

Bachelorarbeit

**Konzeption, Realisierung und Evaluation
einer Patienten- und Medikamentenhandel-
Verwaltungssoftware für eine Klinik für Der-
matologie in Vietnam**

von

Nghiem Phan

Matrikelnummer: 7099115

im Studiengang Praktische Informatik

am Fachbereich Informatik

der Fachhochschule Dortmund

Erstgutachter: Prof. Dr. Sebastian Bab

Zweitgutachter: Prof. Dr. Robert Rettinger

Dortmund, den 26.08.2021

Abstract

This bachelor thesis discusses about software development processes for small to medium-sized clinics with a specific focus on patient administration and drug distribution system. In this regard, a case study is conducted working on developing a management system for a small clinic specialized in dermatology in Vietnam. Even though the system was built and customized upon specific business' requests, most of the software development principles are generally applicable and the architecture of the program is kept as generic as possible.

One of the main goals of this project is to collect customer's requirements to understand business partner's needs, upon which the most suitable software engineering processes and technologies will be applied. The actual architecture of the software is then presented by using UML diagrams in chapter 3. After that, the architecture will be implemented with the chosen technologies.

There are multiple stages of building the product. In the first step, a demo of the solution is presented to business partners to gather feedbacks, from which possible modifications will be suggested. Afterward, final product is repeatedly refined and delivered. Besides, the architecture and development process of the product will also be evaluated.

In the final chapter of this thesis, the software development process will be re-evaluated and possible improvements in the future for the software, not only from the technical but also from the business standpoint, will be discussed.

Kurzfassung

Im Rahmen dieser Arbeit wird das Verfahren der Softwareentwicklung einer Patienten- und Medikamentenhandel-Verwaltungssoftware für eine Klinik behandelt. Obwohl der Anwendungsfall auf eine kleine Klinik für Dermatologie in Vietnam beschränkt wird, sind die Methodik und Architekturen dieser Software überall anwendbar.

Ein Ziel ist, die Anforderungen der Zielbenutzer und die bestmögliche Auswahl an Verfahren und Technologien zu ermitteln. Dabei werden die Architekturen der Software mit Hilfe von UML-Diagrammen erstellt. Mit dieser Basis wird es durch die ausgewählten Technologien in ein ausführbares Programm umgesetzt.

Mit dem ersten Entwurf der Lösung und der Vorstellung bei den Zielbenutzern werden die Demo bewertet und mögliche Änderungen vorgeschlagen. Die Architektur der Software und das Entwicklungsverfahren werden auch in Kapitel 6 evaluiert.

Im Abschluss werden das Entwicklungsverfahren und Ergebnisse evaluiert. Mögliche Verbesserungen von technischer und fachlicher Seite werden dann in dem Ausblick diskutiert.

Inhaltsverzeichnis

ABSTRACT	I
KURZFASSUNG	II
INHALTSVERZEICHNIS	III
ABBILDUNGSVERZEICHNIS	V
TABELLENVERZEICHNIS	VI
1. EINLEITUNG	7
1.1. PROBLEMSTELLUNG	7
1.2. ZIELE UND ERGEBNISSE DER ARBEIT	8
1.3. AUFBAU DER BACHELORARBEIT	8
2. REQUIREMENTS ENGINEERING	9
2.1. ANFORDERUNGSERMITTLUNG	9
2.2. ANFORDERUNGSANALYSE	11
2.3. ANWENDUNGSFÄLLE	13
2.3.1. Anwendungsfalldiagramm	13
2.3.2. Beschreibung der Anwendungsfälle	14
2.4. SYSTEMANFORDERUNGEN	22
3. ARCHITEKTUR DESIGN	23
3.1. ENTITY-RELATIONSHIP-DIAGRAMM	23
3.2. OBJEKTORIENTIERTES DESIGN	26
3.3. SOFTWAREARCHITEKTUR	28
3.3.1. Konzepte der Softwarearchitektur	28
3.3.2. Übersicht von Frontend und Backend	30
3.3.3. Frontend	32
3.3.4. Rest API	35
3.3.5. Backend	38
3.4. ANWENDUNGSUMGEBUNGEN	41
3.4.1. Entwicklungsumgebung	41
3.4.2. Produktionsumgebung	44
4. TECHNOLOGIEN	46
4.1. ANGULAR	46
4.2. SPRING FRAMEWORK UND SPRING BOOT	46
4.3. POSTGRESQL	47

4.4. DOCKER	48
4.5. DIGITAL OCEAN	48
4.6. ZUSAMMENFASSUNG	49
5. IMPLEMENTIERUNG & TESTING	50
5.1. MOCK UP-GUI	50
5.2. EINGESETZTE TECHNOLOGIEN	54
5.3. TESTVERFAHREN UND REPORT	55
6. EVALUATION	57
7. FAZIT UND AUSBLICK	59
LITERATURVERZEICHNIS	61
APPENDIX	63
EIDESSTATTLICHE ERKLÄRUNG	69

Abbildungsverzeichnis

Abbildung 1: Anwendungsfalldiagramm	13
Abbildung 2: Aktivitätsdiagramm des Anwendungsfalls Authenticate Users	16
Abbildung 3: Aktivitätsdiagramm des Anwendungsfalls View Reports.....	17
Abbildung 4: Aktivitätsdiagramm des Anwendungsfalls Manage Prescriptions	21
Abbildung 5: Entity-Relationship-Diagramm	25
Abbildung 6: Klassendiagramm	27
Abbildung 7: Komponentendiagramm des gesamten Systems.....	30
Abbildung 8: Komponentendiagramm der Frontend-GUI-Komponente	32
Abbildung 9: Komponentendiagramm der Frontend-Controller-Komponente	33
Abbildung 10: Komponentendiagramm der Frontend-CacheService-Komponente	34
Abbildung 11: Komponentendiagramm der Frontend-HttpService-Komponente.....	35
Abbildung 12: Komponentendiagramm der Backend-ApiController-Komponente.....	39
Abbildung 13: Komponentendiagramm der Backend-DataService-Komponente	40
Abbildung 14: Komponentendiagramm der Backend-Repository-Komponente	41
Abbildung 15: Verteilungsdiagramm des Entwicklungssystems	43
Abbildung 16: Verteilungsdiagramm des Produktionssystems	45
Abbildung 17: Sign In Page.....	50
Abbildung 18: Products Page	51
Abbildung 19: Product Create-Edit Page	52
Abbildung 20: Order Detail Page	52
Abbildung 21: Bill Template.....	53
Abbildung 22: Order Detail Create-Edit Page	54
Abbildung 23: Ergebnis der Rest-API Integration Tests.....	56
Abbildung 24: Sign Up Page	67
Abbildung 25: Users Page	67
Abbildung 26: Order Create-Edit-Detail Page.....	68
Abbildung 27: Orders Page	68

Tabellenverzeichnis

Tabelle 1: Beschreibung des Anwendungsfalls Authenticate Users.....	16
Tabelle 2: Beschreibung des Anwendungsfalls View Reports	17
Tabelle 3: Beschreibung des Anwendungsfalls Manage Prescriptions	20
Tabelle 4: Beschreibung einiger Endpunkte der-RestAPI zwischen Backend und Frontend...	37
Tabelle 5: Beschreibung der-RestAPI zwischen Backend und Frontend	66

1. Einleitung

In diesem Kapitel werden die zu lösenden Probleme und Ziele festgestellt. Es besteht aus drei Abschnitten: Problemstellung, Ziele bzw. Ergebnisse und Aufbau der Bachelorarbeit.

1.1. Problemstellung

Als Hauptfunktionalitäten der Anwendungssoftware sollen die Verwaltung der Patientenprofile und das Medikamentenhandeln umgesetzt werden. Im Folgenden wird erklärt, wie die aktuelle Situation bei der Klinik ist und welche Probleme noch entstehen.

Nachdem die Patienten durch die Rezeption empfangen werden, werden sie durch die Ärzte untersucht und behandelt. Falls die Patienten zum ersten Mal bei der Klinik sind, wird ein neues Profil mit persönlichen Informationen der Patienten in einer Excel-Datei durch die Mitarbeiter an der Rezeption erstellt. Nach der Behandlung eines Patienten schreiben die Ärzte die Untersuchungsnotizen auf ein Blatt Papier, das später durch die Rezeption in diese Excel-Datei digital gespeichert wird. Entsteht der Bedarf für die Ärzte die alten Rekorde zu lesen, muss die Excel-Datei auf den Rechner der Ärzte kopiert oder ausgedruckt werden. Da das Geschäft eine Klinik für die Dermatologie ist, kommt es sehr häufig vor, dass Fotos auf verschiedenen Hautstellen der Patienten durch Analysemaschinen gemacht werden müssen. Aktuell werden die Bilder ohne Speicherung nur einmal ausgedruckt, analysiert. Sie werden von den Patienten gehalten und können verloren gehen. Die Funktionalität, diese Fotos speichern und später abfragen zu können, ist deswegen auch zu implementieren.

Neben der Behandlungsdienstleistung gegen Hautkrankheiten bietet die Klinik vor Ort noch ein Medikamentengeschäft, wo die Kunden Medikamente entweder nach ihrer Behandlung oder spontan nach Bedarf kaufen können. Das aktuelle Verwaltungssystem des Medikamentenhandels basiert teilweise auf Papierarbeit bzw. auf Excel und Access, wobei alle Daten nur lokal auf einem Rechner an der Rezeption gespeichert und abgefragt werden können. Mit Hilfe von dieser Software soll der Einkauf von Lieferanten und Verkauf an Patienten verwaltet werden. Einige spezifische Aktionen in der Software, die bisher durch alle Benutzer ausgeführt werden können, sollen später nach Bedarf nur durch bestimmte Rollen wie Ärzte oder Manager statt nur Verkäufer zugegriffen werden.

1.2. Ziele und Ergebnisse der Arbeit

Das erste Ziel dieser Arbeit ist, ein ausführbares Programm zu bauen, welches das aktuelle auf Excel basierende Verwaltungssystem ersetzt.

Ein weiteres Ergebnis dieser Arbeit ist neben der Anwendungsentwicklung den Entwicklungsprozess zu vermitteln. Hier geht es um alle Entwicklungsphasen einer Software wie z. B. Anforderungsanalyse, Architektur-Design, Datenbank-Design, Mock-Up GUI, Implementierung, Testing und Rollout. Zu diesem Zweck werden die verschiedenen Artefakte wie Anforderungsdokumente, UML Diagramme, die Softwarearchitektur und Quellcode erstellt.

Am Ende der Arbeit wird die Software bzw. ihr Entwicklungsprozess bewertet. Unter anderem wird über den aktuellen Stand und die zukünftige Weiterentwicklung dieser Software diskutiert.

1.3. Aufbau der Bachelorarbeit

Diese Arbeit wird auf sieben Kapitel aufgeteilt. Die Problemstellung und die Ziele werden im ersten Kapitel festgestellt. Das zweite Kapitel *Requirements Engineering* sammelt und analysiert die Anforderungen der Software. Unter anderem werden Anwendungsfalldiagramme und Aktivitätsdiagramme verwendet, um die Anforderungen zu verdeutlichen. Nachdem die Anforderungen festgestellt wurden, werden die ersten Architekturen der Software im dritten Kapitel gestaltet. Das Datenbankmodell wird in diesem Kapitel entworfen. Danach werden die benötigten Diagramme der statischen und dynamischen Modelle gestaltet, um die Architekturen und Hauptfunktionalitäten darzustellen.

Im vierten Kapitel werden Technologien wie *Angular*, *Spring Boot*, *Postgres*, *Docker* und *Digital Ocean* als Werkzeugkandidaten vorgestellt. Das fünfte Kapitel zeigt die Mock Up-GUIs und erläutert die Umsetzung und Testing der Software mit Hilfe der gewählten Technologien, welche im vorherigen Kapitel diskutiert wurden. Im sechsten Kapitel *Evaluation* werden der Prototyp und die Struktur der Software bewertet. Noch offene Probleme bzw. zu implementierende Funktionalitäten sind auch zu diskutieren. Die Arbeit schließt mit einem Fazit und einem Ausblick im siebten Kapitel.

2. Requirements Engineering

In diesem Kapitel werden die Anforderungen der Zielbenutzer gesammelt und analysiert. Es fängt mit dem Verfahren der Anforderungssammlung an. Hier wird vermittelt, wie die Anforderungen von Kunden gesammelt wurden. Im nächsten Schritt werden die Anforderungen in Textform beschrieben. Danach werden Anwendungsfälle mit Hilfe eines Anwendungsfalldiagramms und mehrerer Aktivitätsdiagramme dargestellt. Das Kapitel schließt mit dem Abschnitt über Systemanforderungen ab.

2.1. Anforderungsermittlung

Zur Ermittlung der Kundenanforderungen können diverse Werkzeuge und Verfahren der Softwareentwicklung zur Anwendung kommen. In diesem Abschnitt werden zunächst die Anforderungsquellen definiert, gefolgt von den Verfahren, um möglichst alle Anforderungen zu erfassen.

Anforderungen eines Projektes können von Stakeholdern, Systemen im Betrieb und Dokumenten kommen¹. Für dieses Projekt sind Stakeholder Ärzte und Ärztinnen, Pflegepersonal und die Managerin der Klinik. Sie sind Leute in der Klinik, welche dieses System verwenden werden und möglicherweise die Anforderungen beeinflussen. Es gibt eine Excel-basierte Software, die aktuell im Betrieb läuft. Teilweise werden Unterlagen in Papierform geschrieben und abgefragt. Da diese Software und die Unterlagen schon lange im Betrieb sind, können Arbeitsverhalten und Änderungswünsche der Stakeholder dadurch gesammelt werden. Dokumente bezüglich der alten Software wie Einleitung und Spezifikationsdokument sind auch hilfreich, weil sie die Hauptfunktionalitäten und Architektur der alten Software beschreiben. Diese können neue Anforderungen beeinflussen.

Um die Anforderungen von Quellen zu sammeln, können verschiedene Verfahren oder Techniken zur Auswahl kommen. Es soll die Frage beantwortet werden, was das Beste ist. Das ist abhängig von Projektumgebung, Erfahrung der Ingenieure und Stakeholdern. Die richtige Ant-

¹ (Pohl & Rupp, 2015) Seite 19

wort ist eine Technik oder eine Kombination von einigen Techniken, welche alle Anforderungen erfassen können². Für dieses Projekt werden die Techniken Beobachtung und Interview angewandt. Die Anzahl der Stakeholder ist nicht hoch und es ist einfach, individuelle Termine mit ihnen zu vereinbaren.

Die Technik Beobachtung ist geeignet, da es schon eine existierende Software gibt. Werden die Benutzer gebeten, ihre Jobs im Alltag mit der Software im Wort zu beschreiben, finden sie es schwierig. Details können verloren gehen oder sind inkorrekt³. Oft sind die Aufgaben so komplex, dass es unmöglich ist, sich alle Schritte zu merken. Andere Fälle sind, dass sich die Benutzer an den Aktivitäten gewöhnen. Sie können nicht mehr artikulieren, was sie machen. Es fällt ihnen leichter, die Aufgaben selber zu erledigen und zu zeigen statt im Wort zu beschreiben. Deswegen kann viel durch Beobachtung gelernt werden, wie die Benutzer ihre Aktivitäten durchführen⁴. Beobachtung kann still oder interaktiv sein. Stille Beobachtungen werden verwendet, wenn beschäftigte Benutzer nicht gestört werden möchten. Auf der anderen Seite können bei interaktiven Beobachtungen Fragen dazwischen gestellt werden. Es wird deutlicher verstanden, warum die Benutzer eine bestimmte Entscheidung an einem Schritt getroffen haben. Wünsche zu möglichen Änderungen an die neue Software können direkt vor Ort geäußert werden.

Offensichtlich ist die einfachste Lösung, um zu wissen, was die Benutzer von einer Software wünschen, sie direkt zu fragen. Es ist eine traditionelle Methode der Anforderungsermittlung für kommerzielle Produkte und Informationssysteme. Interviews werden meist individuell oder unter einer kleinen Gruppe gehalten, woran die Benutzer aktiv teilnehmen können, um Ihre Meinungen bezüglich Funktionalitäten der neuen Software zu äußern. Es ist einfacher, solche Veranstaltungen als große Workshops zu organisieren und zu halten⁵. Mögliche Fragen eines Anforderungsinterviews können wie folgt aussehen⁶:

- Was ist das Hauptmerkmal der Software und warum ist dieses Merkmal wichtig?
- Welche Merkmale sind abhängig und unabhängig von diesem Merkmal?
- Bewertungsskala für die Wichtigkeit der Merkmale verwenden

² (Koelsch, 2016) Seite 213

³ (Wiegers & Beatty, 2013) Seite 125

⁴ (Wiegers & Beatty, 2013) Seite 126

⁵ (Wiegers & Beatty, 2013) Seite 121

⁶ (Laplante, 2018) Seite 65

2.2. Anforderungsanalyse

Nachdem die Anforderungen durch verschiedene Techniken und von diversen Quellen gesammelt wurden, werden sie nun beschrieben und analysiert. Die Hauptbenutzer der Software teilen sich in drei Rollen auf: *Manager*, *Doctor* (Arzt) und *Nurse* (Pflegepersonal).

