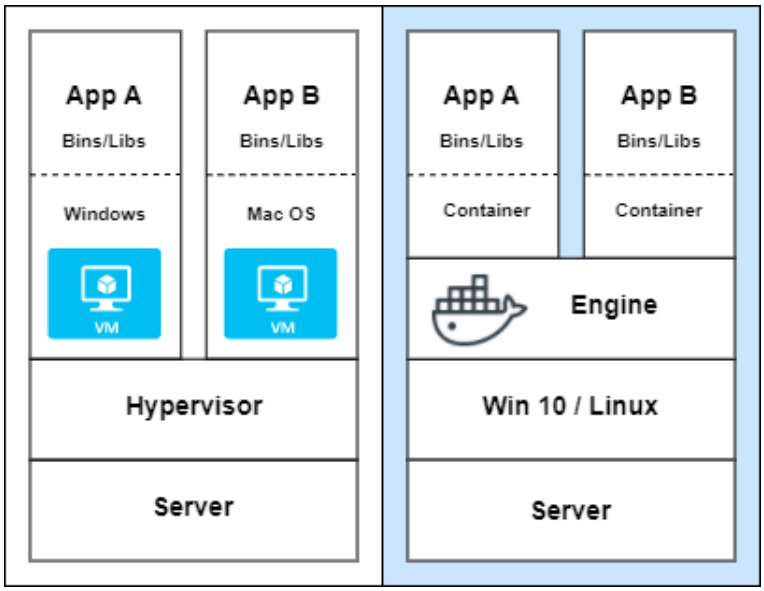


Abbildung 2.1: Virtuelle Machine versus Containerisierung mittels Docker Engine

etc.) bedeuten. Hinzu kommt, dass die Virtualisierung bereits während der Phase der Softwareentwicklung einen deutlichen Mehrwert in Bezug auf agile Methodik bieten kann (z.B. tägliche Integrationstests). Dabei ist der Entwickler völlig unabhängig von der Hardware- und Softwareplattform, auf der er entwickelt. Seit Windows 10 wird auch außerhalb der Linux/UNIX Welt die *Containerbasierte Virtualisierung* nativ unterstützt[1]. Zuvor war dieses nur mittels einer virtuellen Maschine möglich, die ihrerseits einige Ressourcen beanspruchte.

## 2.3 Docker

Die *Docker* Technologie ermöglicht es, Anwendungen in *Containern* auszuführen. Früher musste man z.B., um eine Webseite auf einem Webserver samt Datenhaltung zu testen die Programmpakete XAMPP bzw. LAMPP installieren. Im Anschluss mussten der darin beinhaltete Webserver und die Datenbank gestartet werden.
Mittels Docker Technologie kann diese Anwendung in einem Container gestartet werden, der bereits alle nötigen Komponenten (z.B. Webserver, DB etc.) enthält. Selbst das Betriebssystem kann individuell ausgewählt werden. Jeder Container besteht aus einem *Image*, welches selber aus mehreren Schichten (*Layers*) besteht. Jeder *Layer* ist schreibgeschützt und nur der Letzte, der sogenannte *Container-Layer*, ermöglicht es, Schreiboperationen durchzuführen. Die Abbil-

[1] https://www.docker.com/docker-windows

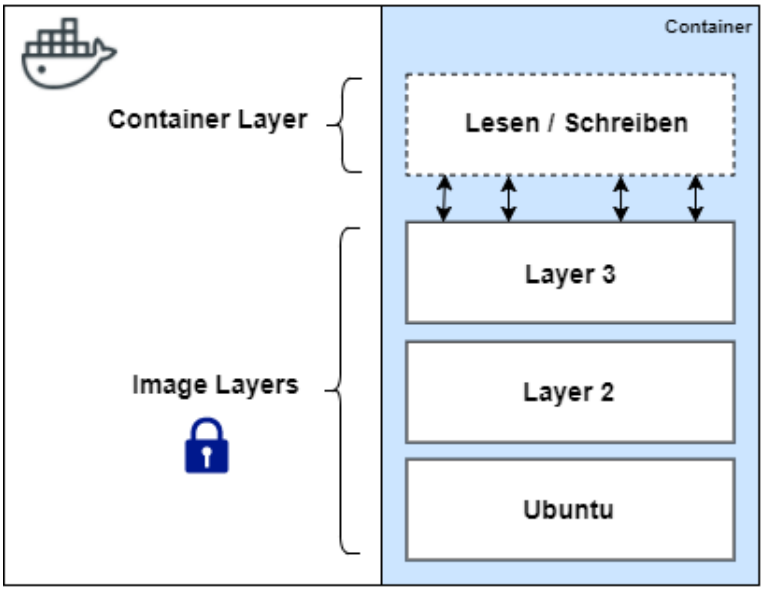


Abbildung 2.2: Docker Container basierend auf einem Ubuntu Image

dung 2.2 erläutern diesen Aufbau.

Bei der Installation von Docker werden mehrere Komponenten installiert. Die Hauptkomponente stellt dabei die *Docker Engine* da, die selber aus einem *Client* und einem *Deamon* besteht. Docker Images werden anhand eines *Dockerfiles* erstellt und können anschließend in einer *Registry* gespeichert werden, Docker bietet mit *Docker Hub* bereits eine solche. Das fertig gebaute Image lässt sich dann in wenigen Sekunden als Container starten.
Die ebenfalls enthaltende Komponente *Docker Compose*[2] ermöglicht es, anhand einer Konfigurationsdatei „docker-compose.yml“ alle benötigten Images zu bauen, Container zu starten und diese z.B. mit einem gemeinsamen Netzwerk zu verknüpfen. Viele manuelle Einzelschritte können so in einer Art Bash-Datei gebündelt werden.

## 2.4 Spring Boot

Das Framework Spring erleichtert die Entwicklung von Java-Enterprise-Anwendungen. Es beinhaltet zahlreiche Features wie *Dependency Injection* oder die grundlegende Unterstützung

[2] https://docs.docker.com/compose/

für *Java Database Connectivity* oder die *Java Persistence API*. *Spring Boot* ist eines von vielen Unterprojekten von Spring, die es dem Entwickler erleichtert, lauffähige, auf Spring basierende Anwendungen zu schreiben. Genauere Ausführungen erfolgen im Kapitel 4.

## 2.5 Datenbankmanagementsysteme

Ein Datenbanksystem besteht aus der Datenbank (DB), in der die Daten abgelegt werden, und dem Datenbankmanagementsystem (DBMS), dem Verwaltungsprogramm, das die Daten entsprechend der vorgegebenen Beschreibungen abspeichert, auffindet oder verändert.
Unterscheiden lassen sich dabei zwei grundsätzliche Modelle: Das „Relationale Modell“ und die sogenannten „NoSQL“-Datenbanksysteme.
Die relationalen Datenbanken basieren dabei auf Tabellen, wobei jede Zeile als Tupel bezeichnet wird und einen Datensatz repräsentiert. Bevor Daten gespeichert werden, wird in einem *Schema* festgelegt, wie ein solcher Datensatz aufgebaut ist. Um komplexere Strukturen in diesem Modell abspeichern zu können, kommen *ORM-Frameworks* wie z.B. Hibernate zum Einsatz. Transaktionen unterliegen den *ACID* Prinzipien und sind so besonders verlässlich und es wird eine hohe Datenkonsistenz erzielt.
Die „NoSQL-DBMS“ haben charakteristischerweise eher kein Schema, sind häufig Open-Source und *ACID* wird normalerweise nicht unterstützt. Nötig wurden diese Formen der DBMS vor allem durch das steigende Datenaufkommen und immer beliebter werdenden *Social Media* Anwendungen seit der Jahrtausendwende Elmasri (2017) . NoSQL-Datenbanken eignet sich gut für die horizontale Skalierung in einem Cluster. Dieses *Scaling Out* erhöht dabei allerdings auch die Komplexität. Die Daten werden nach einem der folgenden Kategorien organisiert:

1) *Key-value* basierend
2) *Document* basierend
3) *Column-family* basierend
4) *Graph* basierend

Im Zusammenhang mit NoSQL-Datenbanken sind auch Begriffe wie das *CAP-Theorem* und *BASE* von Bedeutung. Im Kern geht es dabei darum, dass aufgrund der typischen Eigenschaften, niemals eine hundertprozentige *Availability*, *Consistency* und *Partition Tolerance* gleichzeitig erzielt werden kann Redmond (2012).
Für diese Arbeit wurde die graph-basierte Datenbank *Neo4j* ausgewählt. Weitere Erläuterungen dazu erfolgen im Kapitel 4.

# 3 Anforderungsanalyse und Spezifikation

## 3.1 Analyse

Während der Analysephase werden die *Stakeholder* identifiziert und deren funktionale und nicht-funktionale Anforderungen mittels *User Stories* oder ähnlichen Dokumentationsmitteln des *Requirements-Engineering* erhoben Rupp (2007).

### 3.1.1 Anwendungsszenario

Die zu entwickelnde Anwendung soll als Schwerpunkt persönliche Gesundheitsdaten speichern und sowohl dem Patienten selbst, als auch anderen Personengruppen und Organisationen dabei unterstützen, aus diesen Informationen einen Mehrwert zu gewinnen. Zusätzlich soll die Software besonders PKV-Patienten dabei unterstützen, die aufwendige Abrechnung mit den verschiedenen medizinischen Einrichtungen zu erleichtern.
Konkrete Beispiele werden im Unterkapitel „Use-Cases“ erläutert.

### 3.1.2 Stakeholder

Als *Stakeholder* werden Personen oder Organisationen bezeichnet, die bei der Erstellung des Systems mitwirken, es beeinflussen oder am entwickelten System ein fachliches, technisches oder kommerzielles Interessen haben Starke (2015).
Konkret wurden die Interessen von drei *Stakholdern* herausgearbeitet:

**Patienten** Als graphische Benutzeroberfläche steht ein Webinterface (GUI) zur Verfügung, das sich je nach Endgerät (PC, Handy, Tablet) dynamisch in Form und Größe anpasst. Die medizinischen Gesundheits- und Abrechnungsdaten kann der Benutzer auf unterschiedliche Weise in das Programm einpflegen. Im günstigsten Fall kann dieses über einen automatischen Import der Daten passieren. Dabei verbindet sich die Anwendung mit einer Arztpraxis, einem Krankenhaus oder Labor und es werden die Daten verschlüsselt übertragen.
Sollten die dafür benötigten Schnittstellen fehlen, können alle relevanten Daten auch per Tastatur oder per Scan in das Programm eingepflegt werden. Eine integrierte OCR-Software

erkennt die relevanten Daten. Bevor die endgültige Übernahme der Daten erfolgt, kann der Benutzer noch Korrekturen und Ergänzungen vornehmen.
Prinzipiell werden zwei Szenarien unterschieden. Im „Analyse bzw. Tracking"-Bereich können medizinische Daten Eingesehen und Abrechnungsdaten verfolgt werden. Der „Import"-Bereich dient der Dateneingabe bzw. zum Anstoßen eines elektronischen Datenaustausches.

**Medizinisches Personal** Damit auch Ärzte und andere Berufsgruppen im Gesundheitssystem von den gespeicherten Daten profitieren können, gibt es eine eingeschränkte Zugriffsmöglichkeit. Der Patient kann zuvor ein individuelles Nutzerprofil anlegen und so Daten freigeben bzw. verbergen. Diese Freigaben können auch zeitlich begrenzt werden.

**Versicherungen** Entschließt sich ein Patient, die Krankenkasse zu wechseln oder eine zusätzliche Leistung abzuschließen, müssen häufig umfangreiche Fragebögen ausgefüllt werden. Der zukünftige Versicherungsnehmer läuft dabei ggf. Gefahr, versicherungsrelevante Daten nicht ordnungsmäßig anzugeben. Der Zugriff auf die medizinischen Daten durch das Verwaltungspersonal der Versicherung, kann dabei helfen, das Risikoprofil des Patienten objektiv zu erfassen.

### 3.1.3 User-Stories

Mittels *User-Stories* können funktionale Anforderungen der Stakeholder erfasst werden. Innerhalb des *Requirements-Engineering* werden dafür häufig kleine Karteikarten genutzt, um später eine leichte Um- und Einsortierung der Anforderungen zu ermöglichen. Auf den Karten können auch die Nicht-funktionalen Anforderungen, wie z.B. Performanz oder Benutzbarkeit, vermerkt werden. Zwecks Übersichtlichkeit werden die erarbeiteten *User-Stories* in tabellarischer Form aufgeführt. Größere *User-Stories* werden *Epics* genannt.

| Rolle | Ziel/Wunsch | Kat. |
|---|---|---|
| Als Patient | möchte ich meine Arztbriefe verwalten | Epic |
| Als Patient | möchte ich meine Arztbriefe speichern | Story |
| Als Patient | möchte ich meine Arztbriefe ansehen | Story |
| Als Patient | möchte ich meine Arztbriefe bearbeiten | Story |
| Als Patient | möchte ich meine Arztbriefe löschen | Story |

Tabelle 3.1: User Stories: Patient - Arztbriefe verwalten (Epic)

| Rolle | Ziel/Wunsch | Kat. |
|---|---|---|
| Als Patient | möchte ich meine Erkrankungen verwalten | Epic |
| Als Patient | möchte ich meine Erkrankungen speichern | Story |
| Als Patient | möchte ich meine Erkrankungen ansehen | Story |
| Als Patient | möchte ich meine Erkrankungen bearbeiten | Story |
| Als Patient | möchte ich meine Erkrankungen löschen | Story |

Tabelle 3.2: User Stories: Patient - Erkrankungen verwalten (Epic)

| Rolle | Ziel/Wunsch | Kat. |
|---|---|---|
| Als Patient | möchte ich meine Medikamente verwalten | Epic |
| Als Patient | möchte ich meine Medikamente speichern | Story |
| Als Patient | möchte ich meine Medikamente ansehen | Story |
| Als Patient | möchte ich meine Medikamente bearbeiten | Story |
| Als Patient | möchte ich meine Medikamente löschen | Story |

Tabelle 3.3: User Stories: Patient - Medikamente verwalten (Epic)

| Rolle | Ziel/Wunsch | Kat. |
|---|---|---|
| Als Patient | möchte ich meine Untersuchungen verwalten | Epic |
| Als Patient | möchte ich meine Untersuchungen speichern | Story |
| Als Patient | möchte ich meine Untersuchungen ansehen | Story |
| Als Patient | möchte ich meine Untersuchungen bearbeiten | Story |
| Als Patient | möchte ich meine Untersuchungen löschen | Story |

Tabelle 3.4: User Stories: Patient - Untersuchungen verwalten (Epic)

| Rolle | Ziel/Wunsch | Kat. |
|---|---|---|
| Als Patient | möchte ich meine med. Rechnungen verwalten | Epic |
| Als Patient | möchte ich meine med. Rechnungen speichern | Story |
| Als Patient | möchte ich meine med. Rechnungen ansehen | Story |
| Als Patient | möchte ich meine med. Rechnungen bearbeiten | Story |
| Als Patient | möchte ich meine med. Rechnungen löschen | Story |

Tabelle 3.5: User Stories: Patient - med. Rechnungen verwalten (Epic)

| Rolle | Ziel/Wunsch | Kat. |
|---|---|---|
| Als Patient | möchte ich meine medizinischen (Abrechnungs-)Daten graphisch anzeigen | Epic |
| Als Patient | möchte ich eine graphische Darstellung, welche Arztbriefe zu welchem Arzt gehören | Story |
| Als Patient | möchte ich eine graphische Darstellung, welche Medikamente für welche Erkrankung verordnet wurden | Story |
| Als Patient | möchte ich eine graphische Darstellung, der Status ('erhalten', 'bezahlt', 'eingereicht', 'erstattet') meiner Rechnungen | Story |

Tabelle 3.6: User Stories: Patient - Daten graphisch anzeigen (Epic)

| Rolle | Ziel/Wunsch | Kat. |
|---|---|---|
| Als Patient | möchte ich meine medizinischen (Abrechnungs-)Daten von Fremdsystemen importieren | Epic |
| Als Patient | möchte ich meine Arztbriefe von Fremdsystemen importieren | Story |
| Als Patient | möchte ich meine Diagnosen aus dem Arztbrief als Erkrankungen speichern | Story |
| Als Patient | möchte ich meine Medikamente aus dem Arztbrief unter Medikation speichern | Story |
| Als Patient | möchte ich meine Untersuchungen aus dem Arztbrief separat speichern | Story |
| Als Patient | möchte ich meine Untersuchungen von Fremdsystemen importieren | Story |
| Als Patient | möchte ich meine Rechnungen von Fremdsystemen importieren | Story |

Tabelle 3.7: User Stories: Patient - Daten importieren (Epic)

| Rolle | Ziel/Wunsch | Kat. |
|---|---|---|
| Als Arzt | möchte ich medizinischen Daten eines Patienten graphisch anzeigen | Epic |
| Als Arzt | möchte ich alle Diagnosen und die dazu gehörigen Untersuchungen eines Patienten graphisch anzeigen | Story |
| Als Arzt | möchte ich alle Diagnosen und die dazu gehörigen Medikamente eines Patienten graphisch anzeigen | Story |

Tabelle 3.8: User Stories: Arzt - Patientendaten graphisch anzeigen (Epic)

| Rolle | Ziel/Wunsch | Kat. |
|---|---|---|
| Als Versicherer | möchte ich medizinischen Daten eines Patienten graphisch anzeigen | Epic |
| Als Versicherer | möchte ich alle Diagnosen und Medikamente eines Patienten graphisch anzeigen | Story |

Tabelle 3.9: User Stories: Versicherungskaufmann - Patientendaten graphisch anzeigen (Epic)

## 3.2 Spezifikation

Die erhobenen funktionalen und nicht-funktionalen Anforderungen der Stakeholder werden genutzt, um eine Spezifikation zu erstellen. Dort werden Geschäftsprozesse formuliert, Use-Cases definiert und mittels fachlichem Datenmodell (FDM) und Wireframes erste Designentscheidungen spezifiziert.