Als *Nurse* sollen die Termine und Patientenprofile abgefragt, angelegt, modifiziert und gelöscht werden können. Ein Termin hat die exakte Uhrzeit des Termins, den Terminplatz am Tag und den Status, ob der Patient anwesend zum Termin ist. Je zwanzig Minuten wird ein Terminplatz reserviert. Das heißt, für einen Arbeitstag stehen 24 Terminplätze zur Verfügung. Für ein Patientenprofil werden persönliche Daten wie Name, Geburtsdatum, Versicherungsnummer gespeichert.

Unter anderem möchte *Nurse* für den Medikamentenhandel auch die Produkte, Bestellungen, Bestelldetails und Rechnungen dafür abfragen und anlegen können. Ein Produkt ist in diesem Projekt ein Medikament und hat einen Identifizierungscode, Produktnamen, den Preis aus Lieferanten und den Preis an Kunden. In einer Bestellung wird gesehen, welche Bestelldetails darin mit Produktinformationen und Anzahl sind, ob sie von Lieferanten kommt oder an Kunden geht und wie der aktuelle Stand der Bestellung ist.

Als *Doctor* sollen die Termine, Patientenprofile, Besuche und Rezepte der Patienten abgefragt, angelegt, modifiziert und gelöscht werden können. Zusätzlich kann *Doctor* auch andere Rechte von *Nurse* haben. Nach einem Termin sollen Untersuchungsdaten eines Besuches gespeichert werden. Diese Daten sind z. B. Besuchsgrund, Untersuchungsbeschreibung und Kommentare des Arztes. Benötigt der Arzt Medikamente für die Behandlung, werden Rezepte mit Notizen durch den Arzt geschrieben und im System gespeichert.

Als *Manager* sollen die Produkte, Rechnungen, Bestellungen und Reporte abgefragt, angelegt, modifiziert und gelöscht werden können. Für die Rechnung einer Bestellung muss eine PDF-Datei zur Verfügung gestellt und ausgedruckt werden können. Die täglichen und monatlichen Reporte können nach Bedarf in Form von PDF-Dateien für *Manager* gestellt werden. Außerdem hat *Manager* auch alle Rechte von *Doctor* und *Nurse*. Zum Medikamentenhandel möchte *Manager* die Produkte, Bestellungen, Bestelldetails, Rechnungen und Reporte anlegen, modifizieren und löschen können. Zur Benutzerverwaltung möchte *Manager* eine Sicht haben, um zu lesen, welcher Benutzer was zu welchem Zeitpunkt anlegt, modifiziert oder löscht.

Die Rollen begrenzen die Sicht auf die Daten. *Nurse* kann um einen Vorgang zu kennzeichnen z. B. eine neue Bestellung anlegen, hat aber keine Möglichkeit die zugehörigen Daten später zu verändern.

Da der Großteil der Arbeit mit dem System aus Suchen und Filtern der Daten besteht, ist eine Such- und Filterfunktion für jede Entität zu implementieren. Zum Schluss jedes Tages bzw. Monats sollten Reports automatisch erstellt werden.

Da eine Anforderung der Chefin der Klinik feststellt, dass der Medikamentenhandel zunächst gebraucht wird, soll dieses Teilsystem als erstes implementiert werden. Die anderen Komponenten können im Laufe der Zeit hinzugefügt werden.

Zu guter Letzt wird das Thema Datenschutz auch betrachtet, weil die Software mit sensiblen Patienteninformationen arbeitet. Es wird nach dem Patientenrechtegesetz in Vietnam verwaltet. Gemäß Kapitel 1 § 3 Abs. 2 und Kapitel 2 § 8 Abs. 2 des Patientenrechtegesetzes haben Patienten Recht, personen- und untersuchbezogene Informationen zu schützen. Diese Daten dürfen nur durch Zustimmung der Patienten für die interne Bildung der Pflegepersonale oder für bestimmte Zwecke verwendet werden, wenn andere relevanten Gesetze erlauben. Die Software darf nur durch authentifizierte Konten zugegriffen werden. Pflegepersonale oder Mitarbeiter mit Zugriff auf Patientendaten dürfen nach Kapitel 1 § 6 Abs. 10 und Kapitel 3 § 37 Abs. 5 die Daten nicht verbreiten oder verfälschen. Lautet Kapitel 5 § 59 Abs. 3 müssen Patientenprofile mindestens zehn Jahre archiviert werden. Falls die Daten in der Cloud gespeichert werden, muss ein Backup regelmäßig durchgeführt werden⁷.

In diesem Projekt wird *Digital Ocean* verwendet, um Daten in der Cloud zu speichern. *Digital Ocean* schützt Daten von Nutzer nach GDPR (*General Data Protection Regulation*)⁸, welche die vietnamesische Patientenrechtegesetze auch erfüllen. Zur Datenbackup bietet *Digital Ocean* auch die Möglichkeit, alle Daten des Servers wöchentlich zu archivieren⁹.

⁷ <https://thuvienphapluat.vn/van-ban/The-thao-Y-te/Luat-kham-benh-chua-benh-nam-2009-98714.aspx>
[Stand 2021.08.25]

⁸ <https://www.digitalocean.com/legal/gdpr-faq/> [2021.08.25]

⁹ <https://www.digitalocean.com/pricing> [2021.08.25]

2.3. Anwendungsfälle

In diesem Abschnitt werden nach den oben beschriebenen Anforderungen die Hauptanwendungsfälle durch ein Anwendungsfalldiagramm und Beschreibungen in Form von Tabellen bzw. Aktivitätsdiagrammen analysiert.

2.3.1. Anwendungsfalldiagramm

Während der gesamten Arbeit werden UML Diagramme als standardisiertes Werkzeug zur Erstellung grafischer Softwaremodelle verwendet. In diesem Teil kommt das UML-Anwendungsfalldiagramm zum Einsatz. Ein Anwendungsfalldiagramm besteht aus Akteuren und Anwendungsfällen. Das zeigt im ersten Blick, wer das System nutzt und was das System machen kann. Das Anwendungsfalldiagramm dieser Software wird auf der Abbildung 1 dargestellt.

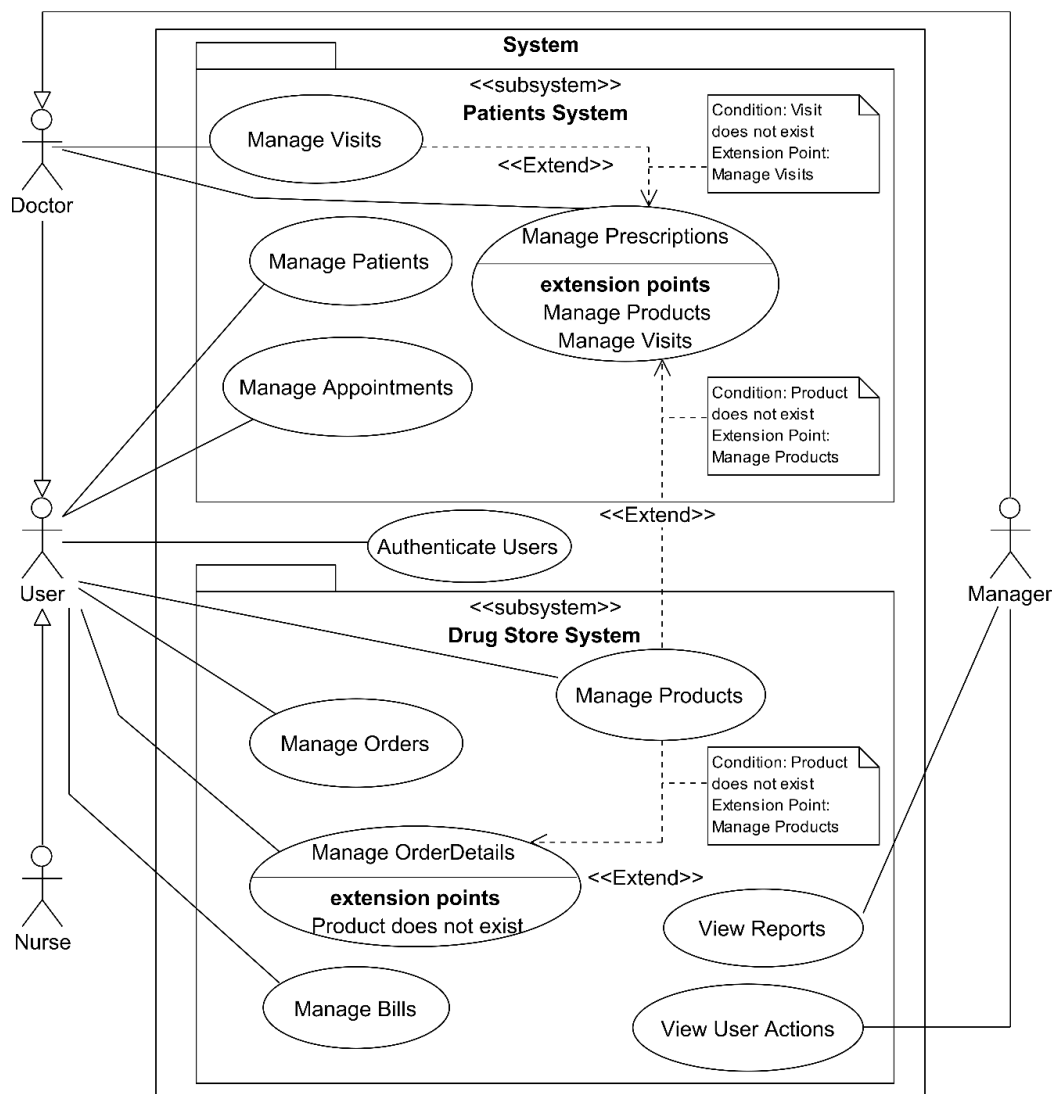


Abbildung 1: Anwendungsfalldiagramm

Das System besteht aus zwei Subsystemen: *Patients System* und *Drug Store System*. Das *Patients System* kapselt die Anwendungsfälle der Patientenverwaltung und das *Drug Store System* ist verantwortlich für die Anwendungsfälle des Medikamentengeschäftes.

Das *Patients System* stellt die folgenden Anwendungsfälle zur Verfügung: *Manage Patients* (Patientenprofileverwaltung), *Manage Appointments* (Terminverwaltung), *Manage Visits* (Besuchsverwaltung) und *Manage Prescriptions* (Rezeptverwaltung).

Auf der anderen Seite bietet das *Drug Store System* die Anwendungsfälle: *Manage Orders* (Bestellungsverwaltung), *Manage OrderDetails* (Bestellungsdetailverwaltung), *Manage Bills* (Rechnungsverwaltung), *Manage Products* (Produktverwaltung), *View Reports* (Reportanzeige) und *View User Actions* (Benutzeraktionenanzeige).

Dieses System wird von drei Hauptrollen benutzt: *Nurse*, *Doctor* und *Manager*. Die Rolle *User* ist eine Basisrolle, welche Zugriff auf die Anwendungsfälle *Manage Patients*, *Manage Appointments*, *Manage Products*, *Manage Orders*, *Manage OrderDetails* und *Manage Bills* hat. Erbt die Rolle *Nurse* von *User* wie auf Abbildung 1 angezeigt, übernimmt *Nurse* automatisch alle Rechte von *User*. Neben den Rechten aus der Vererbung der Rolle *User* wie *Nurse* kann die Rolle *Doctor* zusätzlich Besuche und Rezepte durch die Anwendungsfälle *Manage Visits* und *Manage Prescriptions* verwalten. Zum Schluss gibt es die Rolle *Manager*, welche alle Rechte von *Doctor* übernimmt und Reporte bzw. andere Benutzeraktionen durch die Anwendungsfälle *View Reports* und *View User Actions* lesen darf.

Mit dem Anwendungsfall *Authenticate Users* kann jede Rolle bzw. jeder Benutzer am System mit einem Konto authentifiziert werden. Alle anderen Anwendungsfälle sind von diesem Anwendungsfall abhängig, da das System gegen nicht authentifizierte Zugriffe geschützt ist. Um die Verständlichkeit des ganzen Anwendungsfalldiagramms zu halten, wird diese Abhängigkeit auf dem Diagramm nicht angezeigt.

2.3.2. Beschreibung der Anwendungsfälle

In diesem Abschnitt werden nur drei Anwendungsfälle schriftlich in Form von Tabellen und grafisch in Form von Aktivitätsdiagrammen stellvertretend für das Gesamtprojekt beschrieben. Die Ausführung aller Anwendungsfälle würde den Rahmen dieser Arbeit sprengen, ohne einen signifikanten Mehrwert zu generieren. Daher sind die Beschreibungen weiterer Anwen-

dungsfälle lediglich im Appendix angehängt und können bei Bedarf nachgelesen werden. Aktivitätsdiagramme sind geeignet, um den Arbeitsablauf einer Dienstleistung bzw. eines Anwendungsfalls zu modellieren. Nicht nur die Hauptreihenfolge eines Anwendungsfalls sondern auch die alternativen Abläufe werden dadurch präzise dargestellt.

Diese drei Anwendungsfälle sind *Authenticate Users*, *View Reports* und *Manage Prescriptions*. Ein Anwendungsfall führt die Hauptfolge der Aktionen im Abschnitt *Main Sequence* in der Beschreibungstabelle aus, falls nichts Anderes gegeben ist. Im Abschnitt *Alternative Sequence* werden die alternativen Abläufe dargestellt, falls die gegebenen Bedingungen zutreffen. Jeder Ablauf fängt mit einer Nummer gefolgt von einem Buchstaben z. B. „3a“ an. Die Nummer bezieht sich auf den Schritt direkt davor. Der Buchstabe hilft dabei, verschiedene Abläufe eines Schnittes zu unterscheiden. Die alternativen Abläufe z. B. „3a“ und „3b“ sind zwei alternative Abläufe für Schritt „3“ direkt davor, falls besondere Bedingungen zutreffen.

Use case name	<i>Authenticate Users</i>
Summary	Bevor das System verwendet werden darf, müssen sich die Benutzer zuerst authentifizieren
Actor	<i>Nurse, Doctor, Manager</i>
Precondition	Das System wird auf einem Rechner oder Smartphone gestartet. Nur die Anmeldungsseite wird angezeigt
Main sequence	<ol style="list-style-type: none">1. Benutzer gibt seine E-Mail-Adresse und sein Passwort in die geeigneten Eingabefelder ein2. Benutzer bestätigt mit dem Button <i>Sign In</i>3. System zeigt eine Notifikation mit einer Nachricht an, dass es ein paar Sekunden dauert und der Benutzer warten muss4. System zeigt eine Notifikation an, dass der Benutzer erfolgreich angemeldet hat5. System leitet den Benutzer auf andere Seiten weiter
Alternative sequence	<ol style="list-style-type: none">1a. E-Mail wurde in einem falschen Format eingegeben1. System zeigt den Button <i>Sign In</i> nicht

	<ol style="list-style-type: none"> 2. Benutzer versucht nochmal E-Mail in dem korrekten Format einzugeben <p>2a. E-Mail oder das Passwort ist falsch</p> <ol style="list-style-type: none"> 1. System leitet den Benutzer nicht weiter 2. System zeigt eine Fehlermeldung mit dem Grund auf dem Bildschirm an 3. Benutzer versucht nochmals E-Mail und Passwort einzugeben
Postcondition	Benutzer ist erfolgreich angemeldet und hat Zugriff auf andere Seiten

Tabelle 1: Beschreibung des Anwendungsfalls Authenticate Users

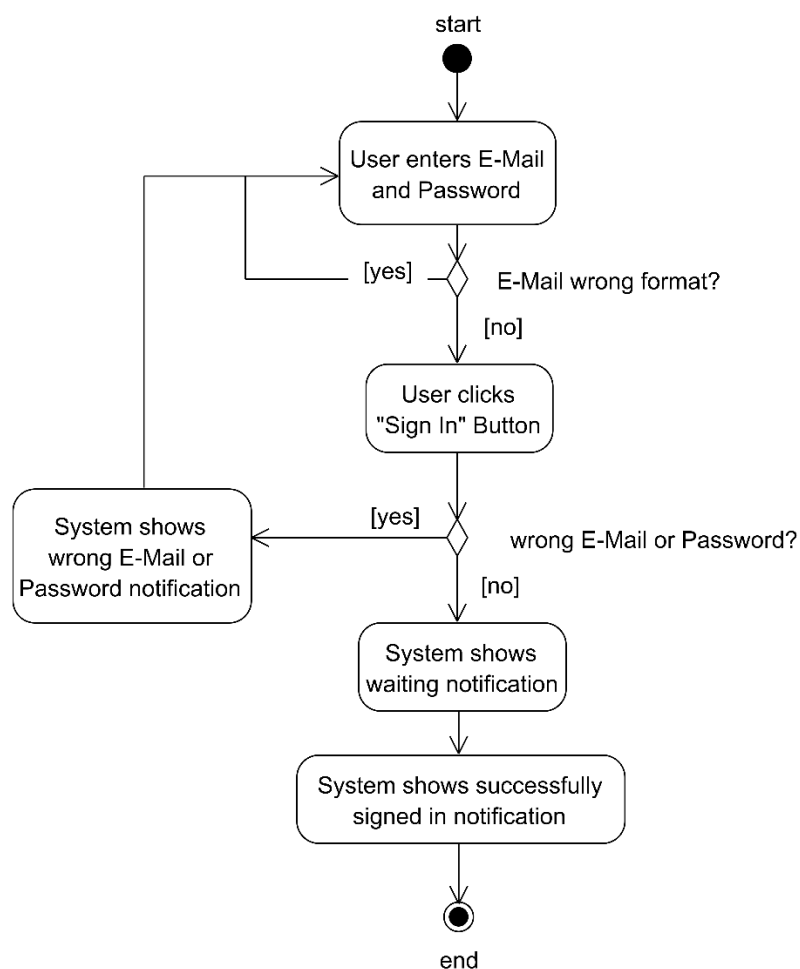


Abbildung 2: Aktivitätsdiagramm des Anwendungsfalls Authenticate Users

Use case name	View Reports
Summary	Abrechnungen von vergangenen Tagen und Monaten, die automatisch abgerechnet wurden, werden für <i>Manager</i> angezeigt
Actor	<i>Manager</i>
Precondition	Bestellungen der Tage bzw. Monate existieren und sind abgerechnet
Dependency	Inkludiert Anwendungsfall <i>Authenticate Users</i>
Main sequence	<ol style="list-style-type: none"> 1. Benutzer ruft die Seite der Reports auf 2. System zeigt auf dem Bildschirm alle Reports, die am gewählten Tag oder im gewählten Monat schon abgerechnet wurden 3. Benutzer wählt anderen Tag oder Monat, um weitere Reports zu sehen 4. System zeigt die Reportseite des gewählten Tages oder Monats dem Benutzer an
Postcondition	Reports der Bestellungen der Tage bzw. Monate wurden abgefragt.

Tabelle 2: Beschreibung des Anwendungsfalls View Reports

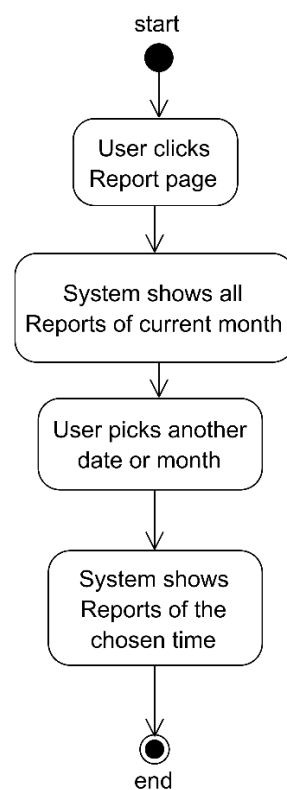


Abbildung 3: Aktivitätsdiagramm des Anwendungsfalls View Reports

Use case name	<i>Manage Prescriptions</i>
Summary	Rezepte können angelegt, abgefragt, modifiziert und gelöscht werden.
Actor	<i>Manager, Doctor</i>
Precondition	Benötigte Produkte und der dazugehörige Besuch des Patienten wurden schon angelegt
Extensions	Extendiert Anwendungsfälle <i>Manage Visits</i> und <i>Manage Products</i>
Dependency	Inkludiert Anwendungsfall <i>Authenticate User</i>
Main sequence	<ol style="list-style-type: none"> 1. Benutzer ruft die Seite der Rezepte eines Besuches auf. 2. System zeigt alle Rezepte auf dem Bildschirm an 3. Benutzer wählt die Aktion Anlegen, Abfrage, Modifikation oder Löschen aus 4. System zeigt eine Notifikation an, dass der Benutzer die Aktion erfolgreich durchgeführt hat
Alternative sequence	<p>3a. Benutzer legt ein neues Rezept an</p> <ol style="list-style-type: none"> 1. Benutzer klickt auf <i>New Prescription</i> Button 2. System leitet Benutzer auf die <i>New Prescription</i> Seite weiter 3. Benutzer gibt die benötigten Informationen eines neuen Rezeptes ein 4. Benutzer klickt auf den <i>Submit</i> Button <p>2a. Benutzer hat die Rolle <i>Nurse</i></p> <ol style="list-style-type: none"> 1. System zeigt eine Fehlermeldung an, dass der Benutzer keinen Zugriff hat, um die Aktion durchzuführen 2. Benutzer wird wieder auf die Seite der Rezepte weitergeleitet <p>3a. Patientenbesuch existiert noch nicht. Bedingung <<<i>Visit does not exist</i>>> trifft zu</p> <ol style="list-style-type: none"> 1. Extendiert den Anwendungsfall <i>Manage Visits</i>

	<ol style="list-style-type: none"> 2. System leitet Benutzer auf die Seite zur Anlegung eines neuen Besuchs weiter 3. System leitet Benutzer zurück <p>3b. Medikamenteninformationen existieren noch nicht. Bedingung <<Product does not exist>> trifft zu.</p> <ol style="list-style-type: none"> 1. Extendiert den Anwendungsfall <i>Manage Products</i> 2. System leitet Benutzer auf die Seite zur Anlegung eines neuen Produktes weiter 3. System leitet Benutzer zurück <p>3b Benutzer ruft ein angelegtes Rezept auf</p> <ol style="list-style-type: none"> 1. Benutzer klickt auf <i>Prescription Detail</i> Button 2. System zeigt die Informationen des Rezeptes vollständig an <p>3c Benutzer modifiziert ein Rezept</p> <ol style="list-style-type: none"> 1. Benutzer klickt auf <i>Edit Prescription</i> Button 2. System zeigt die Inputfelder mit eingefügter Information des Rezeptes im System an 3. Benutzer modifiziert Informationen des Rezeptes nach Wunsch 4. Benutzer klickt auf <i>Submit</i> Button <p>3d Benutzer löscht ein Rezept</p> <ol style="list-style-type: none"> 1. Benutzer klickt auf <i>Delete Prescription</i> Button 2. System zeigt eine Notifikation, um zu fragen, ob der Benutzer das Rezept wirklich löschen will 3. Benutzer klickt auf <i>Yes</i> <p>1a. Benutzer hat die Rolle <i>Nurse</i></p> <ol style="list-style-type: none"> 1. System zeigt eine Fehlermeldung an, dass der Benutzer keinen Zugriff hat, um die Aktion durchzuführen 2. Benutzer wird wieder auf die Seite der Rezepte weitergeleitet
--	---

	<p>3a. Benutzer klickt auf <i>No</i></p> <p>1. Benutzer bleibt auf der Seite der Rezepte stehen</p>
Postcondition	Rezepte wurden erfolgreich angelegt, abgefragt, modifiziert oder gelöscht

Tabelle 3: Beschreibung des Anwendungsfalls Manage Prescriptions

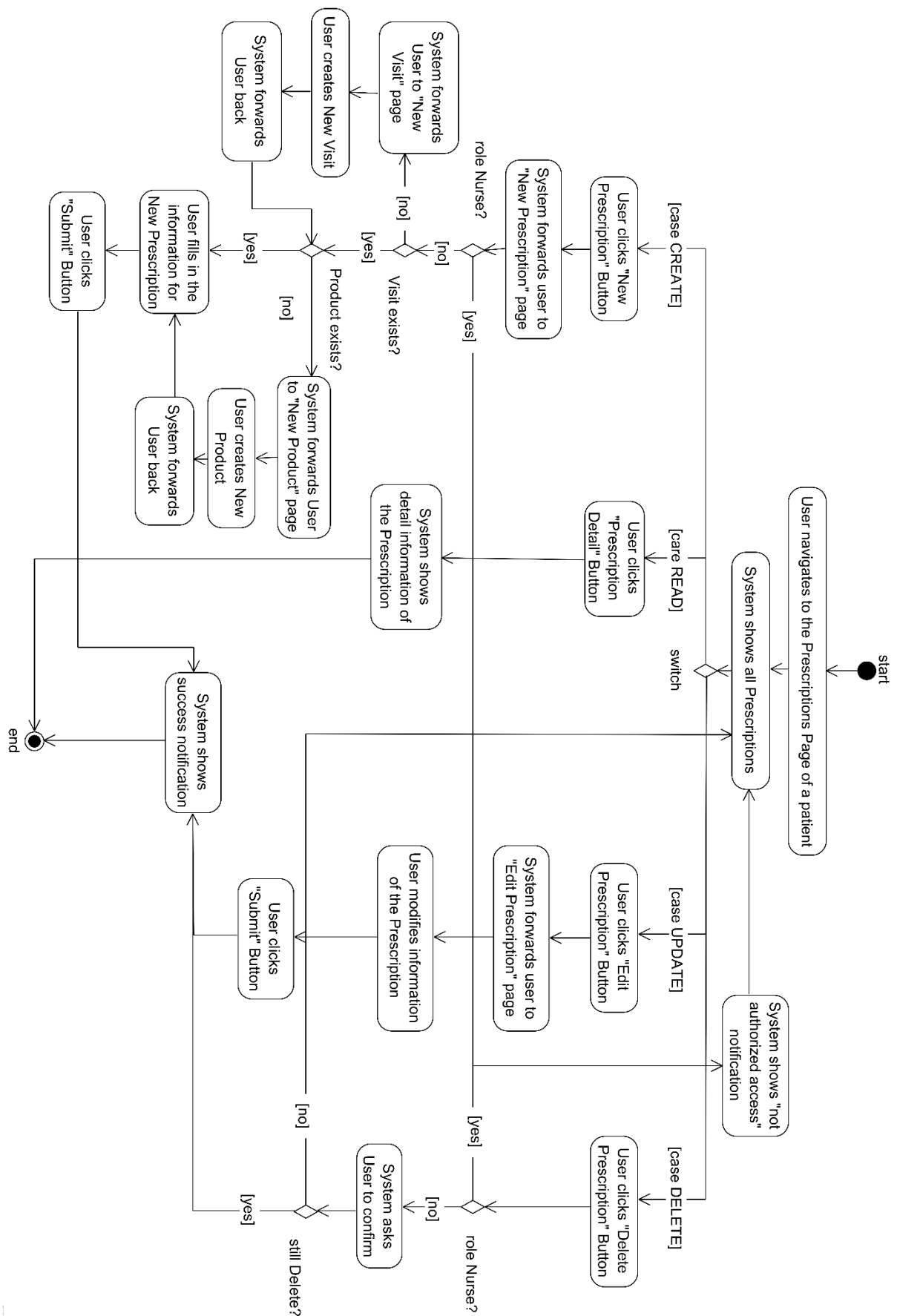


Abbildung 4: Aktivitätsdiagramm des Anwendungsfalls Manage Prescriptions

2.4. Systemanforderungen

In diesem Abschnitt werden Systemanforderungen, die das System aufweisen muss, festgestellt. Faktoren wie Performanz, Zeitverzögerung und Plattform gehören zu dieser Kategorie.

Folgende Systemanforderungen sind im Rahmen dieser Arbeit umzusetzen:

1. Das System sollte eine webresponsive Version für mobile Geräte (*Android* und *iOS*) und eine für Desktop-Browser (1920x1080) zur Verfügung stellen.
2. Auf das System muss durch das Internet auf einem Web-Browser zugegriffen werden.
3. Die Zeitverzögerung darf maximal fünf Sekunden für jede Aktion sein.
4. Das System darf nur im Zeitraum von 0 bis 6 Uhr GMT +7 zum Maintenance offline gestellt werden.
5. Auf das System muss bis zu 99% der Zeit eines Arbeitstages von 8Uhr bis 18Uhr zugegriffen werden können.
6. Das System muss so skalieren, dass fünf bis zehn Benutzer gleichzeitig am System arbeiten können.

3. Architektur Design

In diesem Kapitel werden die Komponenten der Software mit Hilfe von verschiedenen UML-Diagrammen strukturiert. Es fängt mit dem *Entity-Relationship-Diagramm* als die Struktur der Datenbank an, gefolgt von dem Klassendiagramm als Darstellung der Datenbank in einer Programmiersprache. Darüber hinaus werden die Strukturen der Software in Form von Komponentendiagrammen dargestellt. In diesem Abschnitt wird eine *RestAPI* als die Kommunikationsschnittstelle des *Frontends* und *Backends* vorgestellt. Zum Schluss werden die Entwicklungsumgebung und Produktionsumgebung der Software mit Hilfe von Verteilungsdiagrammen beschrieben.

3.1. Entity-Relationship-Diagramm

Als Erstes wird das Datenbankmodell der Software durch ein *Entity-Relationship-Diagramm* dargestellt. In Abbildung 5 lässt sich sehen, dass vierzehn Tabellen zu speichern sind. Diese sind: *User*, *UserLog*, *Role*, *Appointment*, *Visit*, *Patient*, *Prescription*, *Product*, *Inventory*, *InventoryStats*, *Order*, *OrderDetail*, *Bill* und *Report* aufgeteilt in zwei Subsystemen. Das *Patients System* ist verantwortlich für die Verwaltung der Patienten, Untersuchungen, Termine und Rezepte. Auf der anderen Seite verwaltet das *Drug Store System* den Handel der Medikamente und monatliche Reporte.

Im Patientensystem enthält die Tabelle *User* Benutzerdaten, die im System registriert wurden. Ein Benutzer hat neben den normalen Attributen wie Name und Passwort auch eine *id* als eine eindeutige Referenz auf sich selbst. Außerdem hat jeder Benutzer eine Referenz auf die Tabelle *Role*, um zu identifizieren, welche Rolle der Benutzer besitzt.

Zu einer Administratoranforderung möchte die Rolle *Manager* wissen, zu welchem Zeitpunkt wer was anlegt, modifiziert und löscht. Als Lösungsansatz wird hier eine Tabelle *UserLog* gebraucht, um diese Informationen zu speichern und für *Manager* zur Verfügung zu stellen. Ein *UserLog* hat neben der *id* die Art der Aktion *crudType*, *entityBefore* für Originalwert, *entityAfter* für geänderter Wert der Entität und die angelegte Zeit, wann die Aktion durchgeführt wurde.

Braucht ein Patient einen Termin, werden die Termininformationen in die Tabelle *Appointment* gespeichert. Es werden die Attribute *appointmentTime* und *appointmentStatus* benötigt, zu wissen, zu welchem Zeitpunkt der Termin stattfindet und wie sein aktueller Status ist. An einem Arbeitstag kann ein Arzt maximal 24 Termine leisten. Deswegen ist es auch wichtig, einen Termin zu dem geeigneten Platz des Arbeitstages mit dem Attribut *appointmentSlot* zuzuordnen. Die Referenzen *doctorId* und *patientId* zeigen, zu welchem Arzt und Patienten der Termin zugeordnet ist.

Wird der Patient am Termin mit der *appointmentId* untersucht, schreibt der Arzt die Untersuchungsinformationen in die Tabelle *Visit*. Einige wichtige Attribute sind *reasons* (Gründe der Untersuchung), *diagnosis* (Diagnose) und *notes* (andere Notizen des Arztes).

Zum Schluss einer Untersuchung können durch den Arzt je nach Bedarf Rezept-Informationen in die Tabelle *Prescription* geschrieben werden. Mit den Referenzen *visitId* und *productId* wird angezeigt, zu welcher Untersuchung das Rezept gehört und was für Medikamente (Produkt) das Rezept enthält. Anzahl der Medikamente und andere Notizen eines Rezeptes werden auch durch die Attribute *quantity* und *notes* mitgespeichert.

Im Medikamentensystem wird die Tabelle *Product* gebraucht, um die Informationen aller Medikamente zentral zu speichern. Neben der *id* hat jedes Produkt einen Namen, Code, Einkaufspreis, Verkaufspreis und die angelegte Zeit. Dazu gehört auch die Tabelle *Inventory*, um zu zeigen, wie vorrätig das Medikament im Lager ist.

Für Bestellungen wird die Tabelle *Order* gebraucht. Eine Bestellung hat *id*, *orderType* (Einkauf vs. Verkauf), *orderStatus*, den Preis der gesamten Bestellung und den Anlagezeitpunkt.

Für eine Bestellung wird eine Rechnung in der Tabelle *Bill* erstellt. Ähnlich zu einer Bestellung werden hier die ähnlichen Attribute wie Rechnungstyp (Einkauf vs. Verkauf), Rechnungsstatus, der gesamte Preis und der Anlagezeitpunkt benötigt.

Um zu beschreiben, was eine Bestellung enthält, kommt die Tabelle *OrderDetail* als Bestellungswünsche zum Einsatz. Jeder Bestellungswunsch hat zwei Fremdschlüssel als Referenzen auf die Tabellen *Product* und *Order*, um die Beziehungen zwischen ihnen zu modellieren. D. h. für jeden Bestellungswunsch wird dann verdeutlicht, zu welcher Bestellung der Bestellungswunsch gehört bzw. welche Produkte mit der dazugehörigen Anzahl der Bestellungswunsch enthält.