### 3.2.1 Geschäftsprozesse

Ein Geschäftsprozess unterstützt ein unternehmensbezogenes Ziel, besteht aus mehreren Einzelschritten (Aktionen), wird häufig arbeitsteilig durch mehrere Bereiche durchgeführt, erfordert in der Regel Unterstützung durch ein oder sogar mehrere Softwaresysteme, verarbeitet Informationen und führt zu einem durch das Unternehmen gewünschten Ergebnis. Gadatsch (2017)
Für die nachfolgende Geschäftsprozessmodellierungen wurde die *Business Process Modeling Notation* gewählt Freund (2014). Um die Übersicht zu bewahren, wurden teilweise komplexere Aufgaben mittels einem Teilprozess-Symbol dargestellt und die „Interna“ einiger *Pools* verborgen. Die Nachrichtenflüsse beginnen und enden dann am Rand des Pools. Allweyer (2008).

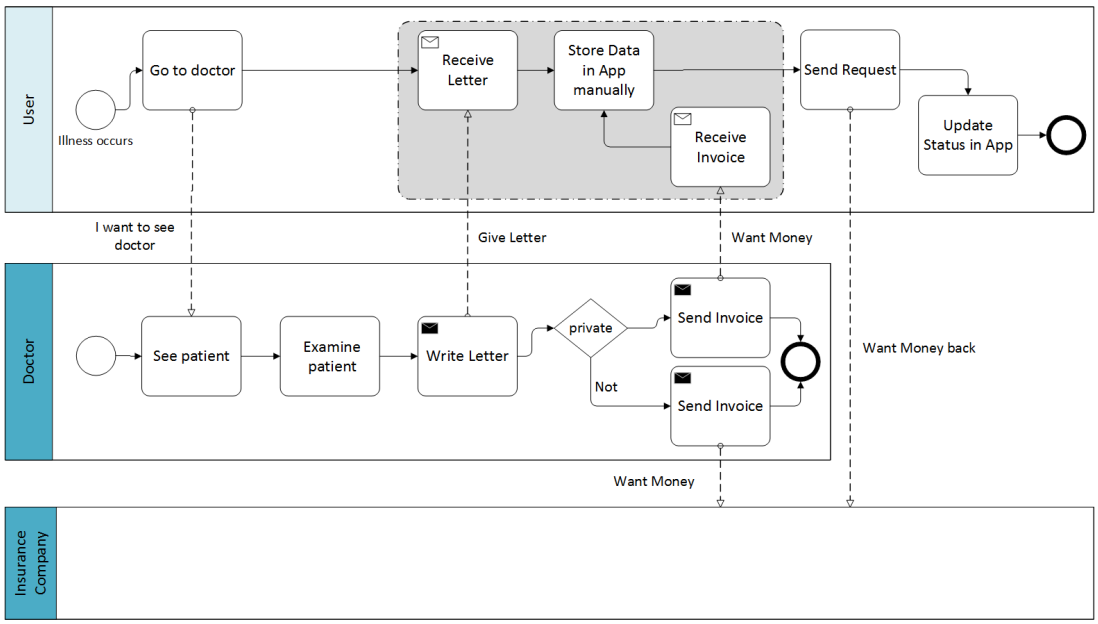


Abbildung 3.1: BPMN: Medizinische Daten und Rechnungen manuell speichern

### 3.2.2 Use-Cases

Use-Cases konzentrieren sich auf die Art und Weise wie ein Aktor die *Benutzung* des Systems aus *Außensicht* wahrnimmt Umbach und Metz (2006).
Für die prototypische Anwendung wurden vier verschieden Use-Cases ausgearbeitet. Die Basis dafür bilden sowohl die bereits auf dem Markt befindlichen Konkurrenzprodukte (Microsoft Vault, LifeTime), als auch die persönlichen beruflichen Erfahrungen des Autors als praktizierender Arzt.

#### Use-Case 1: Erkrankungen und Medikamente als Graph ansehen

Akteur: Arzt
Ziel: Medizinische Daten eines Patienten einsehen
Auslöser: Patient kommt zur Behandlung
Vorbedingungen: Arzt muss sich eingeloggt haben
Nachbedingungen: Im Log-Bereich ist der Zugriff vermerkt worden
Erfolgsszenario:
1. In der Navigationsleiste den Reiter „Analyse“ anklicken
2. Häkchen vor den gewünschten Daten machen, hier bei: „Diseases“ und „Medication“
3. Auf den Button „View Cypher“ klicken
4. Das System prüft die Eingaben
5. Die benötigte Cypher Syntax wird in einem Fenster angezeigt
6. Die Syntax kopieren und das Fenster schließen
7. Auf den Button „View Graph“ klicken
8. Das System öffnet einen neuen Browser-Tab mit dem Oberflächen-Interface der DB
9. Die Cyphersyntax in das vorgesehene Feld kopieren und auf den Pfeil daneben klicken
10. Die Daten werden als Graph angezeigt

#### Use-Case 2: Neue Diagnose (Erkrankung) anlegen

Akteur: Patient
Ziel: Neue Diagnose im System speichern
Auslöser: Patient erhielt neue Diagnose beim Arzt
Vorbedingungen: Patient muss sich eingeloggt haben
Nachbedingungen: Der neue Datensatz befindet sich im System
Erfolgsszenario:

1. In der Navigationsleiste den Reiter „Import“ anklicken
2. Im Drop-Down Menü des Bereiches *Diseases* „Manually Import“ auswählen
3. Den Button „Start“ klicken
4. Ein Formular öffnet sich
5. Daten ins Formular eingeben (Diagnose, Datum, Arzt)
6. Auf den Button „Check & Save“ klicken
7. Das System prüft die Eingaben
8. Eine Meldung „Import successful!“ erscheint im Formular

Fehlerfälle:

7.a System erkennt Fehler bei der Eingabe: System meldet Anwendungsfehler und markiert fehlerhafte oder fehlende Eingaben

#### Use-Case 3: Status einer Rechnung ändern

Akteur: Patient
Ziel: Status von Rechnungen verwalten
Auslöser: Patient hat eine von ihm bezahlte Rechnung bei der Krankenkasse eingereicht
Vorbedingungen: Patient muss sich eingeloggt haben
Nachbedingungen: Status der ausgewählten Rechnung hat sich geändert
Erfolgsszenario:

1. In der Navigationsleiste den Reiter „Tracking“ anklicken
2. Gewünschte Rechnung in der Spalte „PAID“ anklicken
3. Das System zeigt eine Detail-Ansicht der Rechnung
4. Im Dropdown-Menu „SUBMITTED“ auswählen
5. Den Button „Change Status“ drücken
6. Das System prüft die Eingaben und ändert den Status der Rechnung
7. Das System lädt die Seite neu und die Rechnung wird in der Spalte „SUBMITTED“angezeigt

Fehlerfälle:

6.a System konnte keine Verbindung zur Datenbank aufbauen und bittet den Anwender es später noch einmal zu versuchen

#### Use-Case 4: Datenimport eines Arztbriefes durchführen

Akteur: Patient
Ziel: Medizinische Daten aus Fremdsystem importieren

Auslöser: Patient wurde im Krankenhaus untersucht
Vorbedingungen: Patient ist eingeloggt, Daten liegen auf Fremdsystem vor
Nachbedingungen: System hat die neuen medizinische Daten gespeichert
Erfolgsszenario:
1. Zum Bereich „Import“ wechseln
2. Im Drop-Down Menü des Bereiches *Physician letters* „Auto Import“ auswählen
3. Den Button „Start“ drücken
4. Ein Formular öffnet sich
5. Relevante Daten (Datum) eingeben
6. Den Button „Check & Save“ drücken
7. Das System prüft ob Daten bei der Gegenstelle vorliegen
8. Das System import die neuen Daten
9. Eine Meldung „New Physician Letter was found and imported!“ erscheint
Fehlerfälle:
7.a System findet keine neuen Datensätze zum importieren: System meldet dieses und fordert, die Eingabe des Datums zu überprüfen

### 3.2.3 FDM

Das fachliche Datenmodell von graph-basierten Datenbanken unterscheidet sich gegenüber den klassischen Modellen von relationalen Datenbanken. Vier Hauptcharakteristika sind dabei im Vordergrund. Man spricht vom *Labeled Property Graph Model*. Robinson (2015)

*Nodes* (Knoten)
*Relationships* (Beziehungen)
*Properties* (Attribute/Eigenschaften)
*Labels* (Rollen/Typen)

Es besteht auch die Möglichkeit, einzelne Attribute aus den Knoten heraus zu ziehen und als eigenständigen Knoten zu repräsentieren. Generell stellt ein Knoten nur einen Datencontainer dar, der erst durch das hinzufügen von Attributen und Labels eine Entität sinnvoll repräsentieren kann.
Da es kein festes Schema für das Vorhandensein von Attributen gibt, kann man jedem Knoten eine unterschiedliche Anzahl von Eigenschaften zu weisen. Kennt man z.B. nur wenige Daten über eine Person, werden nur diese dem Knoten hinzugefügt. Im folgenden Datenmodell sind

die angegebenen Attribute, abgesehen von der *id*, optional. Anders als bei relationalen Datenbanken, müssen keine *null*-Werte vergeben werden. Die Anzahl der *Labels* ist nicht beschränkt und kann für jeden Knoten neu definiert werden.
Die Abbildung 3.2 zeigt alle Hauptcharakteristika des Graph Models. Da diese Ansicht auf Grund ihrer mehrfachen Überlappungen in der späteren Darstellung im Browser eher selten vorkommt, wurden exemplarisch noch reduzierte Ansichten in den folgenden Abbildungen dargestellt.
Im Fokus der Abbildung 3.3 steht der *Patient* Knoten und seine wichtigsten Relationen zu den Arztbriefen (*Letter*), den Rechnungen (*Invoice*), den Medikamenten (*Drug*) und seinen Krankheiten (*Disease*).
Die Abbildung 3.4 zeigt die späteren Komponenten einen Arztbriefes, die übrigen Knoten und Relationen wurden zwecks Übersichtlichkeit nicht dargestellt.