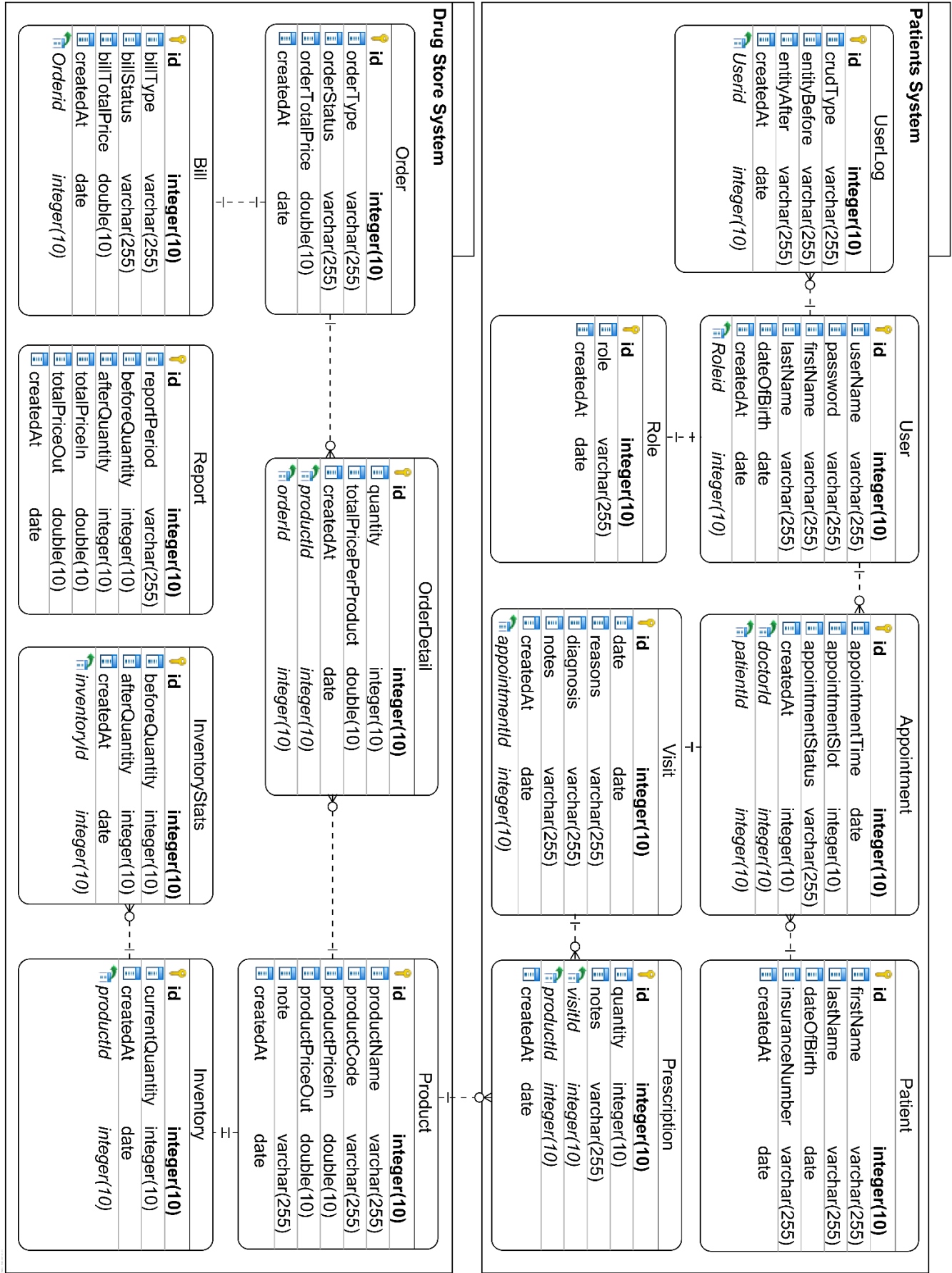


Abbildung 5: Entity-Relationship-Diagramm

3.2. Objektorientiertes Design

In diesem Abschnitt wird im Rahmen der objektorientierten Analyse ein Datenklassendiagramm erstellt und analysiert. Dieses Diagramm modelliert die Datenbausteine und ihre Beziehungen, die sich zwischen verschiedenen Komponenten der Software bewegen und als eine Schnittstelle zur Datenbank fungieren. Diese Klassen sind wie in Abbildung 6 unten dargestellt.

Da die Datenklassen und ihre Beziehungen eine Repräsentation der relationalen Tabellen in der objektorientierten Programmierwelt sind, ist das Klassenmodell ein Äquivalent zu dem *Entity-Relationship-Diagramm* oben. Der Fokus hier ist die Aggregation und Komposition-Beziehung zwischen Klassen.

Aggregation und Komposition sind spezifische Fälle von Besitzbeziehungen zwischen Klassen. Genauer gesagt ist eine Klasse ein Teil anderer Klasse. Auf einer Seite impliziert Aggregation eine Beziehung, wobei das Objekt der Teilklassse weiter existiert, wenn das Objekt der Ganze-Klasse gelöscht wird. Auf der anderen Seite beschreibt Komposition eine engere Beziehung, wobei das Objekt der Teilklassse zusammen mit der Ganze-Klasse gelöscht wird.

Auf diesem Klassendiagramm bezüglich der Komposition gibt es vier Kompositionsbeziehungen. Die sind: *Appointment* (ist Teil von *Patient*), *OrderDetail* (ist Teil von *Order*), *InventoryStats* (ist Teil von *Inventory*) und *Inventory* (ist Teil von *Product*). Ein Bestellwunsch gehört immer zu einer Bestellung und darf nicht allein existieren. Ähnlich dazu muss ein Termin immer zu einem Patienten zugeordnet werden.

Denn die Klasse *InventoryStats* beschreibt, wie sich die Anzahl eines Medikamentes im Lager ändert, darf sie nicht ohne die Klasse *Inventory* existieren. Von der gleichen Bedeutung besitzt die Klasse *Product* und *Inventory* auch diese Beziehung, weil die Klasse *Inventory* den aktuellen Stand der Anzahl eines Medikaments meldet.

Die Beziehung der anderen Klassen sind Aggregationen oder reine Assoziationen. Das heißt, nach der Trennung ihrer Beziehungen können sie selbständig weiter existieren.

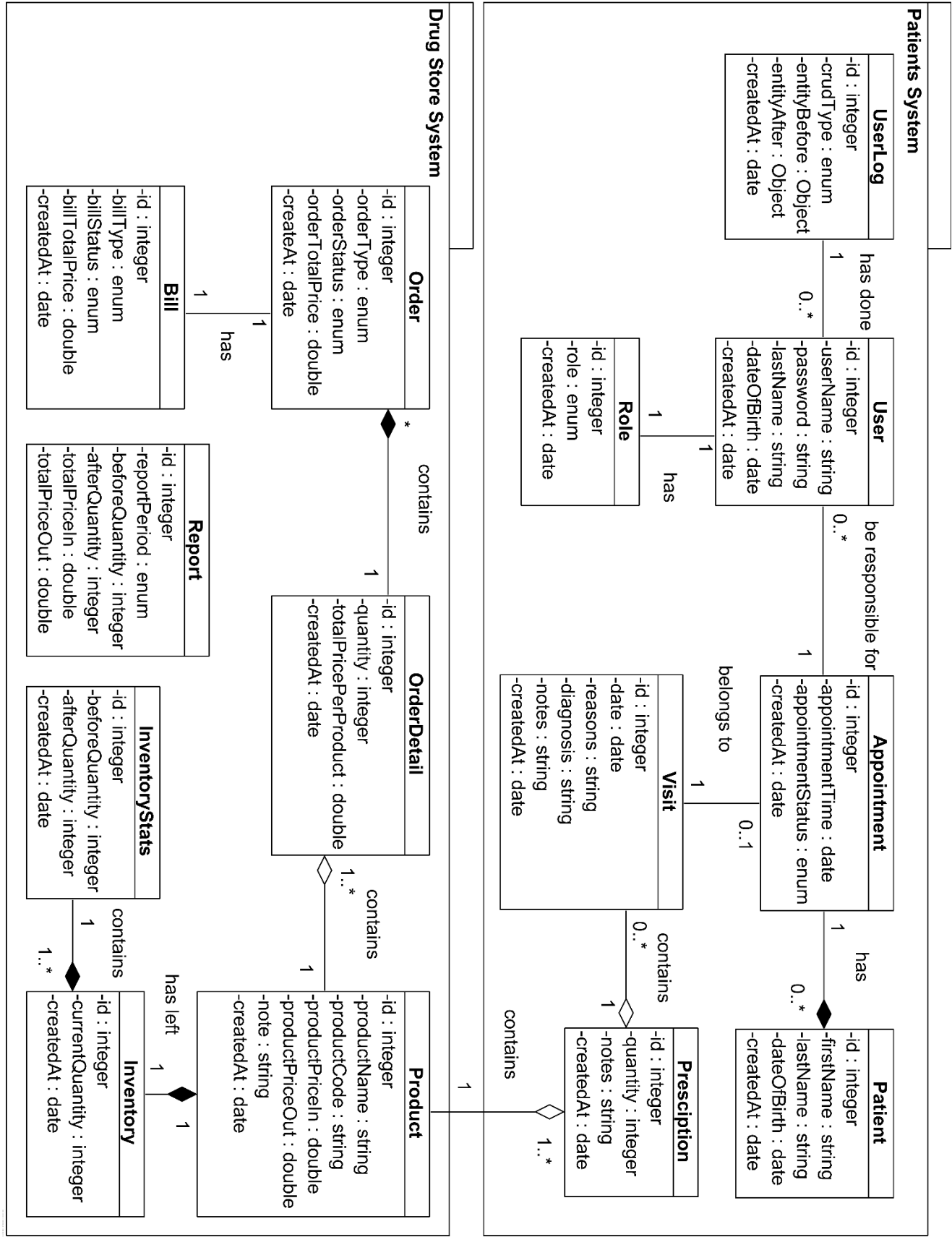


Abbildung 6: Klassendiagramm

3.3. Softwarearchitektur

Dieser Abschnitt beantwortet einige Fragen bezüglich der Softwarearchitektur und stellt die Architektur dieser Software in Form von Komponentendiagrammen dar. Zunächst fängt es mit den Basiskonzepten der Softwarearchitektur an. Hier wird es diskutiert, was Softwarearchitektur ist, warum diese wichtig ist, was die populären Architekturmodelle sind und welches Architekturmodell für dieses Projekt ausgewählt wird. Danach wird die Architektur dieser Software im Detail beschrieben. Die Software besteht aus einem Frontend, einem Backend und einer Datenbank. Darüber hinaus wird eine RestAPI als Kommunikationsschnittstelle zwischen dem *Frontend* und dem *Backend* vorgestellt.

3.3.1. Konzepte der Softwarearchitektur

Jedes Softwaresystem besteht aus mehreren Teilen, welche zusammen verbunden sind, miteinander und mit den externen Systemkomponenten kommunizieren. Die Anzahl der Teile ist unterschiedlich, aber die Kommunikationsart ist deterministisch. Das heißt, ob das Verhalten einer Komponente simpel oder kompliziert ist, muss immer vorhersehbar und beschreibbar sein. Das ist die Architektur einer Software. Mit anderen Worten ist es ähnlich wie die Architektur eines Gebäudes oder eines Schiffes¹⁰. Die Softwarearchitektur wird als ein gemeinsames Werkzeug aufgebaut, um die Bedürfnisse, Bedenken, Ziele und Zielsetzungen der Stakeholder zu beschreiben. Das fasst eine Anzahl der Architekturelemente und ihre Beziehungen um. Eine Softwarearchitektur soll dokumentiert werden, um zu demonstrieren, dass das die Bedürfnisse der Stakeholder erfüllt¹¹.

In Folgendes sind einige populären Architekturmodelle, die zur Wahl stehen.

- *Layered Architecture*: die Softwarekomponenten werden in Horizontalebene organisiert. Jede spielt eine spezifische Rolle in der Applikation (GUI, Datenbankabfragen oder Businesslogik). Eine Ebene muss nicht wissen, wie andere Ebenen Daten behandeln. Sie müssen nur wissen, wie sie durch ihre Schnittstellen kommunizieren¹².

¹⁰ (Rozanski & Woods, 2005) Seite 17

¹¹ (Rozanski & Woods, 2005) Seite 25

¹² (Richards, 2015) Seite 2

- *Event-Driven Architecture*: das Model besteht aus stark entkoppelten Einzweck-Ereignisverarbeitungs-komponenten, welche Ereignisse asynchron empfangen und verarbeiten. Das Model wird in zwei Haupttopologien aufgeteilt: *Mediator* und *Broker*. Die Mediator-topologie wird üblicherweise verwendet, wenn mehrere Schritte innerhalb eines Ereignisses durch einen zentralen *Mediator* orchestriert werden. Die Broker-topologie wird verwendet, wenn die Ereignisse ohne Hilfe eines zentralen *Mediators* verkettet werden sollen¹³.
- *Microkernel Architecture*: dieses Model ermöglicht es, der Kernanwendung zusätzliche Anwendungsmerkmale als Plug-Ins hinzuzufügen. Das bietet Erweiterbarkeit, Funktionstrennung und Isolierung¹⁴.
- *Microservices Architecture*: Seit kurzem bekommt dieses Model viel Aufmerksamkeit in der Softwareindustrie als eine Alternative zu monolithischen Anwendungen und service-orientierten Architekturen. Da das Model sich noch entwickelt, gibt es noch viele Diskussionen darüber, worum es bei dem Model geht und wie es implementiert wird. Unabhängig von der gewählten Topologie oder Implementation besteht diese Architektur aus kleinen Service-Komponenten, die weitgehend entkoppelt sind, separat bereitgestellt werden und miteinander durch Standardverfahren mit geringem Overhead wie REST, JMS oder SOAP kommunizieren¹⁵.

Für dieses Projekt wird eine Variante von *Layered Architecture* und *Microservices Architecture* verwendet. Die Software wird in drei Teilen aufgeteilt: *Client*, *Server* und *Datenbank*. Mehrere Instanzen des *Clients* / *Frontends* kommunizieren mit dem *Server* / *Backend* durch eine *Rest-API* in Form von http-Requests. Der *Server* behandelt die Requests und leitet Datenanfragen an die Datenbank durch *JDBC*. Diese drei Hauptkomponenten werden unabhängig voneinander entwickelt und in separaten Docker-Container in bereitgestellt. Jedes Teil wie *Client* und *Server* hat seine eigene *Layered Architecture* wie die Ebenen von GUI, Controller, Business-Services oder Datenanfragen, die miteinander durch festdefinierte Schnittstellen sprechen. In jeder Ebene stecken die kleinen Komponenten, die verantwortlich für Teile eines Anwendungsfalls sind.

¹³ (Richards, 2015) Seite 11

¹⁴ (Richards, 2015) Seite 21

¹⁵ (Richards, 2015) Seite 28

Diese Architektur ist ein solides Allzweckmuster, was oft als ein guter Anfangspunkt für die meisten Fälle funktioniert. Da die kleinsten Komponenten immer zu einer bestimmten Ebene gehören, können andere Ebenen als Mock in Testing simuliert werden. Testing kann einfach durchgeführt werden. Ein anderer Vorteil dieser Architektur ist der Schwierigkeitsgrad der Entwicklung wegen seiner Verbreitung. Kommt ein neuer Entwickler in das Team, kann er schnell die Codebasis verstehen und sich durch die Komponenten navigieren. Bestehende Komponenten können isoliert erweitert werden. Neue Funktionalitäten können als neue Komponenten auf gehörenden Ebenen implementiert werden¹⁶. Mehr über diese Architektur wird im Detail in den nächsten Abschnitten diskutiert.

3.3.2. Übersicht von Frontend und Backend

Die Software teilt sich in verschiedene Komponenten auf, die miteinander durch Schnittstellen kommunizieren. In diesem Abschnitt werden diese Beziehungen mit Hilfe von UML Komponentendiagrammen analysiert und strukturiert. Zunächst wird die gesamte Software abstrakt wie in Abbildung 7 dargestellt. Daher wird es gesehen, dass das ganze System drei Hauptkomponenten *Frontend*, *Backend* und *Database* enthält.

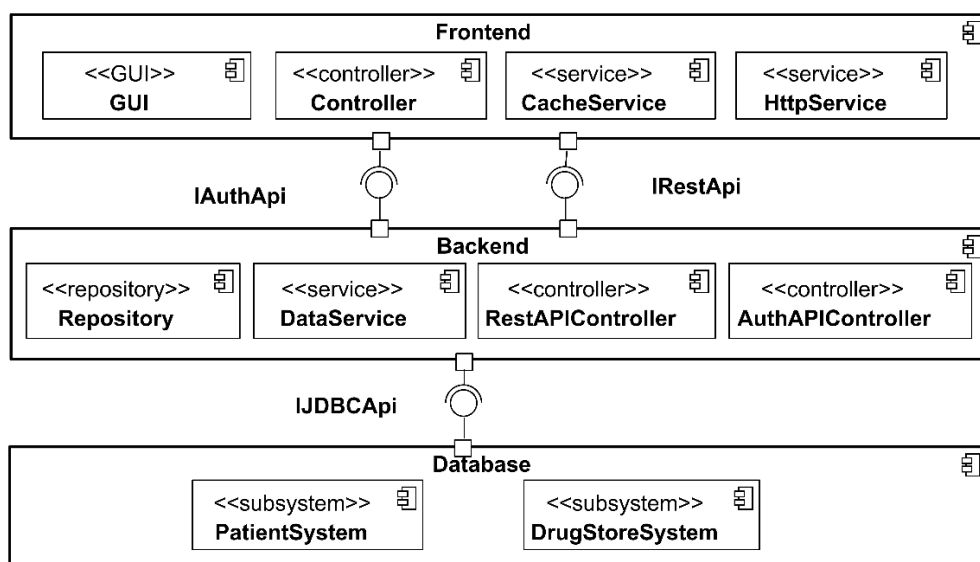


Abbildung 7: Komponentendiagramm des gesamten Systems

¹⁶ (Richards, 2015) Seite 8-9

Database ist die Datenbank, was die Anfragenlogiken aller Datenentitäten kapselt und nur die Schnittstelle *IJDBCApi* für den *Backend* zur Verfügung stellt.

Der *Backend* besteht aus den Komponenten: *Repository*, *DataService*, *RestApiController* und *AuthApiController*. Werden Datenanfragen der Datenbank gebraucht, greift die Komponente *Repository* auf die Datenbank durch die Schnittstelle *IJDBCApi* zu und bietet anderen Komponenten nicht nur CRUD-Dienstleistungen (*Create*, *Read*, *Update* und *Delete*), sondern auch verschiedene Datenfunktionalitäten an.