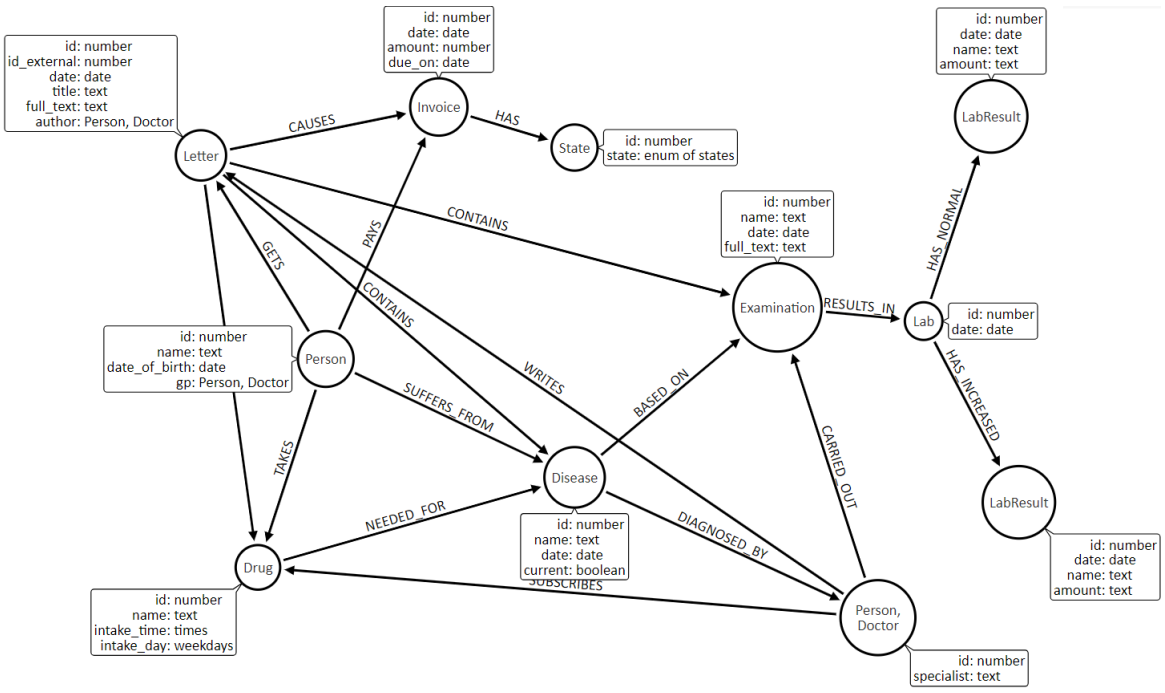


Abbildung 3.2: Labeled Property Graph Model

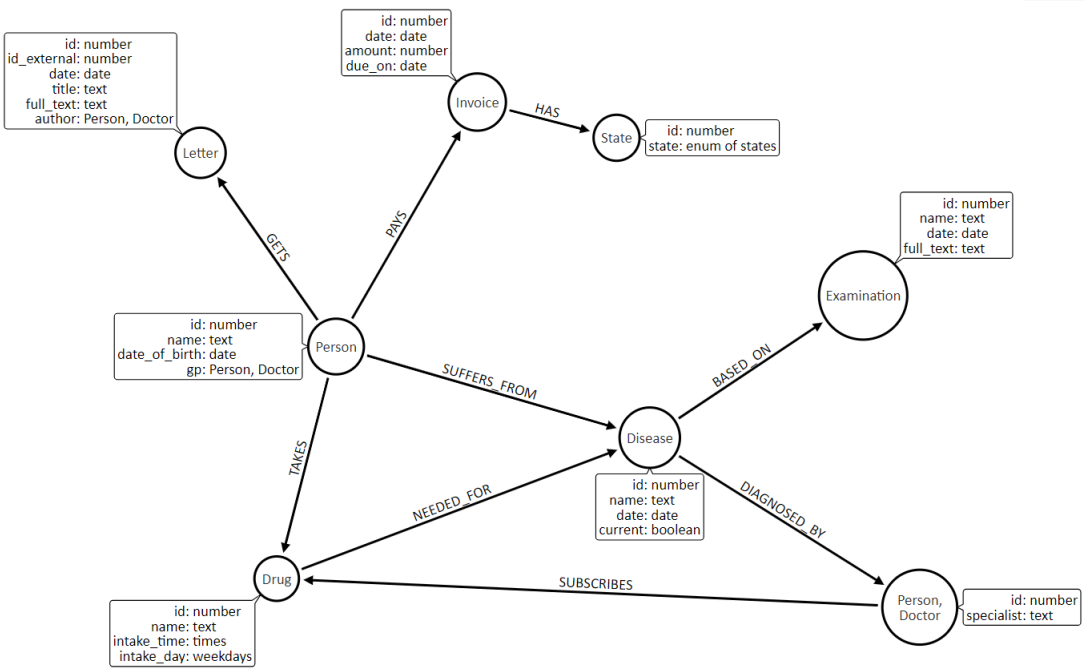


Abbildung 3.3: Labeled Property Graph Model - *Patient*

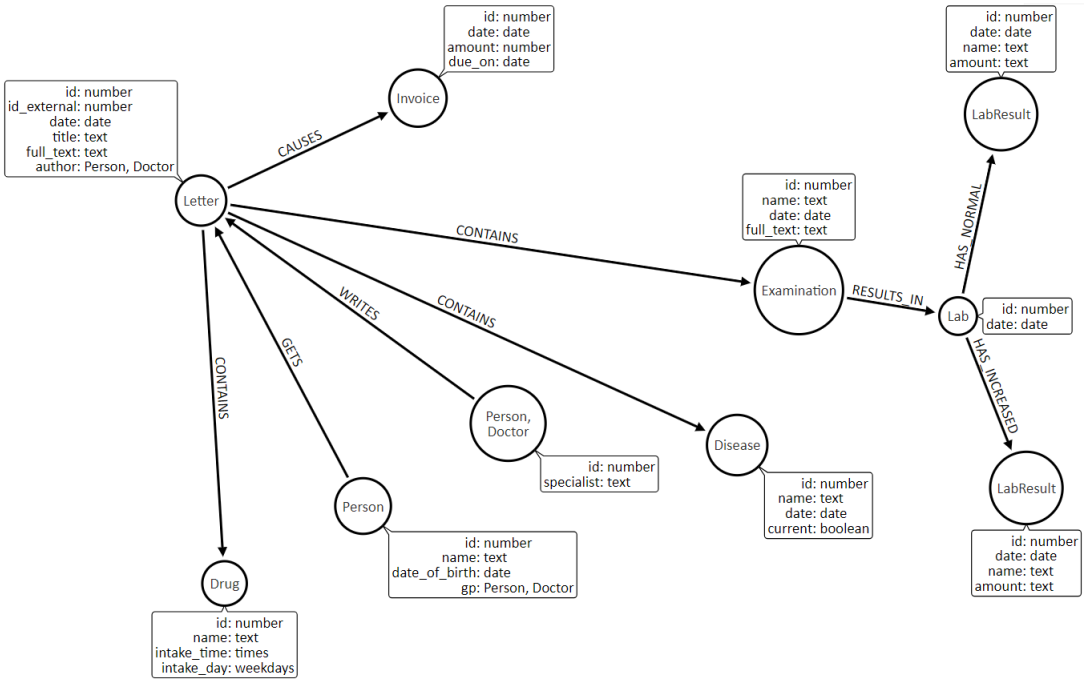


Abbildung 3.4: Labeled Property Graph Model - *Letter*

### 3.2.4 Wireframes

Bei der Entwicklung des Frontends ist es zunächst wichtig, grobe Skizzen bzw. Vorformen einer Website zu erstellen. Ein *Wireframe* enthält dabei noch keine Details wie Farben oder Bilder, sondern es steht eher die Überprüfung der Machbarkeit, des Navigationskonzepts und der Usability im Vordergrund.
Für die folgenden Wireframes wurde das Programm *Balsamiq*[1] verwendet. Nach Abstimmung mit dem Kunden, können weitere Details wie Grafiken, Farbverläufe und ähnliches hinzugefügt werden. Im weiteren Entwicklungsprozess und mit steigendem Detailgrad wird dann auch von *Mockups* gesprochen, die dem Kunden ein besseres „look and feel“ Erlebnis vermitteln als die gröberen Wireframes.

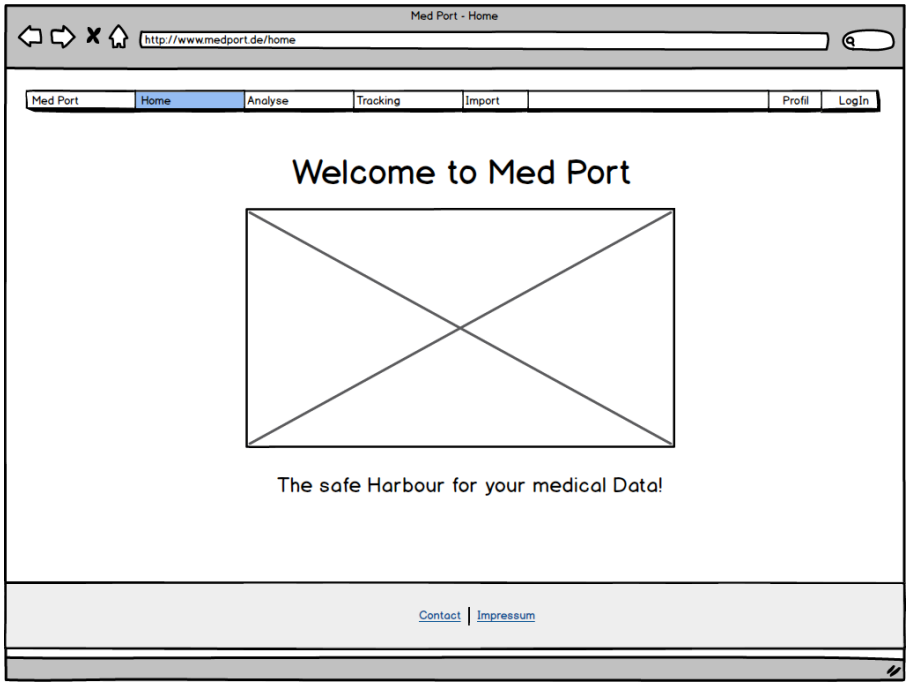


Abbildung 3.5: Startseite: Med Port - Home

[1] https://balsamiq.com

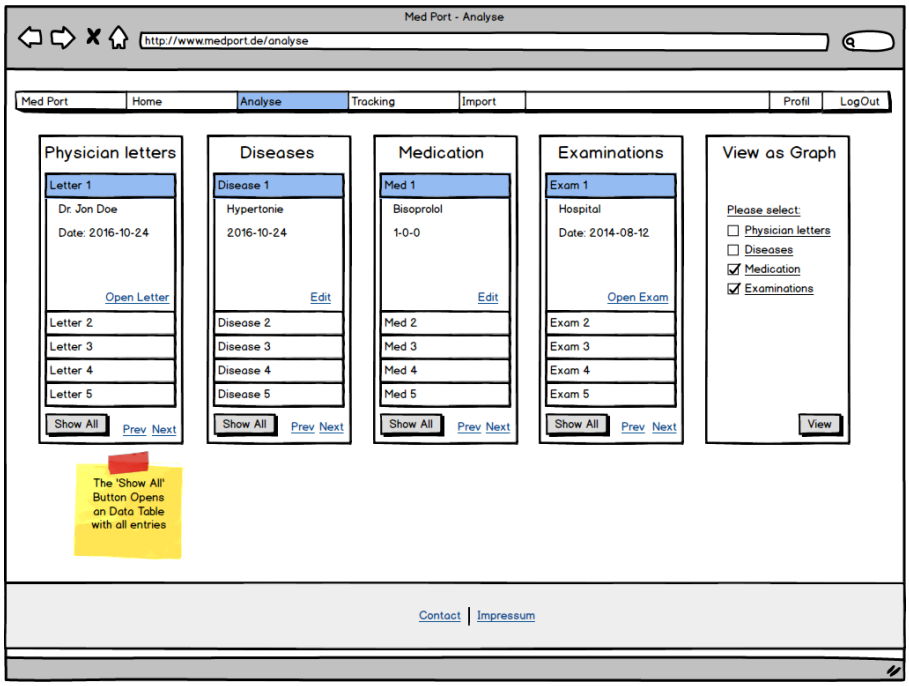


Abbildung 3.6: Analyseseite: Med Port - Analyse

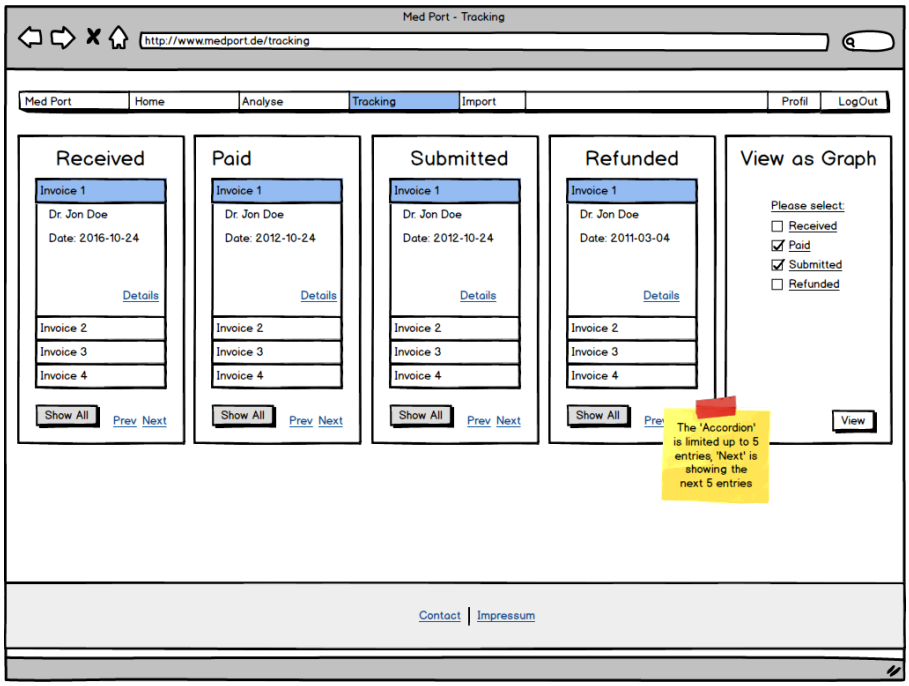


Abbildung 3.7: Trackingseite: Med Port - Tracking

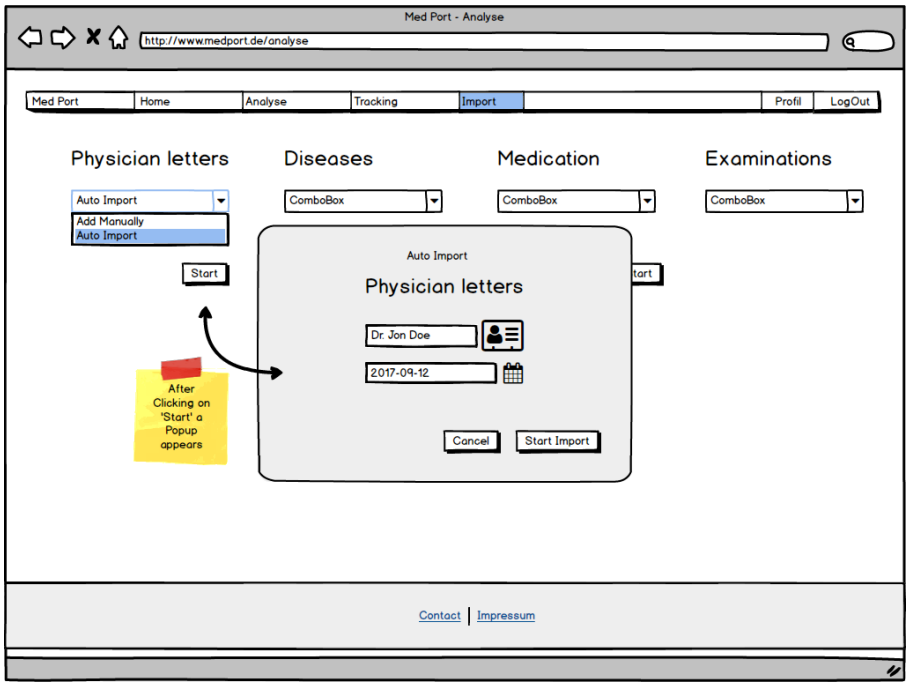


Abbildung 3.8: Importseite: Med Port - Import

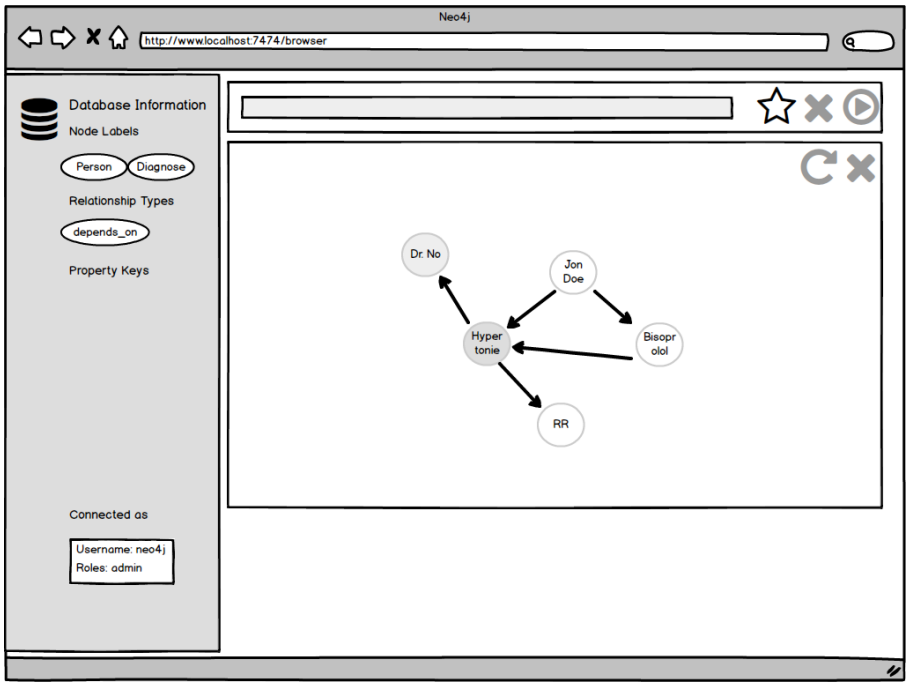


Abbildung 3.9: Im Browser wird der Graph der Neo4j-DB angezeigt

# 4 Konzeption und Architektur

## 4.1 Architektursichten

Für die verschiedenen Projektbeteiligten und deren jeweiligen Bedürfnisse und Belange ist jeweils eine andere Architektursicht von Nutzen. Im weiteren werden folgende Sichten näher betrachtet und erläutert: Kontextabgrenzung, Laufzeitsicht, Bausteinsicht und Verteilungssicht Starke (2015).

### Kontextabgrenzung und Bausteinsicht

Die Kontextabgrenzung und die Bausteinsicht werden in Abbildung 4.1 gemeinsam dargestellt. Das oberste Level 0 der Bausteinsicht ist äquivalent zur Kontextsicht. Es zeigt die Interaktion des Users mit dem System und welche weiteren Systeme beteiligt sind. Das mit dem dem Label *THIRD PARTY* versehende System soll exemplarisch für ein Fremdsystem stehen, aus dem Daten importiert werden können.
Die Bausteinsicht beginnt ab dem Level 1 und zeigt die *Microservice-Architektur* des späteren Prototypen. Der Service „GUI“ stellt die Schnittstelle zum User dar, der Service „BL“ repräsentiert die Geschäftslogik und der dritte Service „DB“ steht stellvertretend für das Datenbankmanagementsystem. Alle Anfragen des Users werden an die Geschäftslogik über eine REST-API weitergeleitet und dort verarbeitet. Die Kommunikation mit der Datenbank und anderen Fremdsystemen erfolgt ausschließlich über die Geschäftslogik und nutzt dafür ebenfalls eine REST Schnittstelle.

Das Level 2 der Bausteinsicht zeigt den näheren Aufbau dieser Logik. Das Package *Core* beinhaltet alle Entitäten und die für den Aufbau und Datenaustausch benötigten Komponenten. Die Schnittstelle zur Datenbank stellen die „Repositories“ dar, eine Anbindung an die GUI erfolgt mittels der „Controller“ (siehe Level 3).
Im Package *Logic* befinden sich die Implementationen der Use-Cases und eine Komponente, um Fremdsysteme mittels API anzubinden. Im Package *Main* wird die Anwendung gestartet. Das Package *Helper* beinhaltet zahlreiche Hilfsfunktionen. Diese dienen vor allem dazu, Trans-

formationen zwischen verschiedenen Eingabeformaten und Datentypen zu ermöglichen.
Um Übersichtlichkeit zu gewährleisten, wurden die Beziehungen der Einzelnen Klassen untereinander in eine separate Abbildung (Vgl. Abb. 4.2) ausgelagert. Auf diesem Level 4 sind die Whitebox-Sichten mehrerer Packages dargestellt.
Im Package Entities nimmt die Klasse *Letter* eine zentrale Rolle ein, da sie alle übrigen Entitäten, ValueTypes und Enums direkt oder indirekt beinhaltet. Da es dem Benutzer auch möglich ist, ohne des Vorliegens eines Arztbriefes neue Daten (Medikamente, Diagnosen...) hinzuzufügen, gibt es auch Assoziationen zwischen den einzelnen Klassen selbst (z.B.: Patient -diseases-> Diseases). Um die Übersichtlichkeit nicht weiter einzuschränken, wurden diese Zusammenhänge nicht graphisch dargestellt, sondern als Attribut in der Klasse selbst aufgeführt.

Das Level 3 zeigt ebenfalls das Package *Logic*, welches die Packages „ThirdParties“ und „UseCases“ enthält. Bei der Umsetzung der in Kapitel 3 beschriebenen *Use-Cases*, nimmt das Package *UseCases* eine zentrale Rolle ein. Es setzt sich aus den Bestandteilen: *UseCaseInterfaces*, *UseCaseImpl* und *UseCaseController* zusammen. Dabei werden die jeweiligen Interfaces von den UseCase-Klassen implementiert. Der REST-Controller steht für den Datenaustausch zur GUI bereit und greift auf das jeweilige Interface zu. (Vgl. Abb. 4.3)

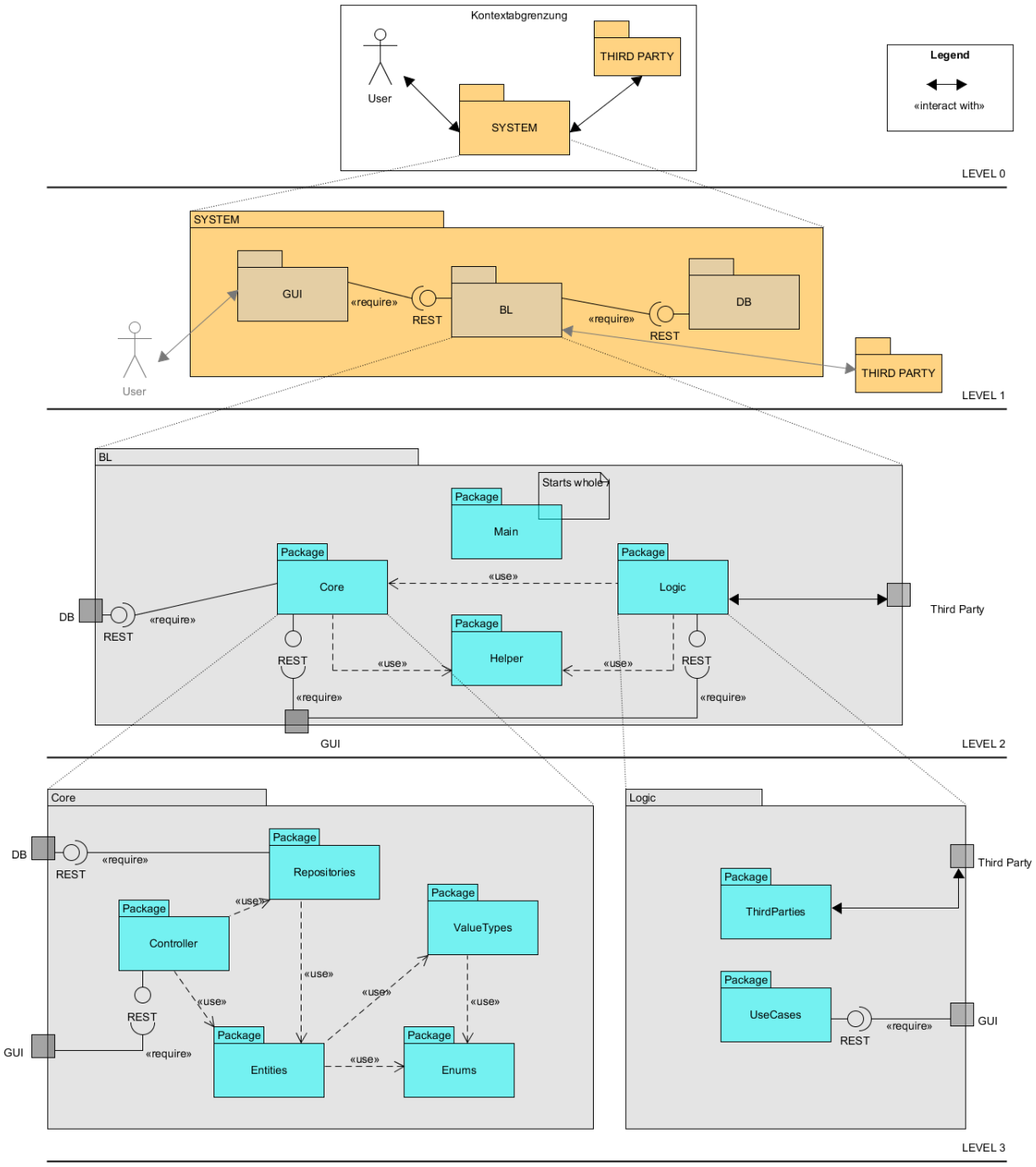


Abbildung 4.1: Hierarchie und Verfeinerung der Bausteinsicht

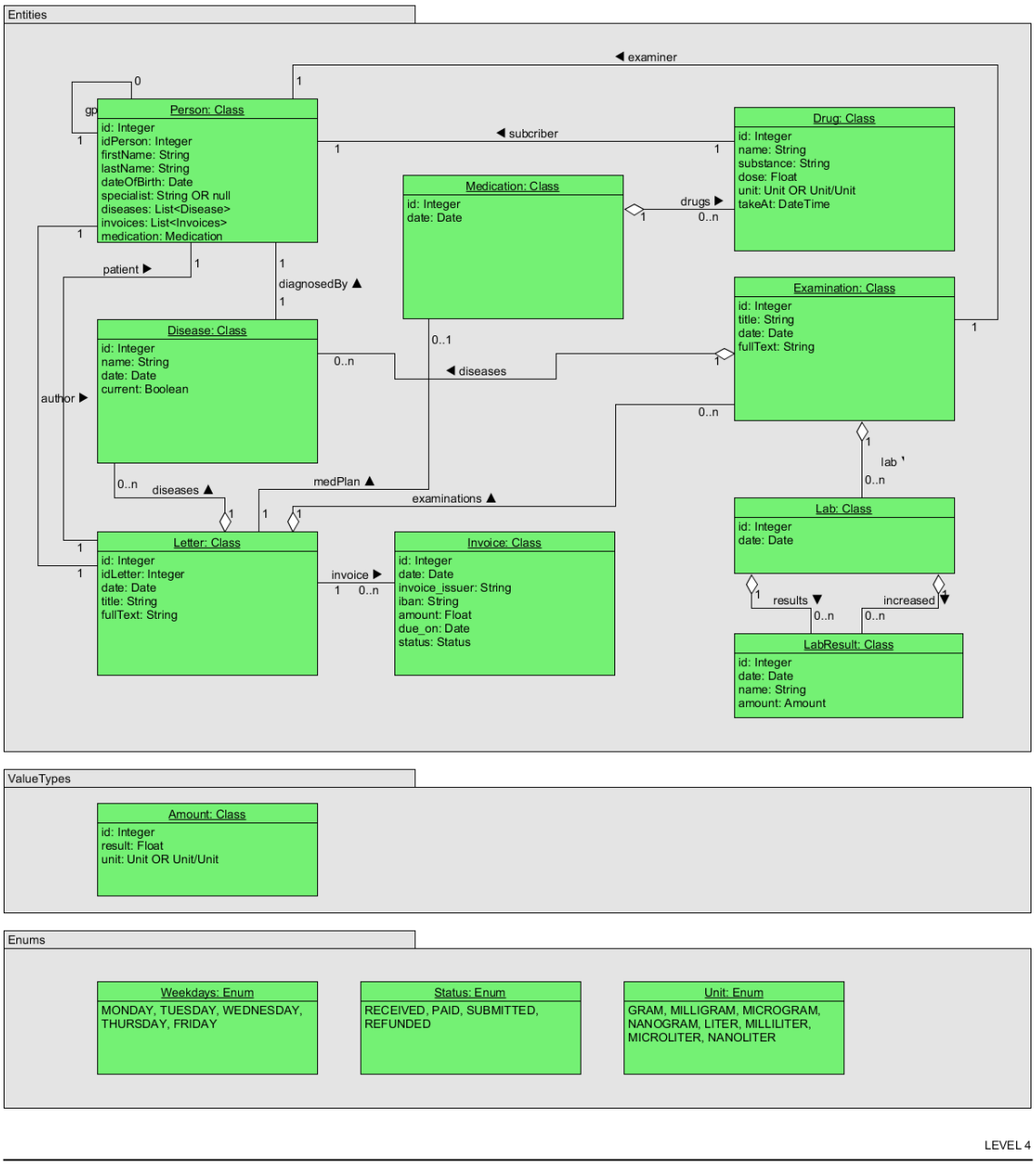


Abbildung 4.2: Assoziationen zwischen *Entities*, *Enums* und *ValueTypes*

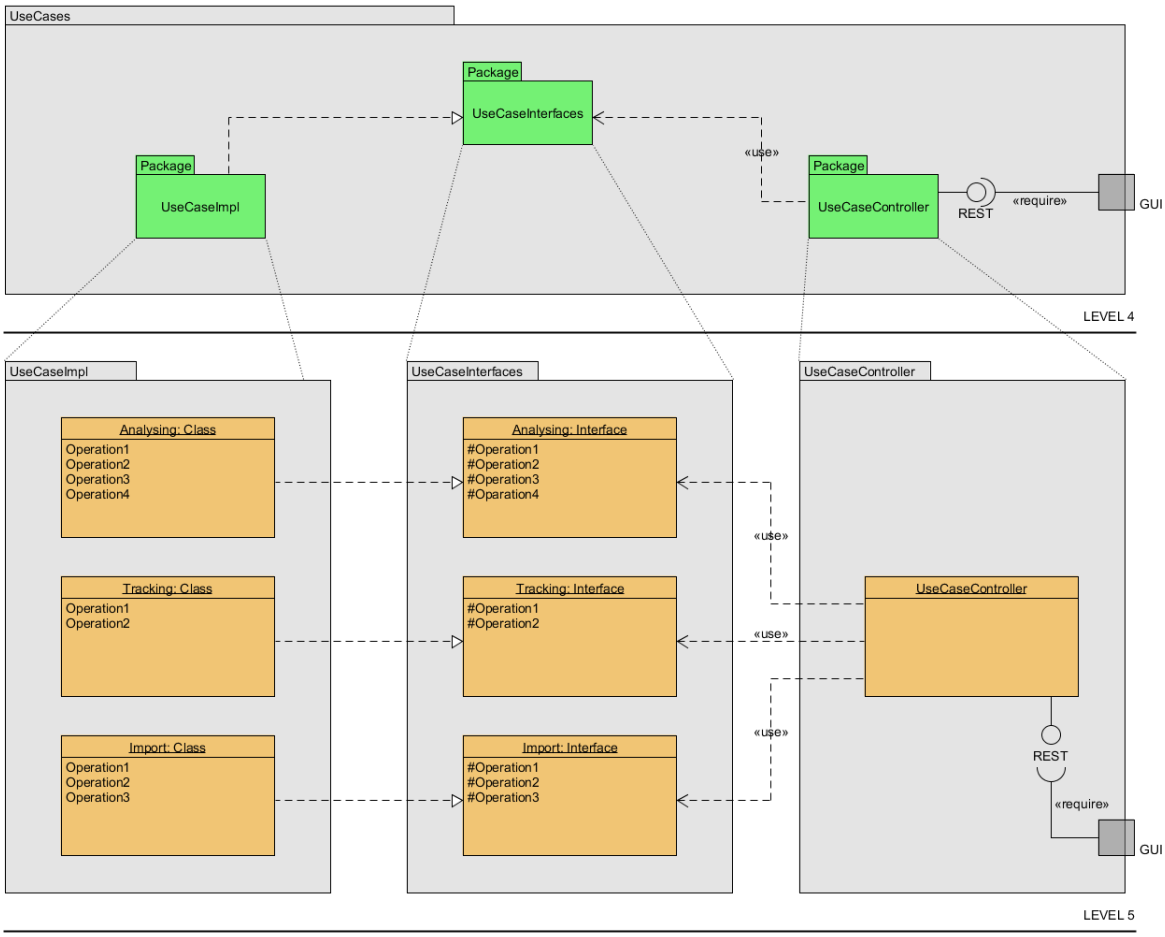


Abbildung 4.3: Whitbox-Sichten der Packages *Interfaces* und *UseCaseImpl*

### Laufzeitsicht und Verteilungssicht

Die Laufzeitsicht beschreibt, welche Bestandteile des Systems zur Laufzeit existieren und wie sie dabei zusammenwirken. Diese Sicht dokumentiert, wie das System zur Laufzeit seine wesentlichen Aufgaben (z.B. Use-Cases) ausführt Starke (2015).
Im Sequenzdiagramm vom *Use-Case 1* (Abb. 4.4) sind die einzelnen Methodenaufrufe und Rückgabewerte, soweit diese bekannt sind, vermerkt. Bei der Umsetzung sind alle Komponenten der Anwendung beteiligt. Die mittels AJAX-GET-Call angeforderte „Cypher Syntax“ wird als *ResponseEntity* an die GUI zurückgegeben und dem Akteur im Modal anzgezeigt. Diese wird dann vom Benutzer für den späteren Einsatz kopiert. Nachdem sich ein neuer Browser-Tab mit der Neo4j-Oberfläche geöffnet hat, fügt der Akteur den Query-String ein und klickt auf den Button „Play“. Die gewünschten Daten werden abgerufen und im Browser angezeigt.

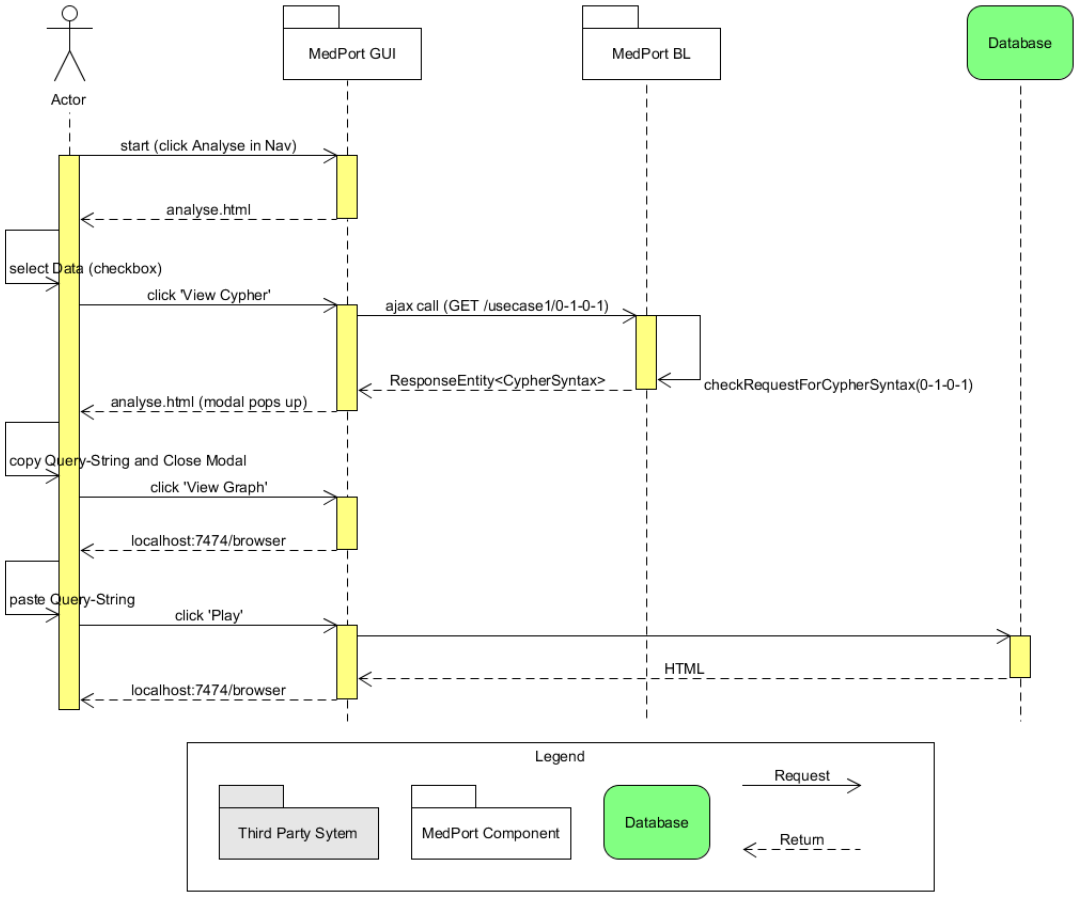


Abbildung 4.4: Sequenzdiagramm: *Use-Case 1*

Die Abbildung 4.5. zeigt den Use-Case für das manuelle Importieren einer Erkrankung (engl.: *Disease*). Auch bei diesem Anwendungsfall sind alle Services beteiligt. Der von *Spring* verwandte „HTTP JSON Converter“ überprüft den JSON-Body des AJAX-POST-Calls. Bei korrekter Syntax wird die neue Knoten-Entität mittels des Disease-Repository gespeichert. Der Rückgabewert der Business Logic enthält die neu gespeicherte Entität und den HTTP-Response Code 201 („CREATED“).
Letztendlich erscheint unterhalb des Import-Formulars für ein paar Sekunden die Nachtricht „Import successful!“. Wechselt der Anwender im Anschluss in den Analyse-Bereich, erscheint dort der eben getätigte Import.