Als Nächstes kommt die Komponente *DataService*, welche als Zwischenebene funktioniert. Nur diese Komponente hat direkten Zugriff auf *Repository*. Falls Datenlogiken nach Bedarf wie *Filter* oder *Pagination* entstehen, werden sie hier in dieser Zwischenebene behandelt. Danach werden diese Funktionalitäten für *Controller* durch Schnittstellen zur Verfügung gestellt.

Der Rest des *Backends* sind die Komponenten *RestApiController* und *AuthApiController*. Diese Komponenten sind verantwortlich für die http-Requests aus dem *Frontend* durch die Schnittstellen *IAuthApi* und *IRestApi*. Hier werden die Dienstleistungen des *Backends* in Form von diversen Endpunkten einer RestAPI angeboten. Mehr zu der RestAPI wird es weiter im Abschnitt 4.3.3 vorgestellt. Andere Aufgaben von *AuthApiController* und *RestApiController* sind Fehlerbehandlung und Weiterleitung der Datenanfragen. Entstehen Datenanfragen im Request nach Bedarf, werden sie direkt an die Komponente *DataService* weitergeleitet. Alle Fehler, die im Responseablauf auftauchen, werden hier in *Controller* behandelt und als Hinweise an *Frontend* geantwortet.

Auf der anderen Seite funktioniert das *Frontend* als der Client des Systems, um Daten aus dem *Backend* zuzugreifen und mit Hilfe von verschiedenen *UI*-Komponenten zu rendern. Das *Frontend* enthält die Komponenten *GUI*, *Controller*, *CacheService* und *HttpService*. Diese Komponenten kommunizieren miteinander in dieser Reihenfolge durch verschiedene Schnittstellen.

Die Komponente *HttpService* greift auf das *Backend* durch die Schnittstellen *IAuthApi* und *IRestApi* in Form von http-Requests zu und bietet nächster Komponente *CRUD*-Dienstleistungen an.

Als Nächstes verwendet die Komponente *CacheService* diese *CRUD*-Dienstleistungen, speichert die benötigten Daten temporär und bietet die Daten für die anderen Komponenten durch Schnittstellen an.

Die Komponente *Controller* enthält alle Business-Logiken des Frontends und behandelt die Aktionen von Benutzern aus der Komponente *GUI* durch die Verwendung der Dienstleistungen von *CacheService* und *HttpService*.

Im Folgenden werden diese Komponenten und ihre internen Komponenten in Detail erklärt.

Auf Grund von zeitlichem Umfang und zur Verständlichkeit werden nur Komponenten des Subsystems von Medikamentengeschäft betrachtet.

3.3.3. Frontend

Zuerst im Subsystem Medikamentengeschäft beinhaltet die Komponente *GUI* des *Frontends* in Abbildung 8 die Komponenten *ReportGUI*, *ProductGUI*, *OrderDetailGUI* und *OrderGUI*, mit denen die Benutzer interagieren kann. Diese Komponenten enthalten die Bausteine der Benutzeroberfläche wie Seiten, Texte, Buttons oder klickbare Elemente. Sie erhalten die Aktionen der Benutzer vom Bildschirm, behandeln sie durch die Verwendung der Dienstleistungen von der Komponente *Controller*- und zeigen dem Benutzer Rückmeldungen auf dem Bildschirm als Feedback.

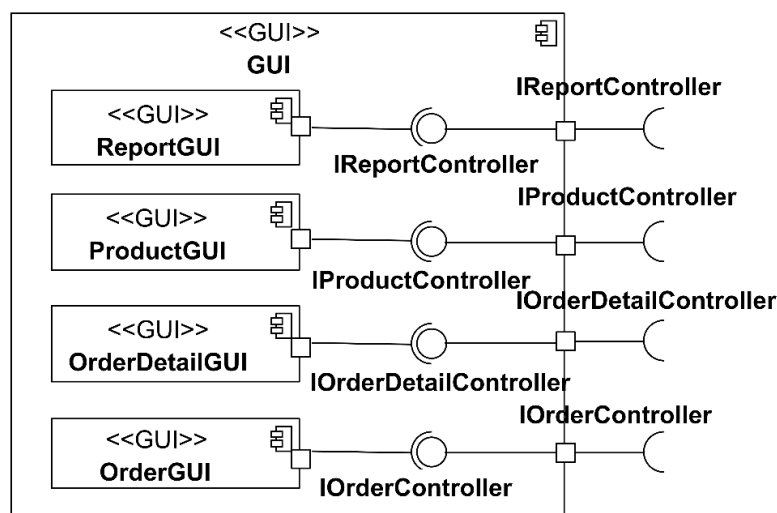


Abbildung 8: Komponentendiagramm der Frontend-GUI-Komponente

Auf der nächsten Ebene befindet sich wie in Abbildung 9 die Komponente *Controller*, die die internen Komponenten *ReportController*, *ProductController*, *OrderDetailController* und *OrderController* enthält. Hier werden die Aktionen der Benutzer als Business Logiken behandelt. Abhängig davon, ob eine Aktion Daten von Backend oder lokalen Daten braucht, wird auf Schnittstellen anderer Ebenen wie *CacheService* und *HttpService* zugegriffen.

Eine besondere Dienstleistung in dieser Komponente ist die *IAuthService*, welche gelb gekennzeichnet ist und von allen Komponenten verwendet wird. Diese funktioniert als zentrale Authentifizierungsdienstleistung des *Frontends*, um den Authentifizierungsstatus des Benutzers zu verwalten. Falls sich der Benutzer anmeldet, abmeldet oder unberechtigten Zugriff auf ein bestimmtes Element hat, werden alle Komponenten durch die Schnittstelle *IAuthService* benachrichtigt.

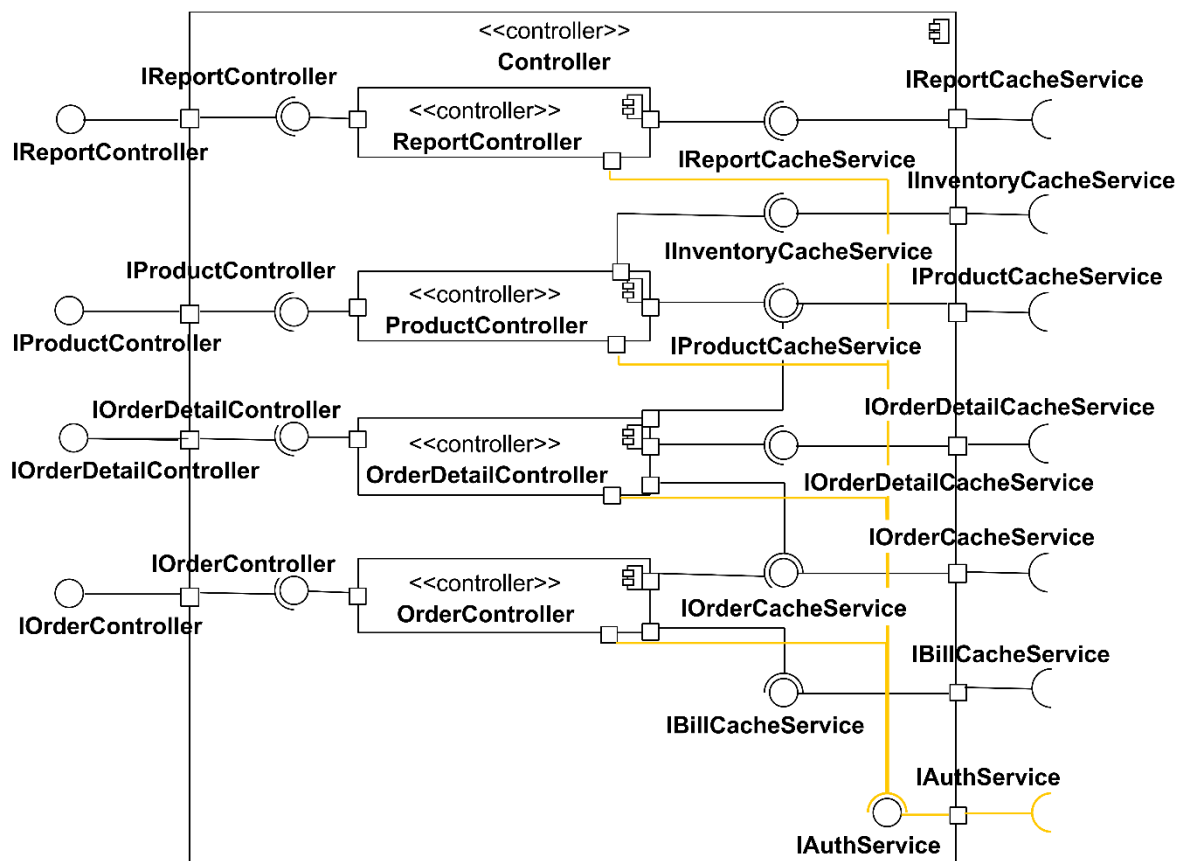


Abbildung 9: Komponentendiagramm der Frontend-Controller-Komponente

Falls die Komponente *Controller* Daten braucht, greift sie zuerst auf die nächste Ebene *Cache-Service* zu. Wie in Abbildung 10 angezeigt besteht dieser Baustein aus den Komponenten *ReportCacheService*, *InventoryCacheService*, *ProductCacheService*, *OrderDetailCacheService*, *OrderCacheService* und *BillCacheService*.

Diese Ebene funktioniert als temporäre Datenquelle für *Controller*. Falls Bedarf auf Daten z.B. von *Product*, oder *OrderDetail* entsteht, werden zuerst lokale Daten verwendet. Wenn das nicht der Fall ist, wird auf die nächste Ebene *HttpService* durch ihre Schnittstellen zugegriffen, um Daten aus dem *Backend* abzufragen und die lokalen Daten zu aktualisieren.

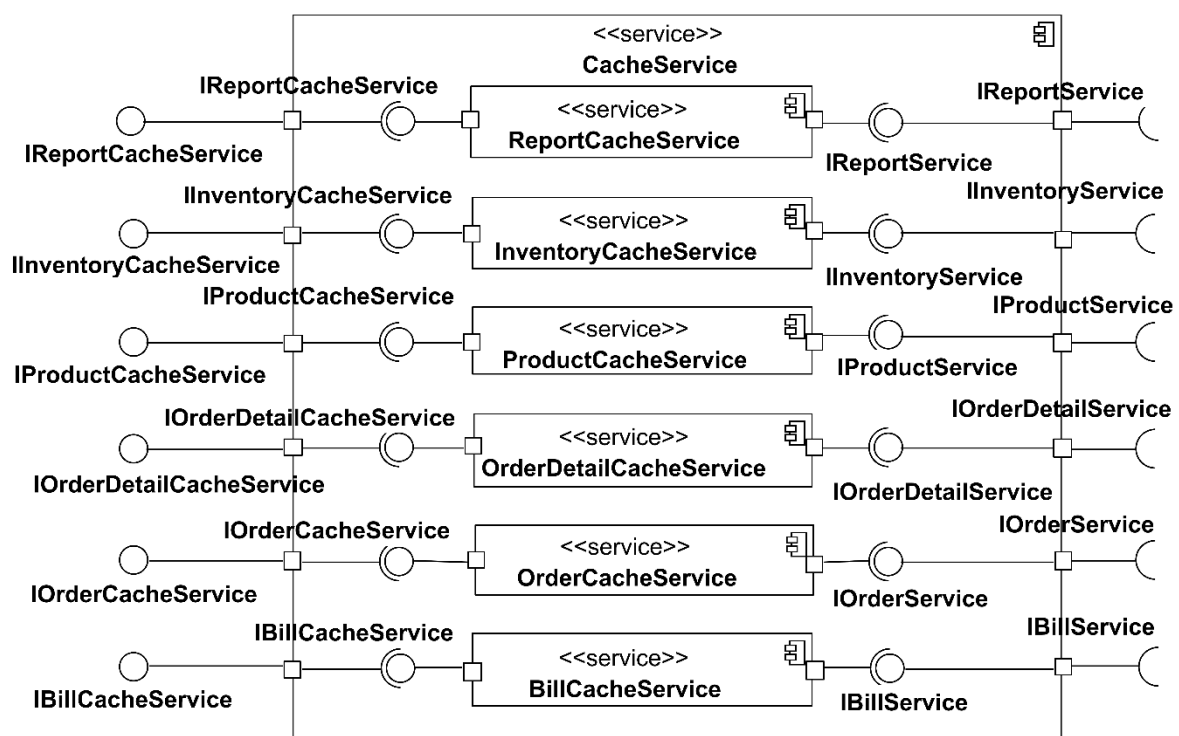


Abbildung 10: Komponentendiagramm der Frontend-CacheService-Komponente

Als nächstes wird die Komponente *HttpService* wie in Abbildung 11 mit ihren internen Komponenten *ReportService*, *InventoryService*, *ProductService*, *OrderDetailService*, *OrderService* und *BillService* dargestellt. Diese Ebene beinhaltet für die anderen Ebenen *CRUD* Funktionalitäten, damit auf den Server durch *http-Requests* zugegriffen werden kann. Unter anderem bietet die Ebene auch die Authentifizierungsinfo durch *AuthService* an. Wie oben im Abschnitt für *Controller* beschrieben, wird *AuthService* für alle Komponenten, sogar für die internen Komponenten dieser Ebene, verwendet, um den Zustand der Authentifizierung zu vermitteln. Die Funktionalität bezüglich der *AuthService* wird durch die Schnittstelle *IAuthApi* in Form von

http-Requests realisiert. Möglichkeiten zur Persistenz der anderen Daten wie Produkte, Bestellungswünsche oder Bestellungen werden durch die Schnittstelle *IRestApi* an den *Backend* zur Verfügung gestellt. Wie die *IAuthApi* und *IRestApi* aufgebaut werden, wird in Detail im nächsten Abschnitt erklärt.

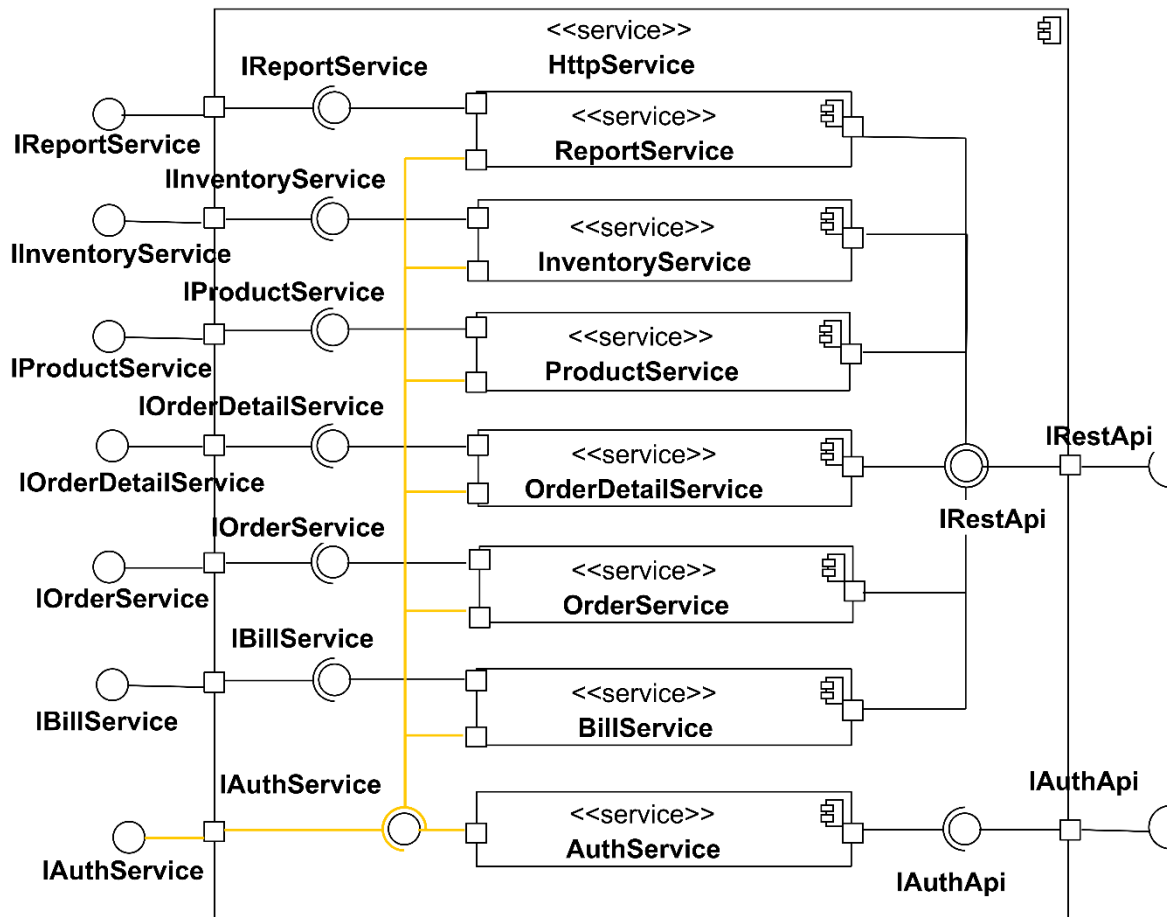


Abbildung 11: Komponentendiagramm der Frontend-HttpService-Komponente

3.3.4. Rest API

In diesem Abschnitt wird die Kommunikationsschnittstelle *RestAPI* zwischen dem *Frontend* und dem *Backend* vorgestellt. *RestAPI* ist die Abkürzung für *Representational State Transfer – Application Programming Interface*, welche von Roy Fielding entwickelt wurde. Das ist ein Standard zum Datenaustausch zwischen verschiedenen Systemen, hier in diesem Fall dem *Frontend* und *Backend* ¹⁷. Eine RestAPI ermöglicht die http-Requests wie z. B. GET, POST, PUT

¹⁷ (Madden, 2020) Seite 8

und DELETE durch diverse verfügbare Endpunkte und ihre benötigten Parameter auf dem *Backend*, damit der *Frontend* die verfügbaren CRUD-Operationen zugreifen kann.