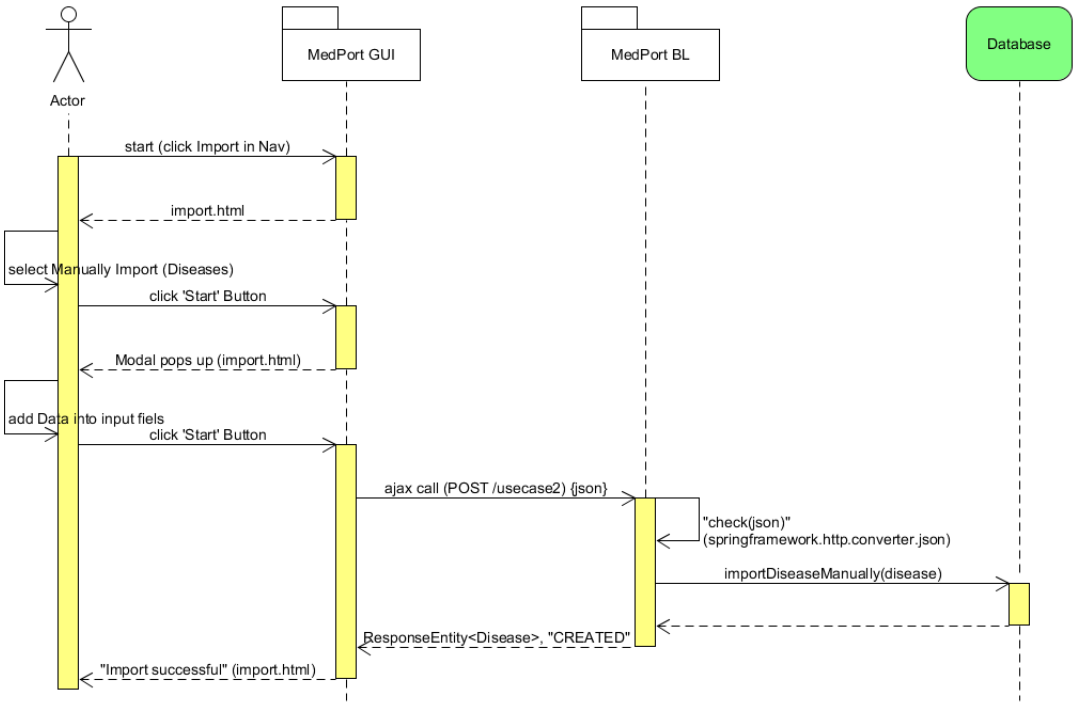


Abbildung 4.5: Sequenzdiagramm: *Use-Case 2*

Wie der Status einer Rechnung (engl.: *Invoice*) geändert werden kann, zeigt das Sequenzdiagramm in Abbildung 4.6. Nachdem der Akteur die Detail-Ansicht der gewünschten Rechnung geöffnet hat, kann einer der vier möglichen Status ausgewählt werden. Der Button „Change Status“ löst einen AJAX-POST-Call aus.
Im Anschluss an die Überprüfung des JSON-Bodys wird ein Java-Objekt daraus erzeugt. Die

übermittelte *InvoiceId* wird genutzt, um die Rechnung zunächst in der Datenbank zu löschen. Danach wird eine neue Invoice-Knoten-Entität in der Graphdatenbank mit den ursprünglich übermittelten JSON-Daten erzeugt.
Der HTTP-Response-Code 201 („CREATED“) signalisiert die erfolgreiche Änderung des Status. Die GUI lädt die HTML-Seite neu. Dabei werden die aktuellen Daten angezeigt.

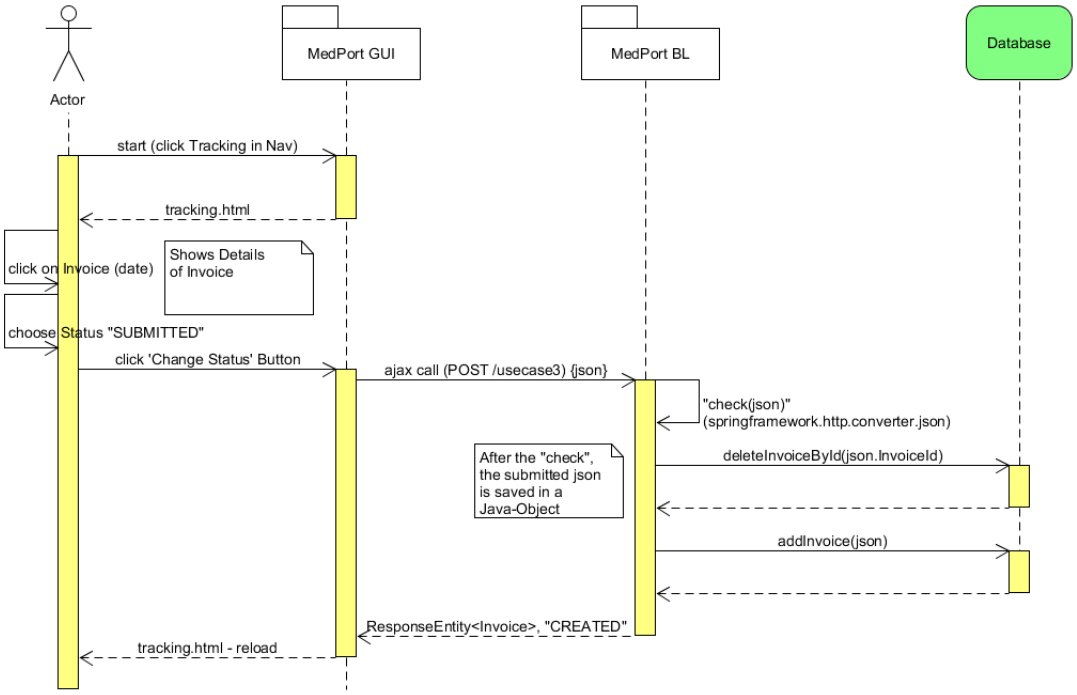


Abbildung 4.6: Sequenzdiagramm: *Use-Case 3*

Im abschließend dargestellten *Use-Case 4* Sequenzdiagramm (Abb. 4.7) wird das Importieren von Daten aus einem Fremdsystem (engl.: *Third Party System*) veranschaulicht. Das Fremdsystem wird bei dieser prototypischen Anwendung durch die Komponente BL *gemockt*.
Um diesen Anwendungsfall erfolgreich durchzuführen, wählt der Benutzer als Datum den letzten Tag des Jahres 2018. Hierfür findet das System später einen neuen Import. Mittels AJAX Call wird eine Abfrage bei der BL gestartet. Das Datum wird dabei zu einem String umgewandelt und dient später als ID. Per festgelegter Konvention bekommt jeder Brief als „LetterId“ das Datum der Erstellung (31.12.2018 -> *20181231*).
Das Fremdsystem übermittelt der BL den neuen Brief, der dann anschließend mittels Letter-

Repository in der Graphdatenbank gespeichert wird. Nach erfolgreicher Rückmeldung durch die DB, wird der neu importierte Brief als *ResponseEntity* an die GUI übermittelt. Die GUI aktualisiert daraufhin das HTML-Dokument und meldet den erfolgreichen Import.
Sollte der Anwender erneut den Import mit diesem Datum anstoßen, erhält er eine Meldung vom System, dass für dieses Datum bereits ein Arztbrief importiert wurde.

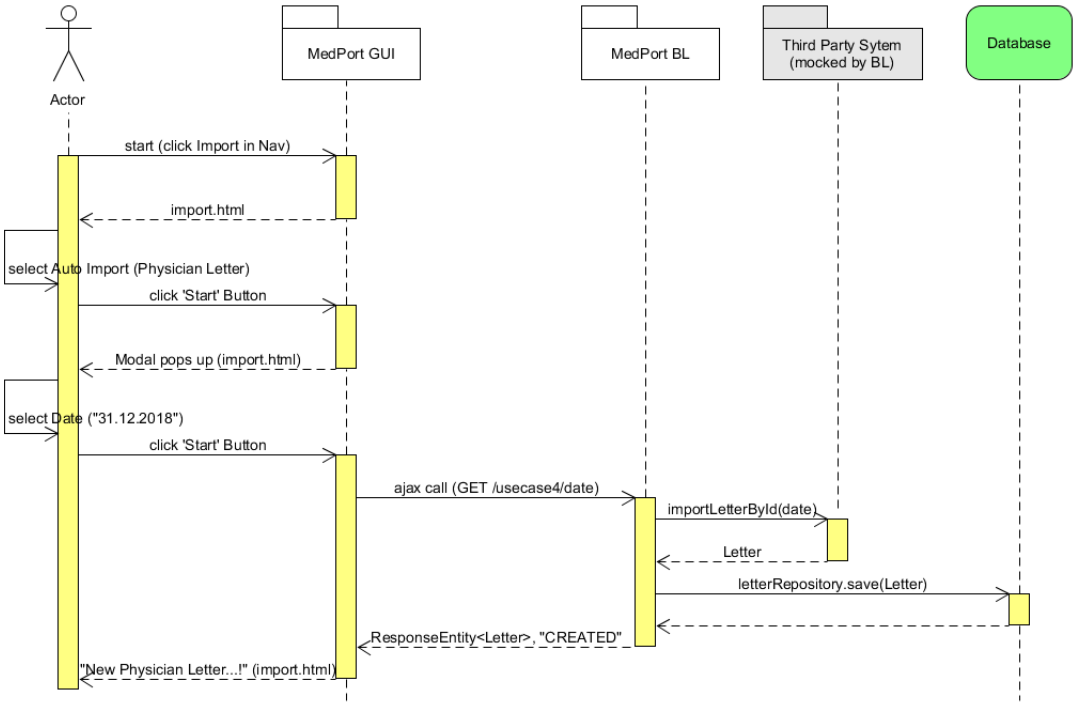


Abbildung 4.7: Sequenzdiagramm: *Use-Case 4*

Bei der Verteilungssicht wird die technische Infrastruktur dargestellt. Dabei werden vor allem drei Elemente verwandt: Die Knoten (engl.: *Nodes*), die Laufzeitelemente und die Kanäle. Die prototypische Anwendung wird mittels dreier Dockercontainer realisiert (*Nodes*). Die Verbindungskanäle werden applikationsintern durch das HTTP Protokoll und eine REST-Schnittstelle realisiert. Der Zugriff auf Dritt-Server wird in Zukunft über das Internet erfolgen. Dabei sollen externe Systeme eine REST und eine SOAP-Schnittstelle für den Datenabgleich anbieten.
Als Web- bzw. App-Server dient in zwei Fällen ein Tomcat-Server. Die einzelnen Dockercontainer haben die angegeben IP-Adressen.

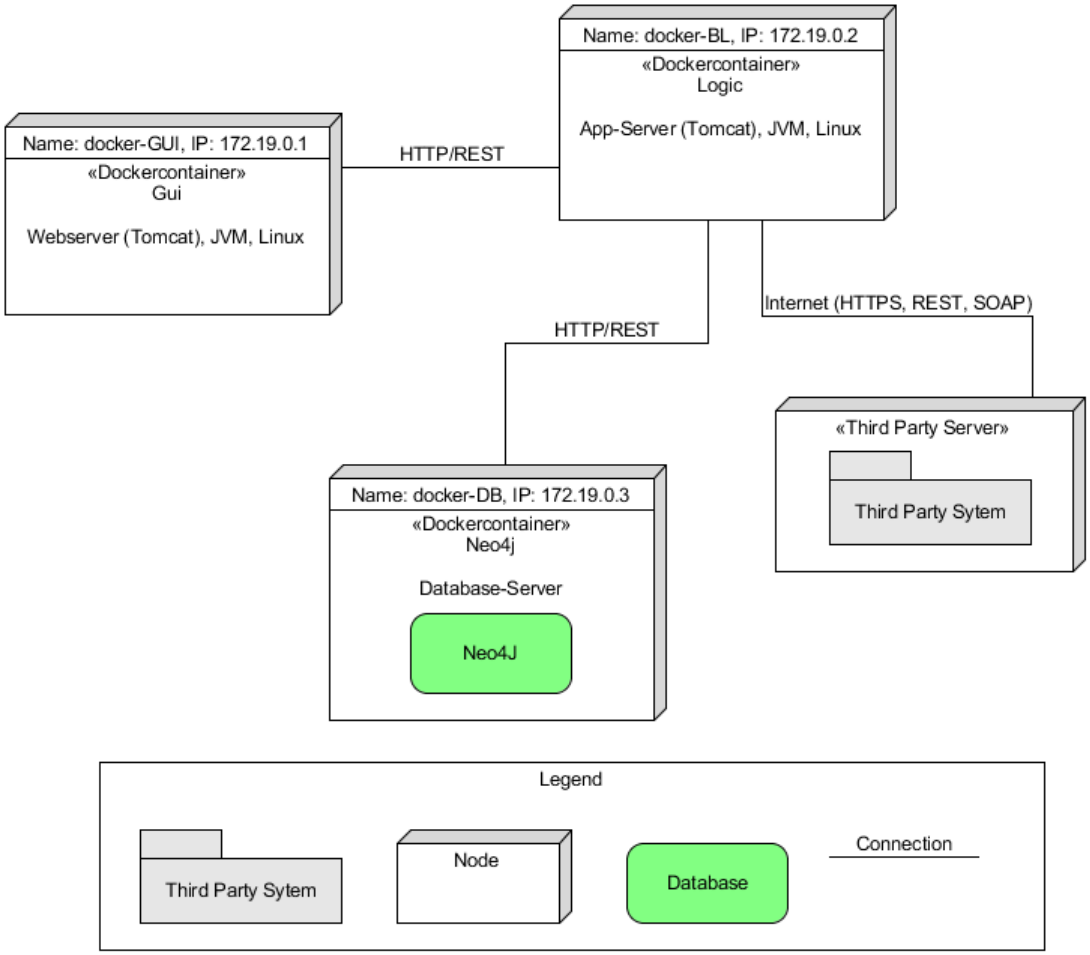


Abbildung 4.8: Verteilungssicht

## 4.2 Design-Entscheidungen

Entscheidungen über das Design einer Software haben große Auswirkungen auf den Entwicklungsprozess und die späteren Qualitätsmerkmale. Bei der Entwicklung dieser prototypischen Anwendung kamen zunächst drei Architekturstile in Frage: Monolith, Microservices oder SOA (*Service Oriented Architecture*).
Die beiden ersten Stile wurden bereits im Grundlagenteil beschrieben. Da bei der serviceorientierten Architektur normalerweise ein ESB (*Enterprise Service Bus*) zum Einsatz kommt, wurde dieser Stil wieder verworfen. Die Schnittstellen sollen leichtgewichtig sein und eine direkte Kommunikation zwischen den Services ermöglichen. Hinzu kommt, dass bei der Nutzung von drei Services, der Aufbau und die Implementation einer übergreifenden Struktur zur Orchestrierung und Nachrichtenverteilung überdimensioniert erscheint.
Generell wurden Design-Entscheidungen mit Blick auf die SOLID-Kriterien und weiteren Entwurfsmustern wie Modularisierung, Low-Coubling und High-Cohesion getroffen. Das endgültige Produkt ist dabei aber häufig eine Mischung aus verschiedenen Architekturstilen und Mustern Takai (2017). Im Kapitel „Realisierung“ und „Evaluierung“ wird noch einmal genauer auf bestimmte Entscheidungen bezüglich des Designs eingegangen.

### 4.2.1 Kommunikation zwischen den Microservices und der Datenbank

Die Kommunikation zwischen den Services bzw. der Datenbank erfolgt mittels REST API, die die einzelnen Bausteine anbieten. *Requests* und *Responses* werden über das *Hypertext Transfer Protokol* (HTTP) verschickt und empfangen. Die ausgetauschten Daten/Dokumente liegen im *JavaScript Object Notation* Format vor (JSON). Ein direkter Zugriff vom Frontend (Microservice GUI) auf die Datenbank ist nicht möglich, sondern erfolgt stets über den Microservice BL.
Die Dokumentation der REST-Schnittstelle erfolgt mittels „Swagger“. Die *OpenAPI Specification*[1] ermöglicht dabei einen sprachenunabhängigen Standard zur Beschreibung und zum späteren Verständnis des Interfaces, ohne Zugang zum Quellcode zu haben.