Diese RestAPI abstrahiert dem *Frontend* die Datenanfragen. D. h. das *Frontend* weiß aus seiner Sicht nicht, wie diese http-Requests behandelt werden oder woher die Daten kommen. Es stimmt nur mit dem *Backend* ab, wie diese Endpunkte der RestAPI aufgebaut sind, wie die aufgerufen werden, welche Parameter dazu benötigt werden, wie die http-Responses im Normalfall und im Fehlerfall aussehen sollen.

Die Schnittstellen *IAuthApi* und *IRestApi*, die im oberen Abschnitt von *HttpService* erwähnt wurden, bauen zusammen die gesamte RestAPI dieser Software. Es werden nach Konvention verschiedene Endpunkte mit Abstimmungen wie Parameter, Rückgabewert, Rollen...zwischen dem *Frontend* und dem *Backend* vorgestellt.

In Tabelle 4 unten werden einige Endpunkte der RestAPI genauer beschrieben. Die komplette Tabelle kann im Appendix angesehen werden. Es gibt vier Spalten in der Tabelle: *Endpunkte*, *http*, *Rückgabewert* und *Rollen*. *Endpunkte* beschreibt, wie ein Endpunkt mit Parametern aussieht. Die Spalte *http* zeigt an, welche http-Methode für diesen Endpunkt akzeptiert wird: *GET*, *POST*, *PUT* oder *DELETE*. Danach kommt *Rückgabewert* als ein kurzer Hinweis zum http-Response des Endpunktes. Hier werden meistens die Datenobjekte des Klassendiagramms als Rückgabetyt verwendet. Werden bestimmte Rechte für den Endpunkt erfordert, werden die erforderlichen Rechte bzw. Rollen für den jeweiligen Endpunkt in der Spalte *Rollen* angezeigt.

Endpunkte	http	Rückgabewert	Rollen
/auth/login	POST	User mit JWT-Token	Public
/auth/logout	POST		Public
/auth/register	POST		Public
/api/patients	GET	List<Patient>	Alle
/api/patients/{patientId}	GET	Patient	Alle
/api/patients	POST	Patient	Alle
/api/patients/{patientId}	PUT	Patient	Alle

/api/visits/{visitId}/ prescriptions	GET	List<Prescription>	Manager, Doctor
/api/visits/{visitId}/ prescriptions/{prescriptionId}	GET	Prescription	Manager, Doctor
/api/visits/{visitId}/ prescriptions	POST	Prescription	Manager, Doctor
/api/visits/{visitId}/ prescriptions/{prescriptionId}	PUT	Prescription	Manager, Doctor
/api/visits/{visitId}/ prescriptions/{prescriptionId}	DELETE		Manager, Doctor
/api/orders	GET	List<Order>	Alle
/api/orders/{orderId}	GET	Order	Alle
/api/orders/{orderId}/pdf	GET	Order als Pdf-Datei	Alle
/api/orders	POST	Order	Alle
/api/orders/{orderId}	PUT	Order	Alle
/api/orders/{orderId}	DELETE		Alle
/api/orders/{orderId}/ orderDetails	GET	List<OrderDetail>	Alle
/api/orders/{orderId}/ orderDetails/{orderDetailId}	GET	OrderDetail	Alle
/api/orders/{orderId}/ orderDetails	POST	OrderDetail	Alle
/api/orders/{orderId}/ orderDetails/{orderDetailId}	PUT	OrderDetail	Alle

Tabelle 4: Beschreibung einiger Endpunkte der-RestAPI zwischen Backend und Frontend

3.3.5. Backend

In diesem Abschnitt werden die Strukturen des *Backends* mit Hilfe von einigen Komponentendiagrammen auseinandergesetzt.

Zuerst sehen die Komponenten *RestApiController* und *AuthApiController* mit ihren internen Komponenten wie in Abbildung 12 so aus. Das ist die erste Ebene, welche alle http-Requests des *Frontends* behandelt. Genauer gesagt befinden sich in diesen internen Komponenten die Endpunkte der RestAPI, was im oberen Abschnitt vorgestellt wurden.

Die Komponente *AuthController* ist verantwortlich für die Endpunkte: */auth/login*, */auth/logout* und */auth/register*. Werden die Benutzerdaten bzw. die Authentifizierungsfunktionen benötigt, wird es durch die Schnittstelle *IAuthService* aus der nächsten Ebene zugegriffen.

In der Komponente *RestApiController* besitzt die Schnittstelle *IAuthService* die ähnliche Funktionalität wie im *Frontend*, um Benutzerrollen bzw. Benutzerechte für alle anderen Komponenten anzubieten. Hier befinden sich die anderen Endpunkte der *RestAPI*. Nachdem die http-Request an den Endpunkten ankommen und geeignete Rechte besitzen, werden Datenanfragen an die nächste Komponente *DataService* durch verschiedene Schnittstelle wie in Abbildung 12 weitergeleitet. Unter anderem werden die Logiken der Fehlerbehandlung hier implementiert.

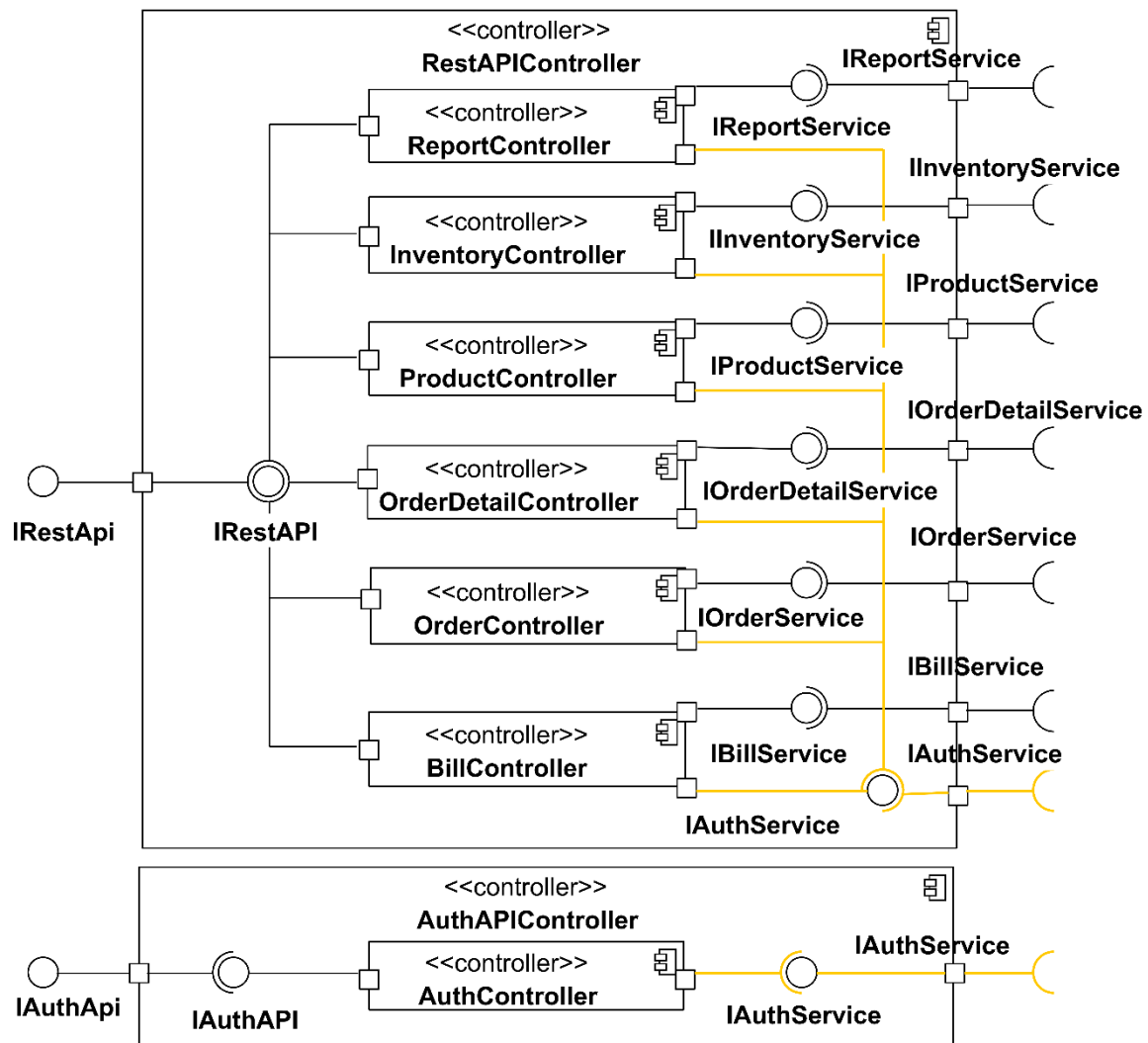


Abbildung 12: Komponentendiagramm der Backend-APIController-Komponente

Als nächstes kommt die Komponente *DataService*, was die Datenanfragenlogiken vom *Controller* abstrahiert. Datenanfragenlogiken sind genauer gesagt die Operationen an einer Entität z.B. Abfragen, Anlegen, Modifikation, Löschen, Filter und Pagination. Für die Komponenten *ReportService* und *BillService* entstehen noch Bedarf für das Ausstellen der täglichen bzw. monatlichen Reporte und Rechnungen. Deswegen können diese Dienstleistungen aus dieser Ebene auch implementiert werden. Einen direkten Zugriff auf die Datenbank hat die Ebene *DataService* nicht, sondern es wird nur durch die verfügbaren Schnittstellen der Komponente *Repository* ermöglicht.

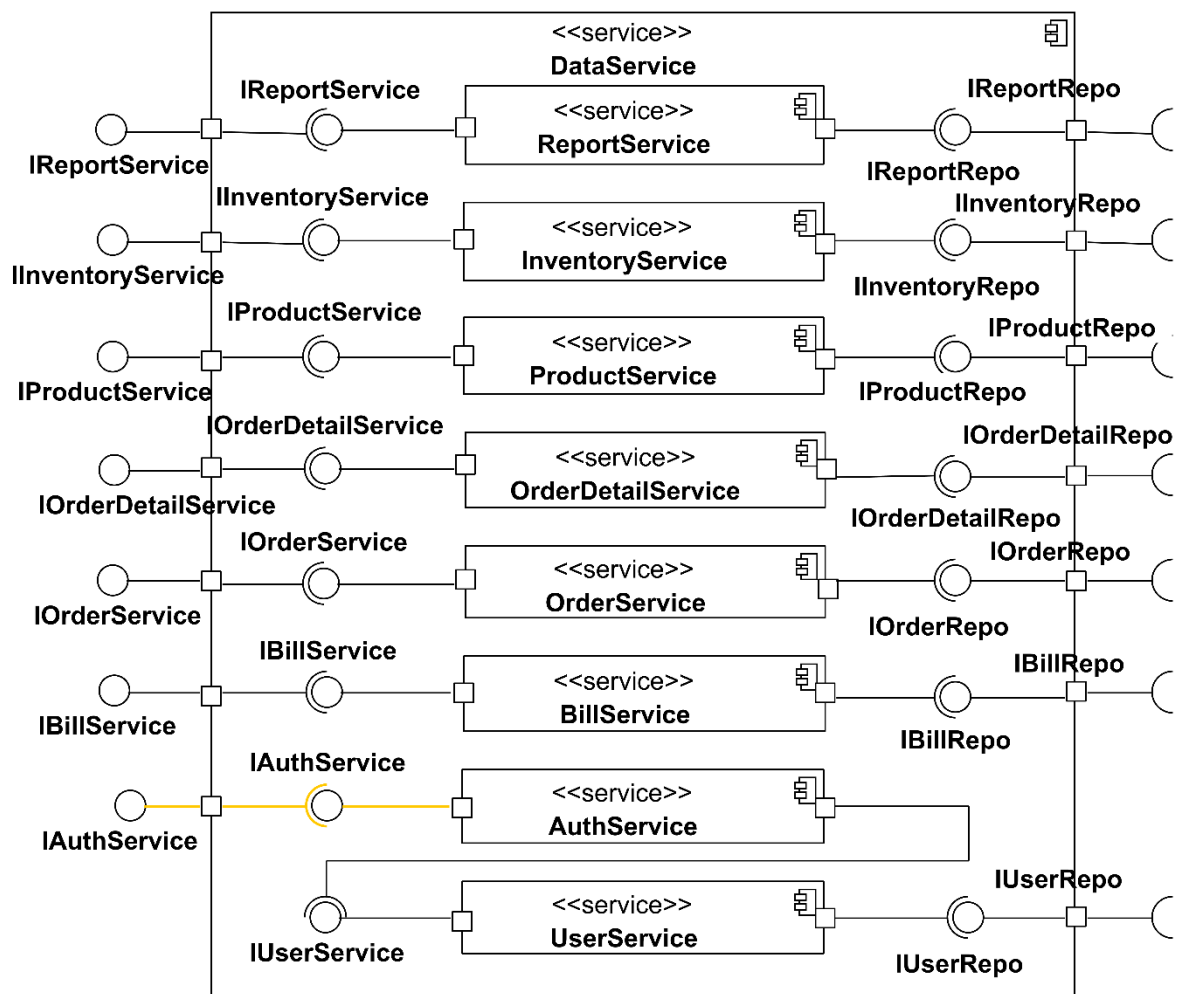


Abbildung 13: Komponentendiagramm der Backend-DataService-Komponente

Zuletzt wie in Abbildung 14 spielt die Komponente *Repository* die Rolle als Kommunikationsmittel zwischen dem *Backend* und der Datenbank. Hier werden die Entitäten im Abschnitt *Entity-Relationship-Diagramm* mit Hilfe von einer Objektrelationale Abbildung in Tabellen der Datenbank gemappt. Die direkten Datenbankoperationen an Tabellen in SQL werden vom *Backend* abgedeckt und nur durch die Schnittstelle *IJDBCapi* angeboten. D. h., die Datenanfragen können komplett in der Programmiersprache des *Backend* durchgeführt werden. Falls Bedarf für komplexere Operationen an Entitäten entsteht, kann dies immer in SQL realisiert werden.

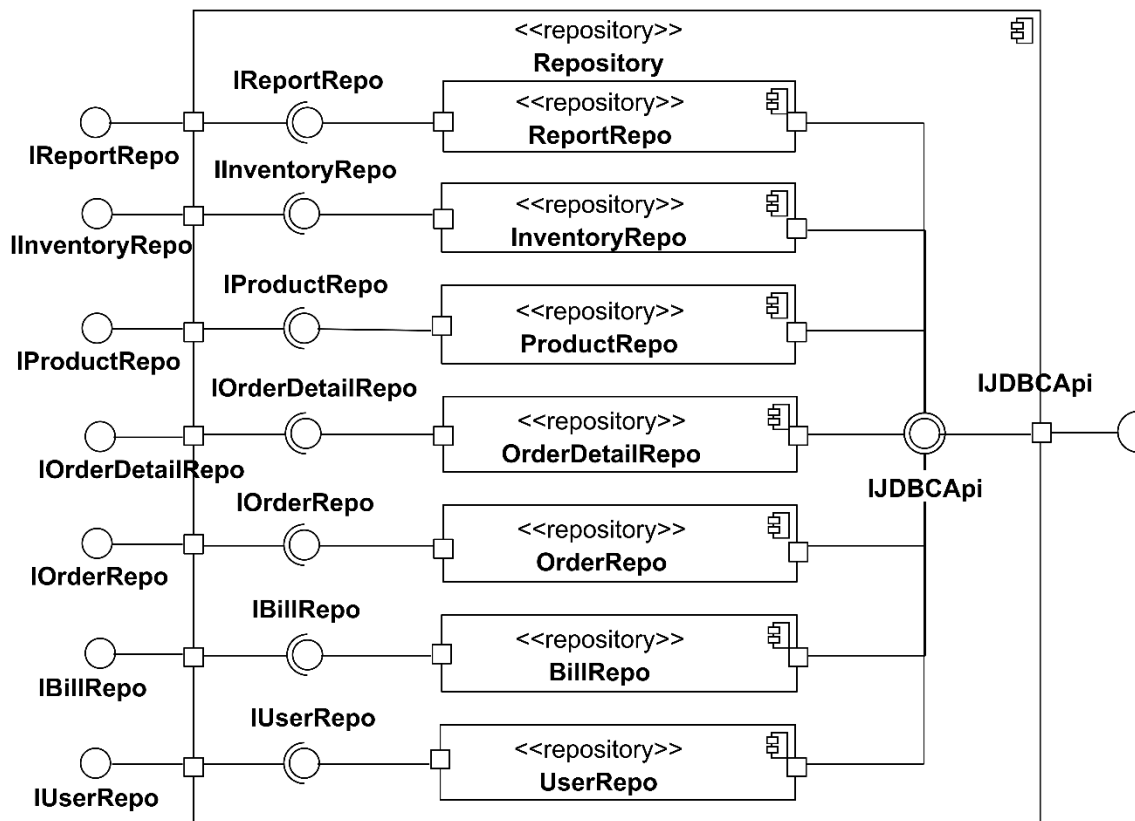


Abbildung 14: Komponentendiagramm der Backend-Repository-Komponente

3.4. Anwendungsumgebungen

In diesem Abschnitt wird die Architektur der Softwarebereitstellung beschrieben. Es werden hier die Entwicklungsumgebung bzw. Produktionsumgebung mit Hilfe von zwei Verteilungsdiagrammen analysiert.