### 4.2.2 Frontend: HTML5 und Javascript

Bei der Umsetzung der Benutzeroberfläche wird die Hypertext Markup Language (HTML5) in Kombination mit der Skriptsprache JavaScript (JS) genutzt, um eine Interaktion des Anwenders mit den anderen Services zu ermöglichen.
Der Einsatz von Frameworks wie *jQuery* oder *Bootstrap* soll den Aufwand bei der späteren Implementierung verringern und bestimmte Qualitätsmerkmale, wie z.B. *responsive design*,

[1] http://www.swagger.io/specification/

sicherstellen.
Als Alternative wäre noch der Einsatz der serverseitig interpretierten Skriptsprache PHP[2] möglich gewesen. Da bereits Vorkenntnisse bei der Verwendung von JavaScript bestanden, fiel die Wahl schließlich auf diese Sprache.

### 4.2.3 Backend: Spring Boot, Spring Data

Für die prototypische Umsetzung der gewünschten Microservice-Architektur gibt es zahlreiche Frameworks und Sprachen. Um später eine möglichst gute Wartbarkeit und Erweiterbarkeit zu garantieren, wurde bei beiden Microservices das selbe Framework eingesetzt: *Spring*.
Bei der Wahl der Spring Projekte wurde vor allem darauf geachtet, dass eine möglichst unkomplizierte Umsetzung der gegebenen Anforderungen möglich war. Da *Spring Boot* mit dem Grundsatz „convention over configuration“ entwickelt wurde, fallen an zahlreichen Stellen aufwendige Konfigurationen weg, um eine Applikation laufen zu lassen. Hinzu kommt, dass das Framework bereits einen eingebetteten HTTP Server bei Bedarf anbietet. So kann die entwickelte Anwendung direkt gestartet werden.
Beim Microservice BL wurde zusätzlich noch das Spring Projekt *Spring Data Neo4j* genutzt, welches auf der Neo4j-ORM (Object-Graph-Mapping) Bibliothek basiert und damit ein CRUD-Interface für die Entiäten bereit stellt.
Als Build-Tool dient *Gradle*, was ebenfalls nach dem Programmierparadigma „convention over configuration“ entwickelt wurde und ohne XML-Konfigurationen auskommt. Die Projekt-Verzeichnisstruktur blieb dabei unberührt und wurde von Apache Maven übernommen. So bekommt man eine verständliche Standardkonfiguration, die nur in Ausnahmefällen angepasst werden muss.

### 4.2.4 Persistenz: NoSQL-DB

Zur Speicherung und Verwaltung der medizinischen Daten wird eine NoSQL-Datenbank zum Einsatz kommen, die graph-basiert arbeitet und eine REST Schnittstelle nach außen anbietet. Graph-Datenbanken können grundsätzlich in zwei unterschiedlichen Modi betrieben werden: *embedded-mode* oder *server-mode*. Für diese prototypische Umsetzung soll die Graph-Datenbank als Server laufen, der selbst in einem Docker-Container gestartet wird.

[2] http://www.php.net

# 5 Realisierung

In diesem Kapitel wird die Umsetzung einzelner wichtiger Komponenten für die zuvor konzipierte Microservice-Architektur dargelegt und auf eventuelle Schwierigkeiten bei der endgültigen Implementierung eingegangen. Codebeispiele werden Inline erklärt, Codesegmente, die nicht richtungsweisend sind (Getter, Setter und kleinere Konfigurationen oder Kommentare), werden der Übersicht halber nicht mit aufgeführt.

## 5.1 Übersicht über die Entitäten

Die Realisierung der Microservices war durch die Nutzung der Spring Projekte *Spring Boot* und *Spring Data Neo4j* relativ simpel. Beide Microservices nutzen den eingebetteten Tomcat-Webserver von *Spring Boot* und bieten eine REST-Schnittstelle an. Warum auch eine REST-Schnittstelle beim Front-End nötig war, wird am Ende dieses Abschnittes erläutert.

Am Anfang wird eine Main-Klasse (Application.java) erstellt, die folgenden Aufbau hat:

```
1 @SpringBootApplication
2 @EnableNeo4jRepositories
3 public class Application {
4
5         public static void main(String[] args) {
6                 initLog4j();
7                 SpringApplication.run(Application.class, args);
8 }
```

Die Annotation `@SpringBootApplication` entspricht den Spring Annotations (@ComponentScan, @Configuration und @EnableAutoConfiguration) und startet die automatische Konfiguration und die Durchsuchung nach Komponenten unterhalb der Main-Klasse (sub-packages).
Die Annotation `@EnableNeo4jRepositories` startet die Durchsuchung nach Repositories innerhalb des Projektes und ermöglicht deren Einsatz.
Die nicht näher im Code erläuterte Methode `initLog4j()` startet das Logging, welches für Fehler und Wartungsarbeiten unerlässlich ist. Das Log-Protokoll lässt sich über *localhost:7070/log* jederzeit einsehen.

Damit Entitäten später als Knoten in der Graphdatenbank gespeichert werden können, wurden sie folgendermaßen implementiert. Als Beispiel wird die Entität „Medication“ erläutert:

```
@NodeEntity
public class Medication {

        @GraphId
        private Long id;

        private Integer medicationId;

        @Convert(DateConverter.class)
        private Date date;

        @Relationship(type="PART_OF", direction = Relationship.INCOMING)
        private List<Drug> drugList;

        public Medication() {
        }
}
```

Die Annotation `@NodeEntity` spezifiziert, dass es sich um eine Knoten-Entität handelt. Das Attribut ID muss den Namen *id* tragen und mit der Annotation `@GraphId` versehen werden. Diese ID wird von der Application selbst vergeben und garantiert, dass keine doppelten IDs vergeben werden. Die Annotation `@Convert` ermöglicht es, eigene Konvertierungen des zugehörigen Attributs vorzunehmen. In der Klasse *DateConverter* befinden sich zwei Methoden, die das Datumformat zwischen den Java-Typen *Date* und *String* konvertieren. Diese Option wurde hier benutzt, um in der Graphdatenbank eine verschlankte Form des Datumsformats anzuzeigen. Ohne die Konvertierung wurde beim Speichern immer eine Uhrzeit hinzugefügt: Beispiel: Input-String: „1992-09-14“ -> Neo4j-DB: Sep 14, 1992 2:00:00 AM
Mittels der Annotation `@Relationship` wird die Beziehung zwischen zwei Entitäten spezifiziert. Die Parameter *type* und *direction* ermöglichen eine Anpassung des Namens der Relation und der Richtung, in der sie gelten soll. In diesem Fall sind eine Liste von Medikamenten (engl.: *drug*), Teil (PART_OF) der aktuellen Medikation des Benutzers. Der leere, öffentliche Konstruktor `public Medication()` ist nötig, damit der Object-Graph-Mapper der Bibliothek das Objekt erzeugen kann.

Im Anschluss an die Entität, wird das zugehörige Respository, in Form eines Interfaces implementiert. Es erweitert dabei das im Spring Data Neo4j Projekt enthaltende *GraphRepository<T>*.

```
public interface MedicationRepository extends GraphRepository<Medication> {

```

```
3         Medication findByMedicationId(Integer medicationId);
4
5         Medication findByDate(Date date);
6
7         @Query(MATCH (n:Medication) + RETURN n + ORDER BY n.date DESC)
8         List<Medication> findOrdered();
9 }
```

Das *GraphRepository* stellt bereits grundlegende CRUD-Operationen bereit. Eine Erweiterung dieser Operationen erfolgt durch die Einhaltung bestimmter Syntaxregeln. Möchte man z.B. eine „Medication“ anhand seiner *medicationId* oder anhand eines anderen Entitäten-Attributes in der Datenbank finden, ergänzt man bei der Deklaration *findBy...* um den gewünschten Feldnamen. Mit der Annotation `@Query` ist es möglich, die Anfragensprache Cypher zu benutzen, um individuellen Rückgaben aus der Graphdatenbank zu erhalten. Die deklarierte Methode *findOrdered()* sucht zunächst alle Knoten (n) mit dem Label „Medication“ und gibt eine absteigende Liste (sortiert nach dem Datum) zurück.

Damit ein Zugriff auf die Daten von außen erfolgen kann, erhält jede Entität einen Controller. Exemplarisch wird der Aufbau anhand des *PersonController* und der *Route* zur Erstellung einer neuen Entität dargestellt:

```
1 @CrossOrigin()
2 @RestController()
3         public class PersonController {
4
5         @Autowired
6         private PersonRepository personRepository;
7
8         * Creates a Person
9         * @param jsonPerson (see yaml, to get detailed information)
10        * @return Person with Graph-Id (key: id)
11        */
12        @RequestMapping(value = "/persons", method = RequestMethod.POST)
13        ResponseEntity<Person> addPerson(@RequestBody Person jsonPerson) {
14        try {
15                personRepository.save(jsonPerson);
16                Application.logger.debug("POST_/persons_saved_to_Neo4j");
17                return new ResponseEntity<>(jsonPerson, HttpStatus.CREATED); //201
18        } catch (Exception ex) {
19                Application.logger.debug("Exception_in_POST_/persons:"+ex);
20                return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR); //500
21        }
22        }
23 }
```

Die Annotation `@CrossOrigin()` spezifiziert, dass der Controller auch aus anderen Domänen erreichbar ist. Dieses ist nötig, damit die JavaScript-Calls der GUI den Controller erreichen können. Mittels der Annotation `@RestController()` wird diese Klasse als REST-Controller erkannt und beim Starten der Applikation initialisiert.
Die Annotation `@Autowired` ermöglicht hier die Umsetzung des Entwurfsmusters *Dependency Injection*. Das benötigte Repository wird damit zur Laufzeit zur Verfügung gestellt. Um Web-Anfragen auf bestimmte Händler-Methoden mappen zu können, erhält die jeweilige Methode die Annotation `@RequestMapping()`, die Parameter *value* und *method* dienen zur Definition des Pfades bzw. der HTTP Methode. Bei einem Aufruf der Methode sorgt die Annotation `@RequestBody` dafür, dass mittels *HttpMessageConverter* das JSON im Body der HTTP-Anfrage an das Java-Objekt (Person jsonPerson) gebunden wird.
Ist die Datenstruktur korrekt, wird eine neue Knoten-Entität mit dem Label „Person“ in der Graphdatenbank gespeichert. Der Rückgabewert enthält neben dem JSON der Anfrage den HTTP-Status 201. Bei einem strukturellen Fehler im Body der Anfrage wird der HTTP-Status Code 400 gesendet. Interne Fehler werden mittels *catch block* abgefangen und mit dem HTTP-Status Code 500 beantwortet.

Zu allerletzt wird mittels der Datei *application.properties*, die sich im Ordner „resources“ oberhalb der anderen Klassen befindet, die Anwendung konfiguriert:

```
1 spring.data.neo4j.username=neo4j
2 spring.data.neo4j.password=docker123
3 spring.data.neo4j.uri=http://neo4j:7474
4
5 server.port=7070
6
7 spring.jackson.default-property-inclusion: NON_NULL
```

Hier werden genauere Angaben bezüglich der verwendeten Datenbank, des Webservers und weitere Einstellungen vorgenommen. Im oberen Teil erfolgt die Definition von Benutzername, Passwort und der URI zur Neo4j Datenbank. Da alle Microservices später als Dockercontainer laufen, wird bei der URI keine IP-Adresse, sondern der Name des Services, der mittels *docker-compose* gestartet wird, angegeben.
Mittels `server.port` wird der Port des Webserver festgelegt. Durch den Wert `NON_NULL` wird der Jackson-Object-Mapper so konfiguriert, dass bei der Umwandlung von Java-Objekten ins JSON-Format *null* Werte ignoriert und nicht mitgesendet werden.

Alle bisherigen Codebeispiele beziehen sich auf den Microservice BL, der die Businesslogik enthält. Wie in der Einleitung dieses Kapitels bereits erwähnt, wird nun nochmal auf eine

Besonderheit innerhalb der Struktur des Microservice GUI eingegangen:
Auch dieser Baustein der Anwendung besitzt eine REST-Schnittstelle, die allerdings nur durch interne Methoden im JavaScript-Code `$.ajax()` aufgerufen wird. Der *ResolverController* leitet Anfragen an die BL weiter und gibt die *Response* an den AJAX-Aufruf zurück.

```
var RESOLVER = "/resolver";
var url = RESOLVER + "/diseases";
$.ajax({
        type: "GET",
        url: url,
        statusCode: {
        200: function (data) {
                for (i = 0; i < data.length; i++) {
                var temp = data[i];
                var dateTemp = new Date(temp.date);
                addItemWithButton("col-dis", temp.name, temp.name, temp.diseaseId);
                }
        },...
});
```

Dieses wurde während der Entwicklung der Anwendung nötig, da jQuery bzw. JavaScript nicht in der Lage ist, Service-Namen der Dockercontainer auf IP-Adressen zur Laufzeit zu mappen. Der nächste Inline-Code zeigt, wie der obige AJAX-Call vom *ResolverControler* an die BL weitergeleitet wird:

```
@CrossOrigin()
@RestController
public class ResolverController {

  private String uriBL = "http://logic:7070/";

  @RequestMapping(path = "/resolver/{route}", method = RequestMethod.GET)
  public ResponseEntity<Object> resolverGET(@PathVariable("route") String route) {
          try {
            uri = uriBL + route;
            RestTemplate rt = new RestTemplate();
            ResponseEntity<Object> response = rt.getForEntity(uri, Object.class);
            return new ResponseEntity<>(response.getBody(), response.getStatusCode());
          } catch (Exception ex) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND); //404
    }
}
```

Die Basis-URI der Geschäftslogik `http://logic:7070/` enthält keine IP-Adresse und wird zur Laufzeit auf die aktuelle IP-Adresse gemappt. Um die Anfrage durchzuführen, wird die Klasse `RestTemplate` genutzt. Der *body* des Rückgabewertes wird dann an den ursprünglichen AJAX-Call zusammen mit dem HTTP Status Code zurückgegeben.

## 5.2 Abhängigkeiten zwischen den Entitäten

Wie die Abbildung 4.2 zeigt, sind die Abhängigkeiten zwischen den einzelnen Entitäten relativ hoch. Dieses wird besonders bei der Entität „Letter“ deutlich, die einen Arztbrief repräsentiert. Sie beinhaltet alle Entitäten direkt oder indirekt. Somit wurde das Entwurfsprinzip der losen Kopplung kaum umgesetzt.

## 5.3 Einsatz von Value-Types und Enums

Beim Einsatz von objektorientierten Programmiersprachen wie Java wird häufiger das Verwenden von Wertetypen vernachlässigt und alles wird als Objekt betrachtet. *Value-Types* zeichnen sich dadurch aus, dass sie keine Methoden zum Verändern der eigenen Werte beinhalten Ritterbach (2011). Ein klassisches Beispiel für Werte-Typen wären z.B. ein Geldbetrag oder ein Zeitraum.
Um spätere Komplikationen oder Fehler zu vermeiden gibt es zahlreiche Pattern, wie und wann Werte-Typen einzusetzen sind Riehle (2006). In dieser Applikation wurde zur Repräsentation von „Laborwerten“ und „Geldbeträgen“ der *Value-Type* „Amount“ implementiert.
Um Wochentage, Status von Rechnungen oder Einheiten (engl.: *Unit*) abzubilden, wurde die Java-Klasse *Enum* verwendet. Damit wird eine hohe Typsicherheit und Übersichtlichkeit gewährleistet.

## 5.4 Front-End: HTML5, jQuery und Bootstrap

Das Benutzer-Interface der Anwendung wird durch den Microservice GUI bereitgestellt. Dafür wurde die fünfte Fassung der Hypertext Markup Language (HTML5) in Kombination mit der Skriptsprache JavaScript (JS) eingesetzt. *HTML5* wird heute als Standard bei der Umsetzung von Websites angesehen Freeman (2012).
Die JavaScript-Bibliothek *jQuery* und das Framework *Bootstrap* in der neuesten Version 4 vereinfachten den Einsatz von JS und dem Styling der Webseite mittels CSS. Somit konnte relativ einfach ein *responsive design* ermöglich werden, das sich allen Endgeräten (Desktop PC, Tablet und Handy) bei der Darstellung dynamisch anpasst. Zusätzlich konnten ein zeitgemäßes Design erzielt werden und auch komplexere Strukturen wie Navigationsleisten ohne großen Aufwand implementiert werden.
Auf die Besonderheiten beim Aufruf der REST-Schnittstelle des Microservices BL wurde im Abschnitt 5.1 genauer eingegangen.

## 5.5 Neo4J

Zur Graph-basierten Speicherung der Daten wurde die NoSQL-Datenbank *Neo4j* gewählt. Diese wird im Server Modus in einem eigenen Docker-Container gestartet. Neben einer umfangreichen REST API bietet diese Datenbank eine Graph-Visualisierung mittels eines herkömmlichen Internet Browsers. Unter der IP-Adresse: *localhost:7474* erscheint ein User-Interface, in dem der Benutzer sich die einzelnen Entitäten und Zusammenhänge graphisch darstellen lassen kann. Erfahrenere Anwender können sich dort direkt mittels der Anfragensprache *Cypher* verschieden Teilaspekte der Daten visualisieren lassen. Beim erstmaligen Aufruf im Browser müssen die hinterlegten Benutzer- und Passwortdaten angegeben werden, um eine visualisierte Sicht auf die Daten zu erhalten.