3.4.1. Entwicklungsumgebung

Zuerst sieht die Entwicklungsumgebung der Software wie in Abbildung 15 aus. Das Diagramm zeigt, was für Bausteine und ihre Beziehungen die Software zur Laufzeit enthält, wenn sie in der Entwicklungsphase ausgeführt wird.

Das Ganze läuft unter einem persönlichen Rechner unter dem Betriebssystem Windows 10. Im `<<executionEnvironment>>` Windows 10 befinden sich zwei andere `<<executionEnvironment>>`: Chrome und Docker Host. Docker Host ist eigentlich Docker Engine, was die Rolle als

eine Ausführungsumgebung aller Containers spielt und verantwortlich für Kernelaufrufe zwischen mehreren Containern und dem Host-Betriebssystem spielt. In Docker Host gibt es zwei Container: *Frontend* und *Backend*.

Der *Frontend* Container läuft komplett unter einem Linux-Betriebssystem, mit der JavaScript-Ausführungsumgebung Node.js vorinstalliert. Node.js funktioniert als ein Webserver, welcher statische Dateien anbietet. Die benötigten statischen Dateien sind z. B. index.html, scripts.js und styles.css, welche als Ergebnisse vom Frontend-Code gebaut wurden.

Nebenbei besitzt der *Backend Container* auch ein separates Linux-Betriebssystem mit *Tomcat* als Webserver für Java und *H2* als die temporäre relationale Datenbank. Der ganze Backendcode in Java wird in eine Datei *server.jar* verpackt, was durch Tomcat für andere Container bzw. Software zur Verfügung gestellt wird.

Da es um eine Entwicklungsumgebung geht, kann der Browser Chrome auf demselben Rechner mit Docker Host laufen. Trotzdem werden von der Sicht des Browsers Chrome die internen Bausteine der Software in den Containern abgedeckt. Möchte der Benutzer die Software testen, wird Chrome einmal geöffnet und der erste http-Request an den *Frontend* Container geschickt. Daher werden die statischen Dateien des *Frontends* als http-Response zurück an Chrome beantwortet. Entstehen noch weitere http-Requests für die Datenanfragen, werden die Requests von Chrome durch den *Frontend* Container an den *Server* Container weitergeleitet. Der Ablauf ist ähnlich wie im Abschnitt *Komponentendiagramme* beschrieben.

Der Vorteil dieses Verteilungsdiagramms hier ist nur, dass es abstrakter dargestellt wird. Ein anderer Vorteil liegt an der Verwendung von Docker. Wird eine eindeutige Entwicklungsumgebung für ein ganzes Entwicklungsteam mit mehreren Mitgliedern oder wird die Software zusammen auf verschiedenen Rechnern entwickelt, wird gar keine Konfiguration auf den einzelnen Rechnern eingerichtet, um das ganze System auszuführen, außer nur den *Docker Client* zu installieren. Die Konfiguration der Umgebung wird in Docker-Dateien bzw. *Docker-Compose*-Dateien geschrieben und ist damit ohne Probleme übertragbar.

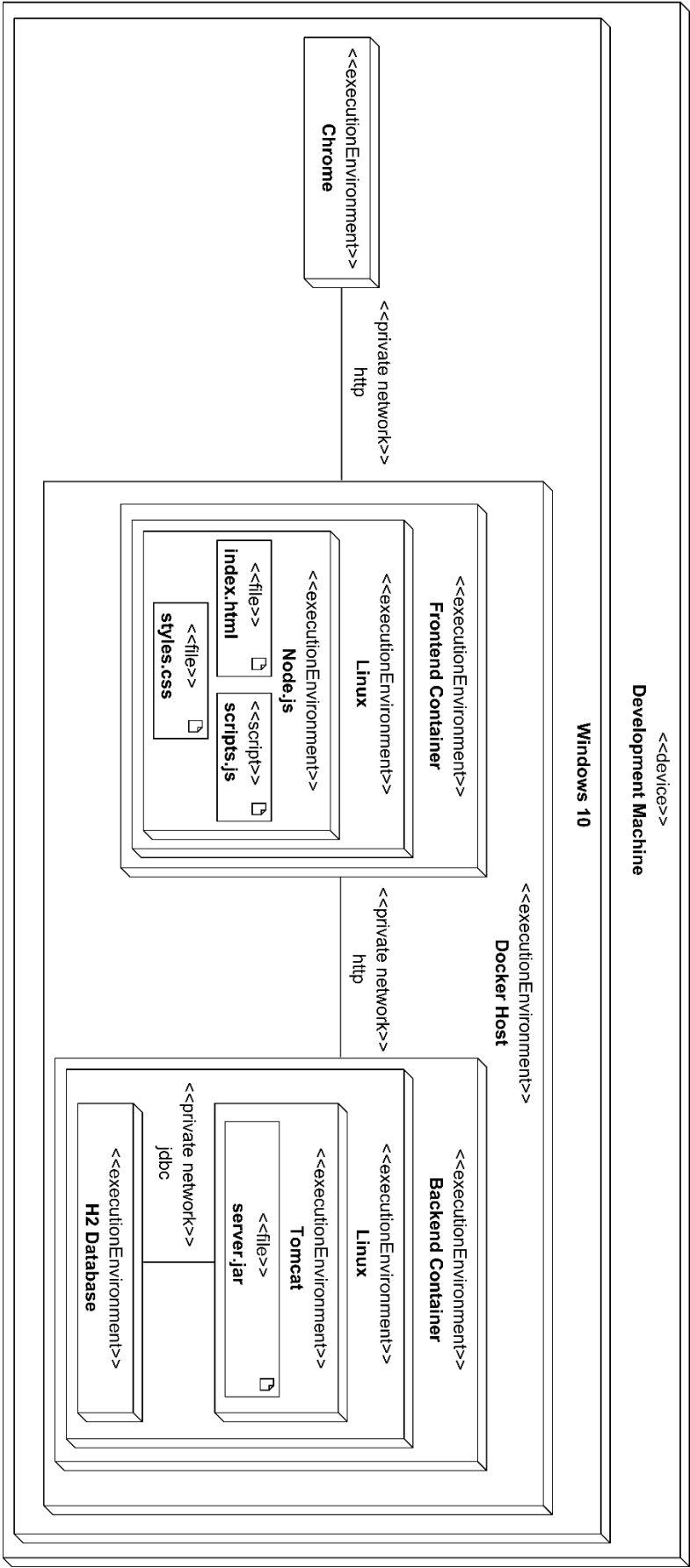


Abbildung 15: Verteilungsdiagramm des Entwicklungssystems

3.4.2. Produktionsumgebung

Als Nächstes kommt die Produktionsumgebung der Software auch in Form von einem Verteilungsdiagramm wie in Abbildung 16 angezeigt. Die Struktur beider Umgebungen ist sehr ähnlich außer einiger Unterschiede, was in Folgendes in Detail vorgestellt werden.

Auf der rechten Seite befindet sich der Server Rechner, wo die Software bereitgestellt wird. Dieser Rechner wird später als eine virtuelle Maschine von *Digital Ocean* gemietet. Es wird nicht mehr Windows 10 als Betriebssystem gewählt, sondern Linux. Unter Linux läuft auch ein `<<executionEnvironment>>` und Docker Host mit drei Containern *Frontend*, *Backend* und *Database*. Jeder Container hat wie früher auch ein Linux-Betriebssystem vorinstalliert.

In dem *Frontend* Container wird nun NGINX als der Webserver verwendet. Vorteil davon ist seine Load-Balancer-Fähigkeit und die Performanz zum Anbieten der statischen Dateien. Der *Backend* Container ist ähnlich wie in der Entwicklungsumgebung und tauscht Daten mit dem *Frontend* durch *http*-Requests. Nun wird die Datenbank in diesem Fall als ein separater Container umgewandelt. Es wird PostgreSQL als die Produktionsdatenbank, was mit dem *Backend* durch die Schnittstelle *JDBC* kommuniziert, gewählt. Die drei Containers werden komplett in diesem Docker Host ausgeführt werden dank einer *Docker-Compose*-Datei, wo alle Konfigurationen für die Produktionsumgebung festgestellt werden können. Daher können diese Container bzw. diese Software mit den exakten Konfigurationen auf einen entfernten Rechner bereitgestellt. *Digital Ocean* war nur eine Wahl für Produktionsrechner in diesem Projekt. Die Software kann ohne weitere Konfigurationen auf einen anderen Rechner zur Verfügung gestellt werden.

Auf der linken Seite liegen verschiedene Client Geräte als eine Darstellung, zu zeigen, wie die Software von entfernt durch das Internet zugegriffen werden. Die Betriebssysteme der Geräte lauten Windows 10, iOS und Android. Das sind die Plattformen, für die diese Software eine Version geliefert werden muss. Da Native-Apps keine Anforderung waren, kann die Software einfach von irgendeinem Webbrowser des Gerätes geladen werden. Unter Windows 10, Android werden Chrome gewählt und unter iOS ist Safari die Wahl.

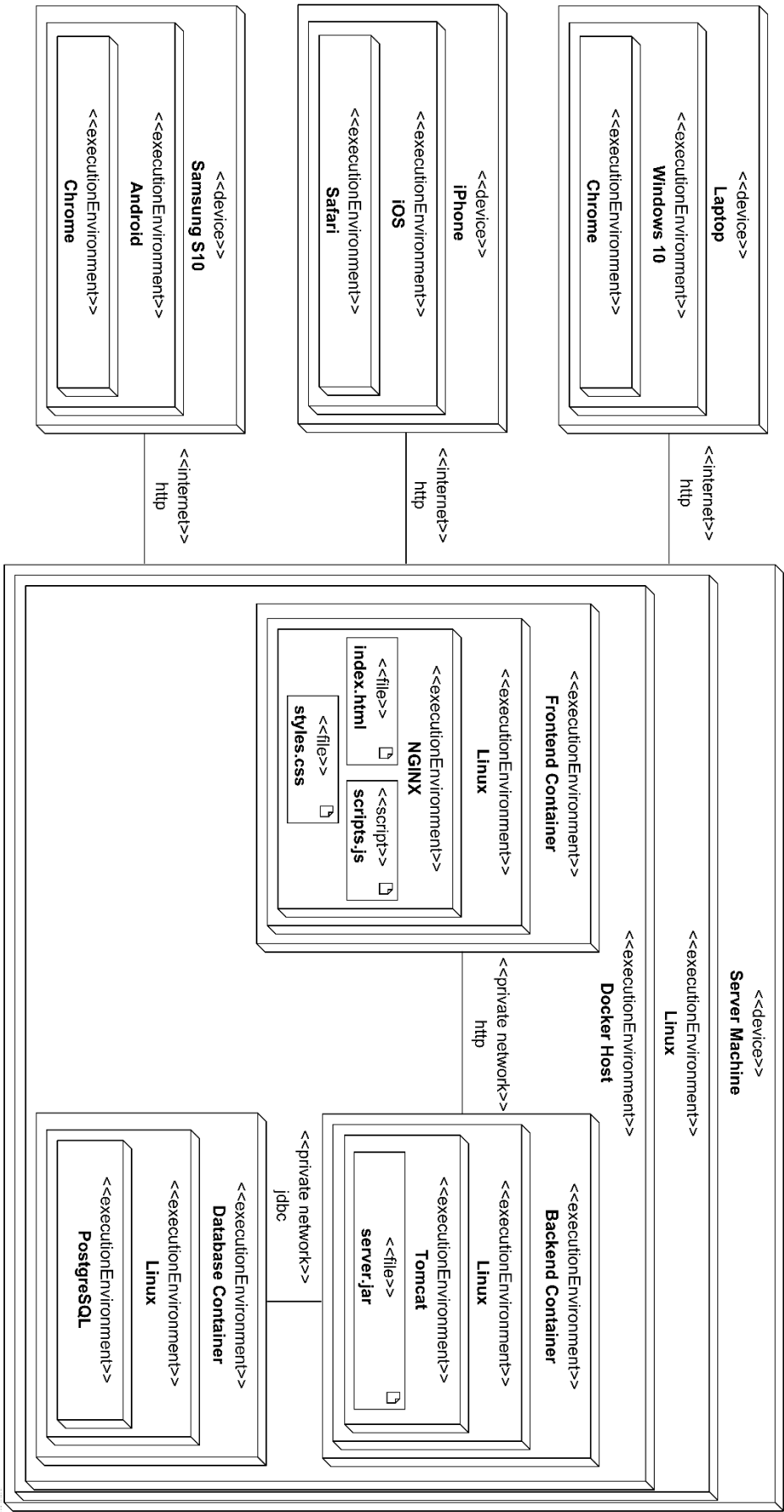


Abbildung 16: Verteilungsdiagramm des Produktionssystems

4. Technologien

In den folgenden Abschnitten werden einige technischen Technologien, die für diese Bachelorarbeit verwendet werden, vorgestellt. Es wird vorrangig auf Open Source Bibliotheken gesetzt, da diese kostenfrei zu nutzen sind, eine hohe Verbreitung aufweisen und öffentlich zugängliche Dokumentationen und Communities haben.

Es wird dabei auch eine kurze Time-to-Market gefördert, da Einarbeitungsaufwände in neue Technologien entfallen. Zu den weiteren Auswahlkriterien gehören die Marktrelevanz, Skalierbarkeit, der Lebenszyklus und die Kosten einer eingesetzten Komponente.

4.1. Angular

Als eines von mehreren etablierten *Frameworks* für *Single Page Applications* ist *Angular* das mit Abstand am weitesten verbreitete. *Angular* basiert hauptsächlich auf vielen Komponenten, die miteinander kommunizieren können. Jede Komponente enthält nur wenige Informationen und übernimmt im besten Fall nur eine kleine Verantwortung, damit sie in der ganzen Lebensdauer der Software besser entwickelt und gepflegt werden kann. Daher werden die Benutzerschnittstellen der *Microservices* mit *Angular* umgesetzt. *Angular* unterstützt die Nutzung von *TypeScript* als Programmiersprache, was neben der nativen Umsetzung mit *JavaScript* weniger fehleranfällig ist und eine bessere Wartbarkeit zur Folge hat.

Das Lizenzmodell wird durch die MIT Lizenz zur kostenfreien gewerblichen Nutzung verwendet¹⁸.

4.2. Spring Framework und Spring Boot

Das *Spring Framework* besteht aus einer Core-Komponente und zahlreichen Adaptern für weitere Technologien und Frameworks. Das *Spring Framework* bietet Adapter für unterschiedliche Aufgaben.

- Persistenz von Daten in Datenbanken
- Kommunikation über asynchrone und synchrone Schnittstellen

¹⁸ (Google, 2021)

- Umsetzung von technischen Schnittstellen
- Authentifizierung über gängige Protokolle
- Autorisierung von Benutzern beim Zugriff auf spezifische Ressourcen

Das Lizenzmodell wird durch die Apache Lizenz zur kostenfreien gewerblichen Nutzung verwendet.

Mittels *Spring Boot* wird eine Laufzeitumgebung für Java-Anwendungen bereitgestellt. Mit Hilfe von *Spring Boot* ist es einfacher, mit wenigen Konfigurationen *Spring*-basierte Software auszuführen. Zu den unterstützten Features gehört unter anderem:

- Bereitstellung eigenständig lauffähiger Anwendungen
- Konfigurationsmanagement
- Automatische Konfiguration von Drittanbieterkomponenten
- Bereitstellung von Metriken zur Applikationsüberwachung

Das Lizenzmodell wird durch Apache Lizenz zur kostenfreien gewerblichen Nutzung verwendet¹⁹.

4.3. PostgreSQL

PostgreSQL (Postgres) ist ein leistungsstarkes, relationales *Open-Source* Datenbanksystem, was nicht nur die SQL-Sprache verwendet und erweitert, sondern auch komplexere Daten-Workloads sicher speichert und skaliert. Postgres bietet Merkmale, die Entwicklern und Administratoren dabei helfen, Applikationen bzw. fehlertolerante Umgebungen zu bauen, Datenintegrität zu sichern und Daten unabhängig von ihrer Größe zu verwalten.

Auf einer Seite kann *Postgres* zur kostenlosen gewerblichen Nutzung verwendet und durch eine *Open-Source* Community gepflegt werden. Auf der anderen Seite ist Erweiterbarkeit auch eine Stärke von *Postgres*, indem Benutzer die eigenen Datentypen, Funktionen oder sogar Code aus anderen Programmiersprachen schreiben dürfen.

Das Lizenzmodell wird durch die MIT Lizenz zur kostenfreien gewerblichen Nutzung verwendet²⁰.

¹⁹ (VMware, 2021)

²⁰ (Group, 2021)

4.4. Docker

Docker ist eine freie Technologie zur Isolierung von Anwendungen mit Container Virtualisierung. *Docker* stellt die Laufzeitumgebung für die *Microservices* bereit und ermöglicht durch die Modularisierung ein einfaches Deployment. *Docker* ist eine Linux Technologie, die in den etablierten Cloud-Umgebungen verfügbar ist. Dadurch können diese Container lokal auf Entwicklungssystemen ausgeführt und später in der Cloud betrieben werden.