## 5.6 Docker

Die Anwendung wurde auf zwei Rechnern mit unterschiedlichen Windows Betriebssystemen entwickelt. Daher kamen sowohl *Docker for Windows*, als auch die *Docker Toolbox* zur Containerisierung zum Einsatz.

Jeder Service benötigt ein Dockerfile, welches Konfigurationsparameter für den Bau des Docker Images beinhaltet. Hier exemplarisch das Dockerfile des Microservice BL:

```
1 FROM openjdk:8-jdk-alpine
2 ADD build/libs/medport.logic-1.0.0.jar /app.jar
3 ENTRYPOINT exec java -jar /app.jar
```

Mit dem Docker Kommando `FROM` beginnt das File, es definiert das Basis-Image und startet den Build Prozess. Für den Microservice BL wurde das `openjdk` Image genutzt. Das `ADD` Kommando erhält zwei Argumente: einen Quellpfad und einen Zielpfad. Dadurch wird die vorher gebildete Jar, in das angegebene Container-Verzeichnis kopiert. Der Shell-Befehl *exec java -jar /app.jar* wird als Argument an das Docker Kommando `ENTRYPOINT` übergeben und führt beim Starten des Containers die Jar aus.

Um mehrere Docker Images zu builden und anschließend zu starten, wird das bereits in Abschnitt 2.3 erwähnte Tool *Docker Compose* benutzt. Hierfür wird ebenfalls eine Konfigurationsdatei (**docker-compose.yml**) benötigt und welche sich normalerweise im Root-Verzeichnis des Software-Projektes befindet:

```
version: '3'
  services:
        gui:
          build:
            context: c:/Users/Sebastian/IdeaProjects/MedPort/medport_gui/
          ports:
                - "6060:6060"
          depends_on:
                - logic
        logic:
          build:
                context: c:/Users/Sebastian/IdeaProjects/MedPort/medport_logic/
          ports:
                - "7070:7070"
          depends_on:
                - neo4j
        neo4j:
          image: neo4j:latest
          environment:
                - NEO4J_AUTH=neo4j/docker123
          ports:
                - "7474:7474"
                - "7687:7687"
```

Nach Angabe der *docker-compose* Version ('3'), werden die `services` definiert: **gui**, **logic** und **neo4j**. Der Konfigurationsparameter `context` erlaubt es, den Pfad zum Dockerfile zu setzen oder auf ein git-Repository zu verweisen. Mittels `ports` kann bestimmt werden, auf welchen Ports die im Service laufenden Anwendungen von außen erreicht werden können.
Durch das Verwenden des Parameters `depends_on` wird beim Starten aller Dienste, eine bestimmte Reihenfolge garantiert. Bei dieser prototypischen Anwendung werden zunächst die Datenbank (neo4j), dann die Logik (logic) und letztendlich die Benutzeroberfläche (gui) durch *docker-compose* gestartet.
Für die Datenbank *Neo4j* existiert bereits ein einsatzfähiges Docker Image, auf welches mit dem Parameter `image` verwiesen wird. Durch den *tag* `lastest` wird die aktuellste Version aus dem Docker-Hub geladen. Letztendlich werden der gewünschte Benutzername und das Passwort durch den Konfigurationsparameter `environment` gesetzt.

Im Anschluß an die Konfiguration kann beispielsweise mit dem Gradle-Befehl `gradlew build` die benötigte Jar des jeweiligen Services generiert werden. Die Jar befindet sich dann im Verzeichnis `build/libs/`, aus der sie später in das Container-Verzeichnis kopiert wird. Nach dem Build der Jar können alle Container mit dem Befehl `docker-compose up - -build` gestartet werden. Die Option `- -build` erzwingt den Build des jeweiligen Image und garantiert so, dass die zuletzt generierte Jar als Grundlage für das Image dient. Ohne diese Option würden nur

gänzlich fehlende Images erzeugt werden.
Das erstellte System kann mit dem Befehl `docker-compose down` wieder gelöscht werden. Alle Container und erstellten Netzwerke werden dadurch entfernt, die zuvor in der Datenbank gespeicherten Werte daher ebenfalls.

# 6 Evaluierung

Nach Abschluss der Implementierung aller Microservices und dessen Betrieb durch Docker, wird in diesem vorletzten Kapitel betrachtet, welche Vorgaben der Spezifikation umgesetzt wurden und welchen Stellenwert dabei das Testen hatte.

## 6.1 Erfüllung der Anforderungen

Im 3. Kapitel wurden zahlreiche Anforderungen an den Prototypen definiert und spezifiziert. Die folgende Tabelle gibt zunächst eine groben Überblick, welche *User Stories* umgesetzt wurden. Es werden nur die *Epics* der *User Stories* dargestellt, um eine kompaktere Form zu erhalten. In der Spalte "nicht erfüllt„ werden die *Stories* aufgelistet, die nicht umgesetzt wurden:

| Rolle | Ziel/Wunsch | nicht erfüllt |
|---|---|---|
| Als Patient | möchte ich meine Arztbriefe verwalten | bearbeiten |
| Als Patient | möchte ich meine Erkrankungen verwalten | bearbeiten |
| Als Patient | möchte ich meine Medikamente verwalten | bearbeiten |
| Als Patient | möchte ich meine Untersuchungen verwalten | bearbeiten |
| Als Patient | möchte ich meine med. Rechnungen verwalten | - |
| Als Patient | möchte ich meine Daten graphisch anzeigen | - |
| Als Patient | möchte ich Daten importieren | - |
| Als Arzt | möchte ich Patientendaten graphisch anzeigen | - |
| Als Versicherer | möchte ich Patientendaten graphisch anzeigen | - |

Tabelle 6.1: User Stories (Epics) - Überblick bezüglich Umsetzung

Die vier vorher spezifizierten *Use Cases* wurden komplett umgesetzt. Damit erfüllt die Implementierung fast vollständig die geforderten Anwendungssituationen. Auch die Umsetzung eines plattformunabhängigen Betriebes durch Docker wurde erreicht. Die Daten werden wie gefordert in einer graphischen Datenbank gespeichert und können graphisch in jedem handelsüblichen Browser angezeigt werden.
Die kritische Auseinandersetzung mit den einzelnen Komponenten erfolgt im abschließenden Kapitel „Fazit und Ausblick".

## 6.2 Testen während der Entwicklung

Beim Softwaretesten unterscheidet man, neben zahlreichen anderen Verfahren, das statische und das dynamische Testen. Der grundlegende Unterschied liegt darin, dass beim statischen Testen das Programm nicht ausgeführt wird, sondern der Quellcode selbst als Testobjekt dient. Während der Entwicklung wurde nur das dynamische Testen durchgeführt. Dabei kamen vor allem JUnit-Tests zum Einsatz. *JUnit*[1] ist ein Test-Framework für Java-Applikationen, welches zur Testautomatisierung dient.
Nach Erstellung der REST-Spezifikation mittels *Swagger* wurden die Klassen implementiert und im Anschluss einfache JUnit-Test geschrieben. Diese werden vor dem Build-Prozess vom Spring Framework automatisch ausgeführt.
Der folgende Inline-Code zeigt den generellen Aufbau eines JUnit-Tests in Spring Boot:

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = Application.class)
public class PersonTests {

        Person person;

        @Before
        public void setUp(){
                person = new Person(1, "John", "Doe");
        }

        @Test
        public void testSetter() {
                person.setFirstName("Chris");
                String resultExpected = "Chris";
                assertEquals(resultExpected, person.getFirstName());
        }
}
```

Die Annotation `@RunWith()` spezifiziert, welcher *Runner* für die Durchführung der nachfolgenden Tests genutzt werden soll. Mittels der `@ContextConfiguration()` Annotation und dem Attribut `classes` wird die Konfigurationsklasse bestimmt.
Bevor die eigentlichen Tests innerhalb der Klasse ausgeführt werden, definiert man eine Setup-Methode, die dann durch die Annotation `@Before` vor *jeder* Testmethode ausgeführt wird.
Alle Methoden die einen Test beinhalten, werden mittels `@Test` gekennzeichnet.
Die in der JUnit-API enthaltende Klasse *Assert* beinhaltet unter anderem die Methode `assertEquals()`, um erwartete und tatsächliche Ergebnisse auf Gleichheit zu prüfen.

[1] https://junit.org

Im Anschluss folgten die Integrationstests. Um das reine Speichern und Lesen von Entitäten zu überprüfen, wurde lokal eine Neo4j-Datenbank aufgesetzt. Die Zugangsdaten und URI wurden in einer zusätzlichen *application.properties* Datei hinterlegt. Damit konnte zunächst auf den Einsatz von Docker verzichtet werden.
Der folgende Inline-Code zeigt exemplarisch einen Integrationstest, wobei die oben bereits beschriebenen Annotationen nicht abgebildet sind:

```
1  public class PersonRepositoryTest {
2
3          private Person paul, kelly;
4          Integer paulId, kellyId;
5
6          @Autowired
7          private PersonRepository personRepository;
8
9          @Before
10         public void setUp() {
11           paul = new Person(5,"Paul", "Doe", createDateFromString("1964/01/25"));
12           personRepository.save(paul);
13           kelly = new Person(6,"Kelly", "Doe", createDateFromString("1976/02/15"));
14           personRepository.save(kelly);
15         }
16
17         @Test
18           public void testFindById() {
19           Person found = personRepository.findByPersonId(5);
20           String expected = "Paul";
21           assertEquals(expected, found.getFirstName());
22         }
23 }
```

Nachdem zwei neue Knoten-Entitäten in der Graph-Datenbank gespeichert wurden, wird im anschießenden Test die Methode *findByPersonId()* überprüft.
Bei diesen Tests fiel zum ersten mal auf, dass beim Lesen der Entitäten einige Attribute nicht direkt zur Verfügung standen. Beinhaltet eine *Untersuchung* z.B. eine Liste mit *Krankheiten*, kann dass Attribut *diagnosedBy* nicht mehr direkt ausgelesen werden, da es im ursprünglichen Rückgabewert (JSON) nicht vorhanden ist. Um diese Information zu erhalten, muss die gewünschte Krankheit direkt aus der Datenbank abgerufen werden.
Dieses führte anfänglich zu zahlreichen *null-Pointer Exceptions*, da der Zugriff auf tiefer liegende Informationen nicht möglich war. Dadurch konnte eine wichtige Erkenntnis für die weitere Umsetzung der Anforderungen gewonnen werden.

Parallel dazu wurde das Programm *Postman*[2] benutzt, um manuelle Integrationstests durchzuführen. Jeder neu implementierte Controller in der Business-Logik wurde im Anschluss per *Postman* aufgerufen und die Rückgabewerte (JSON, HTTP-Status) überprüft. Die so gespeicherten Anfragen konnten im späteren Verlauf erneut benutzt werden, um einzelne Funktionen in der Docker-Umgebung erneut zu testen.

Da die Fehlersuche beim Ausführen von JavaScript-Code sehr mühsam sein kann, wurden die Ajax-Calls erst implementiert, nachdem mittels *Postman* sichergestellt war, dass der Rückgabewert korrekt ist. Parallel wurden alle Aufrufe des Resolver-Controllers mittels *Log4j*[3] geloggt. Dieses war vor allem im späteren Docker-Betrieb von großem Vorteil, da das Log-Protokoll jederzeit mit der Route „/log“ eingesehen werden konnte.

[2] http://www.getpostman.com
[3] http://www.logging.apache.org/log4j

# 7 Fazit und Ausblick

Innerhalb weniger Monate konnte eine prototypische Anwendung für das Verwalten von medizinischen Gesundheits- und Abrechnungsdaten konzipiert und implementiert werden. Viele der geforderten Funktionen konnten dabei umgesetzt werden. Zu beachten ist, dass eine „echte“ Anbindung an Fremdsysteme, die den Import von medizinische Daten ermöglichen, nicht möglich war und diese simuliert wurde.
In dem Unterkapitel „Bewertung“ werden verschiedene technische und architektonische Aspekte sowie Vergleiche mit bereits auf dem Markt befindenden Produkten diskutiert.

## 7.1 Bewertung

Bei der Bewertung werden zwei grundlegende Aspekte genauer beleuchtet und kritisch hinterfragt. Dabei stehen einmal die technische Umsetzung und das Architekturdesign im Vordergrund, anschließend der Vergleich mit anderen Gesundheits-Apps wie *LifeTime* und *HealthVault*.

### 7.1.1 Bewertung von Design und genutzter Technologien

Einer der wichtigsten Eigenschaften einer Software sind **Wartbarkeit** und **Erweiterbarkeit**. Hinzu kommt der Trend, Software in der *Cloud* zu betreiben. Dafür eignet sich derzeit die Microservice-Architecture mit am besten, sie gilt als *cloud-native* Taibi (2017). Zahlreiche Unternehmen versuchen derzeit ihre „alte“ Software in Microservices zu überführen. Damit findet ein Umdenken in vielen großen Unternehmen statt, der frühere Goldstandard SOA wird zunehmend verdrängt Cerny (2017).
Betrachtet man diese Entwicklung, war die Wahl der Architektur zeitgemäß. In wieweit bei der Umsetzung die Vorteile dieses Architekturstiles umgesetzt werden, wird nun hinterfragt.

Die drei Grundideen von Microservices sind folgende Wolff (2016):
- Jeder Service sollte nur *eine* Aufgabe haben, die aber komplett umsetzen
- Services sollen in der Lage sein, zusammen arbeiten zu können

- Es sollte ein universelle Schnittstelle benutzt werden
Alle diese Kriterien werden erfüllt. Der Microservice GUI kümmert sich um die Darstellung von medizinischen Gesunheits- und Abrechnungsdaten, der Microservice BL ermöglicht (mit Hilfe der Graphdatenbank Neo4j) das Verwalten der Daten.
Alle Services können unabhängig von einander entwickelt und deployed werden, laufen also als eigene Prozesse. Eine Skalierung der einzelnen Services ist so jederzeit gegeben, Release-Zyklen können kurz gehalten werden. Die Devise *smart endpoints and dumb pipes* wurde umgesetzt und ein dezentrales Datenmanagement möglich gemacht. Die universelle Schnittstelle REST ermöglicht einen einfachen Datenaustausch zwischen den einzelnen Komponenten, der ohne Middleware funktioniert.

Auch die SOLID-Entwurfs-Kriterien sollten möglichst bei der Planung und Entwicklung der Software beachtet werden. Dabei wurden das ***S**ingle Responsibility Principle* und ***I**nterface Segregation Principle* relativ gut umgesetzt. Jede Komponente hat *eine* Verantwortung, wie bereits oben beschrieben und die Schnittstellen sind schlank gehalten. Keine Klasse muss daher unnötige Methoden implementieren. Auch das Entwurfsprinzip *Dependency Injection* wurde durchgehend benutzt. Somit sind Abhängigkeiten an einem zentralen Ort realisiert worden.
Das Entwurfspattern *Low Coupling* konnte hingegen an einigen Stellen kaum oder gar nicht umgesetzt werden. Besonders die Klasse „Letter“ , die eine Arztbrief repräsentiert, ist hierfür ein Negativbeispiel. In ihren Attributen sind alle anderen Klassen enthalten. Daher führt jede Änderung einer anderen Klasse ggf. zu einer Anpassung dieser einen.

Die eingesetzten Technologien, wie die Spring Projekte *Spring Boot* und *Spring Data* ermöglichten eine zügige Umsetzung der spezifizierten Vorgaben. Nach Einbindung mittels Gradle konnten schnell die benötigten Controller, Repositories und Klassen implementiert werden. Durch die Wahl dieses quelloffenen Frameworks war man auf die Programmiersprache Java festgelegt, welche sich während der Entwicklung aber als zielführend herausstellte, auch durch die zahlreichen vorhandenen Bibliotheken.

Die Containervirtualiserung ist seit Jahren auf dem Vormarsch, insbesondere durch den Einsatz von *Docker* Julian (2016). Während der Entwicklung zeigte sich die Performance sowohl der *Docker Toolbox* auf dem Windows 8.1 Laptop, als auch *Docker for Windows* auf dem Windows 10 PC als gut und stabil. Es mussten beim Wechsel der Entwicklungsumgebung keine Anpassungen an das jeweilige Betriebssystem vorgenommen werden.
Während des Betriebes verbrauchte *Docker for Windows* nur ca. 60 MB Arbeitsspeicher. Alle

Docker-Images zusammen benötigten ca. 500 MB Festplatte. Dabei ist gewisser Overhead nicht zu vermeiden, da jeder Container seine eigene Java Virtual Machine (JVM) starten muss, damit die in Java programmierten REST-Controller und die Funktionen bereitgestellt werden können. Ein Punkt der noch nicht gelöst wurde, ist die Persistenz der Daten. Derzeit werden alle gespeicherten Entitäten in der Graphdatenbank beim Herunterfahren der Container unwiderruflich gelöscht. Eine mögliche Lösung wäre der Einsatz von *Volumes*, dabei legt *Docker* eine Datenbank auf dem Host-System an und wird beim Start des Containers eingehängt.

Die gewählte Graphdatenbank *Neo4j* erfüllte ihren Zweck, auch wenn einige gewünschte Funktionen fehlten. Der große Vorteil war zunächst, dass diese NoSQL-Datenbank bereits als fertiges Docker-Image aus dem Docker-Hub geladen werden konnte. Somit war ihr Einsatz innerhalb einer Container-Virtualisierten-Umgebung nach kurze Einarbeitung denkbar einfach. Ein weiterer Vorteil lag darin, dass *Neo4j* bereits eine eigene Benutzeroberfläche integriert hatte. Damit konnte die Realisierung einer graphischen Präsentation der Daten leicht erfüllt werden.
Negativ fiel während der Entwicklung auf, dass beim Aufruf des Neo4j-Browsers keine Cypher-Query als Parameter mit übergeben werden konnte. Ein unschöner Workaround, bei dem der User die Query selber Einfügen muss, wurde notwendig. Die Akzeptanz durch den Benutzer scheint hier mehr als fraglich.
Eine weitere Problematik trat auf, nachdem längere Texte in einem Knotenattribut abgelegt wurden. Bei der Darstellung im Browser führte dieses zu extrem langen Scrollen, wollte man sich die einzelnen Attribute eines solchen Knotens anschauen.

Abschließend kann man sagen, dass die gewählten Technologien insgesamt ihre Aufgabe erfüllt haben. Kritisch muss man anmerken, dass für die Entwicklung keine vollständige *Deployment Pipeline* aufgebaut wurde. Dieses wäre z.B. mit dem Einsatz von *Jenkins*[1] möglich gewesen.

### 7.1.2 Vergleich mit anderen Gesundheits-Apps

Für den Vergleich wurden das Programm *LifeTime*[2] der Firma connected-health.eu GmbH und die Gesundheitsplattform *HealthVault*[3] von Microsoft analysiert und bewertet. Bewertungskriterien waren dabei: Datenhaltung, Verschlüsselung, Backup, Importmöglichkeiten,

[1] http://www.jenkins.io
[2] http://www.lifetime.eu
[3] http://www.international.healthvault.com

Präsentation der Daten und mögliche Zusatzfunktionen. Die Tabelle 7.1 zeigt einen tabellarischen Überblick.

Kontrolle über die eigenen Daten zu haben, ist bei medizinischen Informationen besonders wichtig. Damit ist auch gemeint, wer diese Daten einsehen und bearbeiten kann. Ziel der prototypischen Anwendung war es, einen Datenaustausch *direkt* mit Praxen und Krankenhaus-Servern zu erreichen. Diesen Ansatz vertritt auch die Applikation *LifeTime*, sensible medizinische Daten sind nur für mehrere Stunden temporär auf dem sogenannten *LifeTime-Hub* zwecks Datenübermittlung gespeichert.
Microsoft verfolgt einen anderen Ansatz. Das Unternehmen speichert zentral alle medizinischen Daten auf Microsoft-Servern. Wo diese Server sich befinden wird auf der Homepage nicht genau erläutert. Die HealthVault-Server befünden sich aber in kontrollierten Einrichtungen[4].
Damit zeigt sich eine zweigeteilte Lösung. Der Prototyp und die in Deutschland entwickelte App speichern nur lokal die Daten - auch, um eine Zweckentfremdung durch Dritte (bzw. durch das Unternehmen selbst) auszuschließen. Microsoft hingegen möchte es dem Anwender so bequem wie möglich machen, auch, weil viele Daten von anderen Microsoft-Produkten mit benutzt werden sollen. Dieses ist nur durch eine zentrale Haltung der Daten möglich und „bequemer“ für den Endanwender.
Liest man das nur in englischer Sprache erhältliche *Privacy Statement*[5] von Microsoft, wird dort häufig betont, wie sicher die Daten vor dem Zugriff Dritter seien, ob Microsoft selbst die Daten nutzt oder auswertet, bleibt allerdings etwas unklar.

Sowohl *LifeTime*, also auch *HealthVault* verschlüsseln die medizinischen Daten. *LifeTime* ließ sich dieses sogar durch das europäische Siegel *ePrivacy*[6] zertifizieren. Der Prototyp verzichtet derzeit auf eine Verschlüsselung, da dieses nicht im Fokus bei der Umsetzung stand.
Auch ein Backup der Daten ist derzeit noch nicht möglich. Diese Option wird für die beiden anderen Produkte bereits angeboten, wobei *LifeTime* ein lokales Backup auf dem PC ermöglicht. Da Microsoft alle Daten zentral hält ist die Wiederherstellung der Daten recht simpel.

Einer der wichtigen Funktionen ist der Datenimport. Dieser sollte möglichst einfach, sicher und vertrauenswürdig sein. Alle Programme ermöglichen das Anlegen von medizinischen Daten per Tastatur, wobei bei *HealthVault* vor allem die manuelle Eingabe von Vitalfunktionen, Gewicht oder Notfalldaten im Vordergrund steht. *LifeTime* ermöglicht auch das Digitalisieren

[4] https://account.healthvault.com/help/de-DE/Content/Topics/PrivacyAndSecurity.htm
[5] http://www.privacy.microsoft.com/en-us/privacystatement/
[6] http://www.eprivacy.eu

von Befunden per Foto. Diese Funktion wurde ebenfalls für den Prototyp in Erwägung gezogen, aus zeitlichen Gründen allerdings nicht umgesetzt.
Beim automatischen Datenimport gehen alle Anwendungen unterschiedliche Wege. Der Prototyp war so konzipiert, dass er für verschiedene Fremdsysteme jeweils eine passende Schnittstelle implementiert. *LifeTime* setzt auf ein proprietäres System. Die jeweilige medizinische Einrichtung muss eine Desktop-Anwendung installieren, die dann entweder per Internet oder direkt in der Praxis die Daten mittels Hub synchronisiert.
*HealthVault* bietet zahlreiche Möglichkeiten für den automatischen Datenimport. Microsoft favorisiert eine direkte Anbindung der jeweiligen medizinischen Einrichtung an die Plattform selbst. Somit werden die Daten direkt in den Account des Anwenders importiert und beim nächsten Start der App direkt angezeigt. Vitalparameter können per zusätzlicher Software, auch von Fitness-Gadgets importiert werden.
Beim Entwicklern und Testen des Prototypens wurde schnell deutlich, dass ein automatischer Datenabgleich für die Akzeptanz und vor allem für die Usability von enormer Wichtigkeit ist. Wer möchte schon seinen kompletten Entlassungsbericht aus dem Krankenhaus per Tastatur in das Programm einpflegen? Auch um eventuelle Tippfehler zu vermeiden, was gerade bei der Eingabe von Medikamentendosen von großer Tragweite sein kann, sollte der automatische Datenabgleich realisiert werden.

Die Präsentation der medizinischen Daten innerhalb von „Gesundheits-Apps“ ist meistens sehr ähnlich, nämlich in tabellarischer Form. Genau in diesem Punkt wollte sich der Prototyp signifikant absetzen. Neben der bekannten tabellarischen Aufbereitung der Daten sollte zusätzlich die Darstellung als Graph, einen spürbaren Mehrwert bieten. Dieser Ansatz ist bisher einzigartig.
Bei der Umsetzung gab es allerdings schnell erste Einschränkungen, die die Akzeptanz des zusätzlichen Features in Frage stellen könnte. Das Hauptproblem liegt darin, dass es beim Aufruf der *Neo4j* Oberfläche nicht möglich ist, eine Cypher-Query zu übergeben. Aktuell wird diese Problematik mit dem Workaround umgangen, dass dem Anwender die benötigte *Query* vor dem Öffnen der Graphen-GUI angezeigt wird. Diese per Copy&Paste selber auf der Oberfläche einfügen zu müssen, schränkt die Benutzerfreundlichkeit stark ein. Inwieweit diese Zusatzfunktion von wirklichen Nutzen sein kann, um z.B. Zusammenhänge besser zu verstehen oder Doppeluntersuchungen zu vermeiden, bleibt fraglich. Der Ansatz, die Daten auf eine neue Art graphisch aufzubereiten erscheint weiterhin spannend, ein wirklicher Nutzen ließe sich wahrscheinlich, aber nur in einem Feldversuch ermitteln.

Jede der Anwendungen stellt noch zusätzliche Funktionen bereit, die nicht direkt mit der Verwaltung von rein medizinischen Daten im Zusammenhang stehen. *LifeTime* bietet dem Anwender die Möglichkeit, seine Arzttermine per App zu machen. Microsoft sieht in der Anbindung von zahlreichen Gesundheits-Gadgets verschiedener Anbieter einen großen Mehrwert und zielt dabei auf den bereits in der Einleitung erwähnten Hype des *Quantified Self* ab.
Besonders für Privatpatienten wollte der Prototyp einen weiteren Mehrwert bieten. Ziel war es, dass Rechnungen über medizinische Leistungen vom Programm automatisch importiert oder manuell hinzugefügt werden können. Der Anwender kann so leicht sehen, welche Rechnungen noch nicht bezahlt bzw. erstattet wurden. Dieses Feature bietet keines der anderen beiden Programme und erscheint auf den ersten Blick praktikabel – sogar, wenn man die Rechnungen per Tastatur einpflegt.

| *Prototyp* | *LifeTime* | *HealthVault* |
|---|---|---|
| **Datenhaltung** | | |
| - lokal in der App | - lokal in der App<br>- LiftTime-Hub (temporär) | - lokal in der App<br>- MS-Server |
| **Verschlüsselung** | | |
| - keine | - ja, AES-256 | - ja |
| **Backup** | | |
| - nein | - ja (lokal) | - ja (MS-Server) |
| **Importmöglichkeiten** | | |
| - manuell<br>- automatisch | - manuell<br>- automatisch | - manuell<br>- automatisch<br>(per Zusatz-App) |
| **Präsentation der Daten** | | |
| - tabellarisch<br>- graphisch (als Graph) | - tabellarisch<br>- graphisch (als Foto) | - tabellarisch<br>- Excel-Charts |
| **zusätzliche Optionen** | | |
| - Rechnungsverwaltung | - Arztterminplanung | - Anbindung von Gadgets<br>- Notfalldaten<br>- Gewichtskontrolle<br>- Blutdruckkontrolle |

Tabelle 7.1: Vergleich des Prototypen's mit anderen Gesundheits-Apps

## 7.2 Ausblick

Die grundlegende Frage zu Beginn der Arbeit war, ob eine graphische Darstellung von medizinischen Daten einen Mehrwert für bestimmte Anwender bedingt. Dieses ließ sich nicht eindeutig belegen. Eine Feldstudie mit Patienten und medizinischen Personal bzw. Versicherern wäre hier hilfreich. Besonders, um die oben beschriebenen Einschränkungen bei der Usability besser abschätzen zu können.
Insgesamt hat der Prototyp noch Verbesserungspotential, besonders in Hinblick auf die Schnittstellen zu Klinik- und Praxisserver, sowie dem Erscheinungsbild der Benutzeroberfläche (Microservice GUI).
International hat sich der Standard HL7 (*Health Level 7*) für den elektronischen Austausch von medizinischen, administrativen und finanziellen Daten zwischen Informationssystemen im Gesundheitswesen durchgesetzt. Die deutsche Organisation „HL7 Deutschland e. V.“ hat in Anlehnung daran, eine ausführliche Spezifikation *Arztbrief Plus* Heitmann (2017) veröffentlicht. Vergleicht man dieses XML-Template mit der Implementierung im Prototypen, wird deutlich, wie viel komplexer die Abbildung eines solchen Dokumentes in elektronischer Form in Wirklichkeit ist.
Die Benutzeroberfläche bedient zwar einige heutige Standards, wirkt aber gerade im Vergleich zu den anderen vorgestellten Produkten sehr simpel und eintönig. Interessant wäre auch eine Implementation dieses Services mit einem größeren JavaScript oder PHP-Framework, um den derzeitig noch benötigten *ResolverController* und damit den Einsatz eines Spring-Projektes überflüssig werden zu lassen.
Spannend wäre abschließend die Frage, welche Erkenntnisse möglich wären, sollten mehrere hunderttausend oder Millionen User einen weiterentwickelten Prototypen nutzen. Es könnte ein flächendeckendes Bild von Erkrankungen in unserer Gesellschaft gezeigt werden, und vor allem, wie diese behandelt und diagnostiziert werden. Dies waren Informationen die sonst auf tausenden von Klinik- und Praxisservern verteilt vorliegen, und für *Big Data* Analysen in dieser dezentralen Form unbrauchbar sind.

# Literaturverzeichnis


[Allweyer 2008] Allweyer, Thomas ; Allweyer, Thomas (Hrsg.): *BPMN Business Process Modeling Notation.* Books on Demand GmbH, 2008

[Cerny 2017] Cerny, Tomas: Disambiguation and Comparison of SOA, Microservices and Self-Contained Systems. In: *RACS 17* (2017)

[Elmasri 2017] Elmasri, Ramez ; Goldstein, Matt (Hrsg.): *Database Systems.* Pearson, 2017

[Freeman 2012] Freeman, Eric ; Robson, Elisabeth (Hrsg.): *HTML5-Programmierung von Kopf bis Fuss: Webanwendungen mit HTML5 und JavaScript.* O'REILLY, 2012

[Freund 2014] Freund, Jakob ; Rücker, Bernd (Hrsg.): *Praxishandbuch BPMN 2.0.* camunda, 2014

[Gadatsch 2017] Gadatsch, Andreas ; Gadatsch, Andreas (Hrsg.): *Grundkurs Geschäftsprozess-Management.* Springer, 2017

[Heitmann 2017] Heitmann, Daniel H.: *Arztbrief Plus auf Basis der HL7 Clinical Document Architecture Release 2 für das deutsche Gesundheitswesen.* HL7 Deutschland (Veranst.), 2017

[Julian 2016] Julian, Spencer: Containers in Research: Initial Experiences with Lightweight Infrastructure. In: *ACM* (2016)

[Knoche 2016] Knoche, Holger: Sustaining Runtime Performance while Incrementally Modernizing Transactional Monolithic Software towards Microservices. In: *ICPE 16* (2016)

[Rad 2017] Rad, Babak B.: Cloud Computing Adoption: A Short Review of Issues and Challenges. In: *School of Computing Asia Pacific University of Technology and Innovation (APU)* (2017)

[Redmond 2012] Redmond, Eric ; Carter, Jacquelyn (Hrsg.): *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement.* The Pragmatic Programmers, 2012

[Riehle 2006] Riehle, D.: Value Object. In: *PLoP* (2006), S. 1 –6

[Ritterbach 2011] Ritterbach, Axel S.: *Werttypen in objektorientierten Programmiersprachen: Anforderungen an eine Sprachunterstützung.* 2011

[Robinson 2015] Robinson, Ian ; Webber, Jim (Hrsg.): *Graph Databases.* O'Reilly Media, 2015

[Rupp 2007] Rupp, Chris ; Rupp, Chris (Hrsg.): *Requirements-Engineering und Management.* Hanser, 2007

[Starke 2015] Starke, Gernot: *Effektive Softwarearchitekturen: Ein praktischer Leitfaden.* Hanser, 2015

[Taibi 2017] Taibi, Valentina L.: Microservices in Agile Software Development: a Workshop-Based Study into Issues, Advantages, and Disadvantages. In: *XP2017* (2017)

[Takai 2017] Takai, Daniel ; Takai, Daniel (Hrsg.): *Architektur für Websysteme: Serviceorientierte Architektur, Microservices, Domänengetriebener Entwurf.* Hanser, 2017

[Umbach und Metz 2006] Umbach, Hartmut ; Metz, Pierre: Use Cases vs. Geschäftsprozesse Das Requirements Engineering als Gewinner klarer Abgrenzung. In: *Informatik-Spektrum* 29 (2006), Dec, Nr. 6, S. 424–432. – ISSN 1432-122X

[Wolff 2016] Wolff, Eberhard ; Wolff, Eberhard (Hrsg.): *Microservices: Flexible Software Architecture.* Eberhard Wolff, 2016

*Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.*

Hamburg, 15. März 2018 Dr. Sebastian Diedrich