Das Lizenzmodell wird durch Apache Lizenz zur kostenfreien gewerblichen Nutzung verwendet²¹.

4.5. Digital Ocean

Digital Ocean ist ein *Cloud Computing* Anbieter, welcher einen benutzerfreundlichen Einrichtungsaufbau und einen planbaren, erschwinglichen Preis bietet. Benutzer, die einen autoskalierbaren, zuverlässigen Cloud-System für kleine bis mittelgroße Software brauchen, sind Zielkunden von *Digital Ocean*.

Folgendes sind einige Dienstleistungen, welche *Digital Ocean* bietet:

- Mögliche Multiinstanzen von Server mit *Docker* auf einem Konto
- Schneller Zugriff auf Server und zuverlässige Betriebszeit
- Planbaren und günstigen Preis
- *Infrastructure as a Service*: mögliche Verwaltung von Sicherheit, Datenbank und Betriebssysteme

Das Lizenzmodell wird für kostenfreie gewerbliche Nutzung kostenpflichtig je nach Bedarf verwendet. Der günstigste Preis für eine virtuelle Maschine liegt bei \$5/ Monat²².

²¹ (Docker, 2021)

²² (DigitalOcean, 2021)

4.6. Zusammenfassung

Da eine Anforderung feststellt, dass die Software auf *Desktop Web*, *iOS* und *Android* zur Verfügung gestellt werden soll, spielt *Angular* hier die richtige Rolle, was ein natives *User Experience* auf Browser bringt. Die einzige Codebase von *HTML*, *CSS* und *JavaScript* wird mit Hilfe von *Angular* auf den oben genannten Plattformen durch *Mobile Web* dargestellt.

Angular hilft auch dabei, den *Frontend*-Code nach Komponenten und Klassen zu strukturieren. Unter anderem funktionieren *Services* von *Angular* als Middleware, welche mit dem *Backend* durch http-Requests kommuniziert und den Zwischenzustand des *Frontends* speichert.

Zu *Backend*-Anforderungen wie Authentifizierung der Benutzer, Persistenz der verschiedenen Daten, Persistenz der Bilder wird *Spring Framework* gewählt, was eine sichere rollenbasierende *Rest-API* an *Frontend* anbieten kann und Daten in die bzw. von der *Postgres*-Datenbank schreibt bzw. abfragt.

Um die Entwicklungs- und Produktionsumgebung möglich mit wenigen Konfigurationen auf mehrere Rechner zu halten, wird jeder Teil des Systems wie *Frontend*, *Backend* und Datenbank in einem *Docker* Container zur Verfügung gestellt. Diese *Docker* Containers werden dann in einem Container als gesamtes System verpackt. Der gesamte Container ermöglicht einen einfachen Deployment-ablauf an die virtuelle Maschine von *Digital Ocean*, da der Container die genauen Konfigurationen inklusiv das Betriebssystem und die Abhängigkeiten aller Komponenten des Systems halten kann oder nach Bedarf modifiziert werden kann, falls die Produktionsumgebung mit der Entwicklungsumgebung abweicht.

5. Implementierung & Testing

Dieses Kapitel beschreibt die Phasen der Prototyperstellung, Implementierung und Testing für die Software. Im Folgenden werden die Konzeptionen des dritten Kapitels über Architekturen umgesetzt. Dabei wird die Einstellung der Programmierungs- und Testumgebung beschrieben.

5.1. Mock Up-GUI

Zunächst werden verschiedene Mock-Up-GUI-Seiten der Software vorgestellt. In diesem Abschnitt werden nur die Seiten *Sign In*, *Products*, *Product Create-Edit*, *Order Details*, *Bill Template* und *Order Detail Create-Edit* erstellt. Die weiteren Mock-Up-GUIs können nach Bedarf im Appendix angesehen werden. Da eine Anforderung festlegt, dass die Software eine responsive Version auf dem Webbrowser auf Desktop und Mobilgeräten zur Verfügung stellen soll, werden für jede Seite zwei Versionen in der Reihenfolge von links nach rechts Desktop und Mobilgeräte dargestellt.

In Abbildung 17 ist ein Mock-Up der Seite *Sign In* dargestellt. Um sich anzumelden, muss der Benutzer seinen Benutzernamen und sein Passwort angeben. Werden Benutzername und Passwort nicht richtig eingegeben, kann der Benutzer nicht weitergeleitet werden und werden Fehler in rot markiert bzw. für den Benutzer angezeigt.

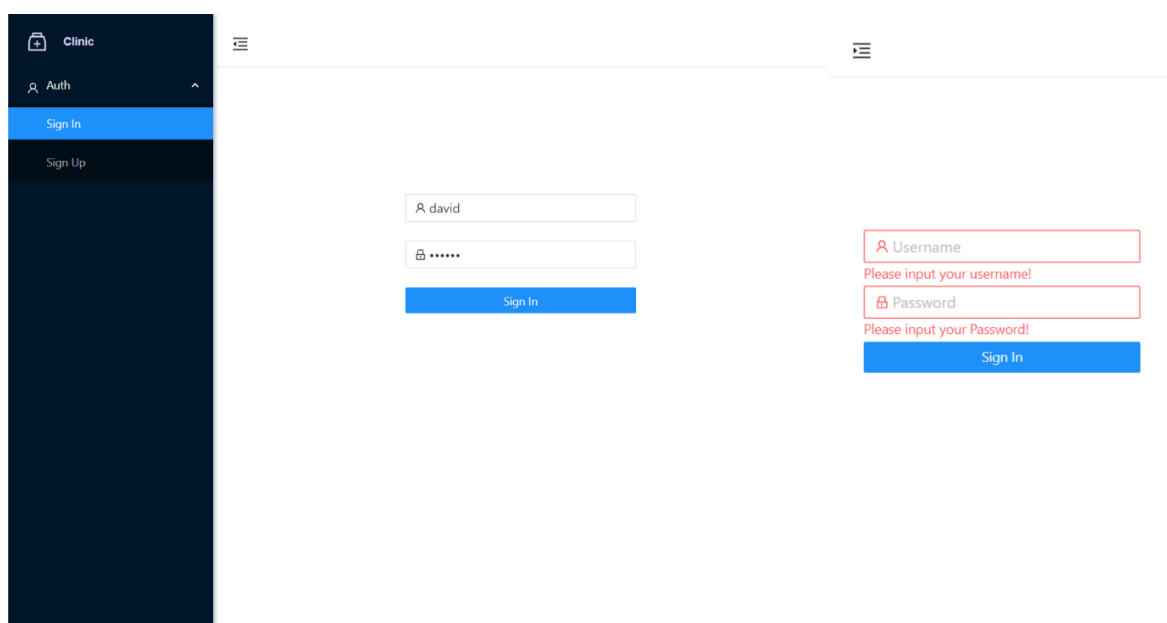


Abbildung 17: Sign In Page

Implementierung & Testing

Die Seite *Products* wird wie in Abbildung 18 modelliert, um die gesamten Medikamente im System anzuzeigen. In der Desktopversion werden die Medikamente in einer Tabelle und in Mobilegeräten in Form von Karten angezeigt. Die Liste aller Medikamente wird nicht komplett auf einmal geladen, sondern nach Seiten aus dem *Backend* durch die *RestAPI* geliefert. Hier können verschiedenen Parameter wie Namen, Code oder Preisstufe gesetzt werden. Ein anderer http-Request wird jeweils an das *Backend* geschickt und diese Liste wird dadurch auch gefiltert. In der Mobilversion gibt es die Möglichkeit, durch das Mülltonne-Icon ein Medikament zu entfernen.

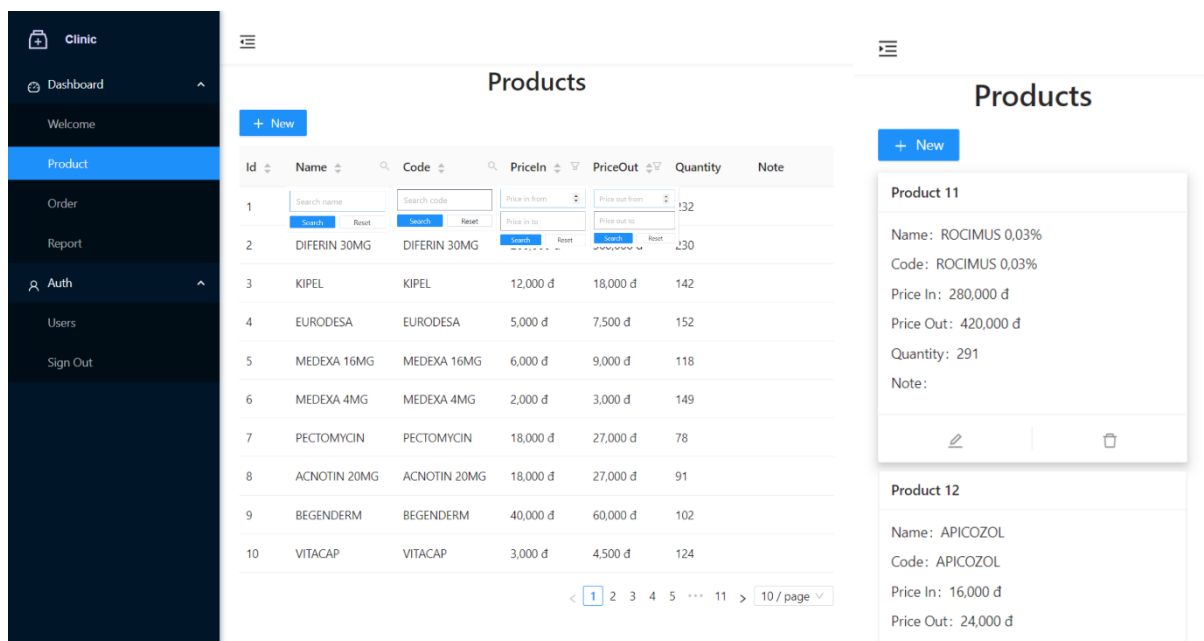


Abbildung 18: Products Page

Wird es auf einer Zeile der Tabelle in der Desktopversion oder auf eine Karte in der Mobileversion geklickt, wird auf die *Detail*-Seite eines Medikamentes wie in Abbildung 19 weitergeleitet. Da die *Create*- und *Edit*-Seite auch diese Form mit allen benötigten Informationen eines Medikamentes enthält, sehen die drei Seiten ähnlich aus. Die Submit-, Detail- und Delete-Buttons sind optional und abhängig von der Aktion (*Create*, *Edit* oder *Detail*). Nachdem die Input Felder in *Frontend* validiert, werden sie nochmal in *Backend* nach Gültigkeit geprüft. Fehlerhinweise können dabei auf dem Bildschirm angezeigt werden.

The screenshot displays the 'Product Create-Edit Page' for a mobile application. On the left is a dark sidebar with navigation options: Clinic, Dashboard, Welcome, Product (highlighted), Order, Report, Auth, Users, and Sign Out. The main content area is split into two views: desktop (left) and mobile (right).

Desktop View: Features a form with the following fields and validation messages:

- Product Name:** Validation: "Please input product name!"
- Product Code:** Validation: "Please input product code!"
- Product Price In:** Validation: "Please input product price in!"
- Product Price Out:** Validation: "Please input product price out!"
- Note:** (Text area)
- Initial Quantity:** Validation: "Please input initial quantity!"

Buttons include "< Back" and a blue "Submit" button.

Mobile View: Shows a simplified form with the following fields:

- Product Name:** Pre-filled with "MENOPAUSE"
- Product Code:** Pre-filled with "MENOPAUSE"
- Product Price In:** Pre-filled with "600000"
- Product Price Out:** Pre-filled with "900000"
- Note:** (Text area)
- Initial Quantity:** Pre-filled with "232"

Buttons include "< Back", "Detail", "Delete", and a blue "Submit" button.

Abbildung 19: Product Create-Edit Page

Als Nächstes ist die Seite einer Bestellung wie in Abbildung 20 dargestellt. Hier werden hauptsächlich die Medikamente einer Bestellung mit Namen, Anzahl, jeweiligen Preisen und den totalen Preisen jedes Medikamentes angezeigt. Die Struktur ist ähnlich wie die Seite der Medikamente mit einer Desktopversion in Tabellenform und einer Mobileversion in Karten.

The screenshot displays the 'Order Detail Page' for a mobile application. On the left is a dark sidebar with navigation options: Clinic, Dashboard, Welcome, Product, Order (highlighted), Report, Auth, Users, and Sign Out. The main content area is split into two views: desktop (left) and mobile (right).

Desktop View: Shows a table titled "Order 2" with the following data:

Id	Date	Product	Quantity	Price	Total
1	May 5, 2021	KEM CERADAN	4	240,000 đ	960,000 đ
2	May 5, 2021	KEM NEOTON	4	800,000 đ	3,200,000 đ
3	May 5, 2021	LACTOPIC CREAM	2	380,000 đ	760,000 đ
4	May 5, 2021	BEPROSALIC XIT	4	130,000 đ	520,000 đ
5	May 5, 2021	KEM BB ISIS	1	800,000 đ	800,000 đ
6	May 5, 2021	AVENCARE	4	500,000 đ	2,000,000 đ
7	May 5, 2021	REGEN CEUTIC	3	1,800,000 đ	5,400,000 đ
8	May 5, 2021	ROCIMUS 0.1%	4	350,000 đ	1,400,000 đ
9	May 5, 2021	PVD SÁT KHUÂN	1	10,000 đ	10,000 đ
10	May 5, 2021	CN SUNCEUTIC 50	4	1,100,000 đ	4,400,000 đ

Buttons include "< Back", "+ New", and a red download icon. A pagination bar at the bottom shows "< 1 2 > 10 / page".

Mobile View: Shows two detail cards for "Order 2":

- Order Detail 1:**
 - Date: May 5, 2021
 - Product Name: KEM CERADAN
 - Quantity: 4
 - Total Price Per Product: 960,000 đ
- Order Detail 2:**
 - Date: May 5, 2021
 - Product Name: KEM NEOTON
 - Quantity: 4
 - Total Price Per Product: 3,200,000 đ

Buttons include "< Back", "+ New", and a red download icon.

Abbildung 20: Order Detail Page

Implementierung & Testing

Zusätzlich wird hier ein roter PDF-Button angeboten, um die Rechnung der Bestellung als PDF herunterzuladen. Wird der Button geklickt, wird zuerst ein Laden-Indikator angezeigt, während die PDF-Datei im Hintergrund im *Backend* erstellt wird. Die Form einer Rechnung sieht wie in Abbildung 21 aus. Neben den Informationen wie in Abbildung 20, enthält eine Rechnung auch Informationen der Klinik, Informationen der Kunden, ob es um Lieferanten oder Endkunden geht und wie hoch die gesamte Rechnung ist.

Your company
Street address
City, street, ZIP code
Phone number, web address, ecc.

Date: Wednesday 28
Invoice INV0001

Bill to:
ABC Company
Company Address
Company state1
Company state 2

Ship to:
Ship name
Ship Address
Ship state1
Ship state 2

BUY: Order 2

No.	Product name	Code	Quantity	Price	Per product
1	KEM CERADAN	KEM CERADAN	4	≈240,000	≈960,000
2	KEM NEOTON	KEM NEOTON	4	≈800,000	≈3,200,000
3	LACTOPIC CREAM	LACTOPIC CREAM	2	≈380,000	≈760,000
4	BEPROSALIC XIT	BEPROSALIC XIT	4	≈130,000	≈520,000
5	KEM BB ISIS	KEM BB ISIS	1	≈800,000	≈800,000
6	AVENCARE	AVENCARE	4	≈500,000	≈2,000,000
7	REGEN CEUTIC	REGEN CEUTIC	3	≈1,800,000	≈5,400,000
8	ROCIMUS 0.1%	ROCIMUS 0.1%	4	≈350,000	≈1,400,000
9	PVD SÁT KHUN	PVD SÁT KHUN	1	≈10,000	≈10,000
10	CN SUNCEUTIC 50	CN SUNCEUTIC 50	4	≈1,100,000	≈4,400,000
11	VASELIN	VASELIN	2	≈20,000	≈40,000
12	BARABIT	BARABIT	3	≈75,000	≈225,000
13	VITISKIN	VITISKIN	5	≈680,000	≈3,400,000
14	DIPALEN	DIPALEN	4	≈130,000	≈520,000
15	PECTOMYCIN	PECTOMYCIN	10	≈18,000	≈180,000

Your notes here

Tota ≈23,815,0

Abbildung 21: Bill Template

Wird auf der Bestellungsünsche-Seite auf eine Zeile der Tabelle in der Desktopversion oder auf eine Karte in der Mobileversion geklickt, wird auf die *Detail*-Seite eines Bestellungsünschs wie in Abbildung 22 weitergeleitet. Diese Form ist auch in *Create*- und *Edit*-Seite eines Bestellungsünschs enthalten. Hier kann ein neues Medikament in eine Bestellung hinzugefügt oder ein Bestellungswunsch für eine andere Anzahl oder ein anderes Medikament modifiziert werden. Wird etwas auf dem Namenfeld eingegeben, werden zusätzliche http-

Requests an das *Backend* geschickt und die Medikamente nach diesem Suchschlüssel gefiltert. Das Ergebnis wird an das *Frontend* zurückgegeben und in einer Dropdown-Liste angezeigt.

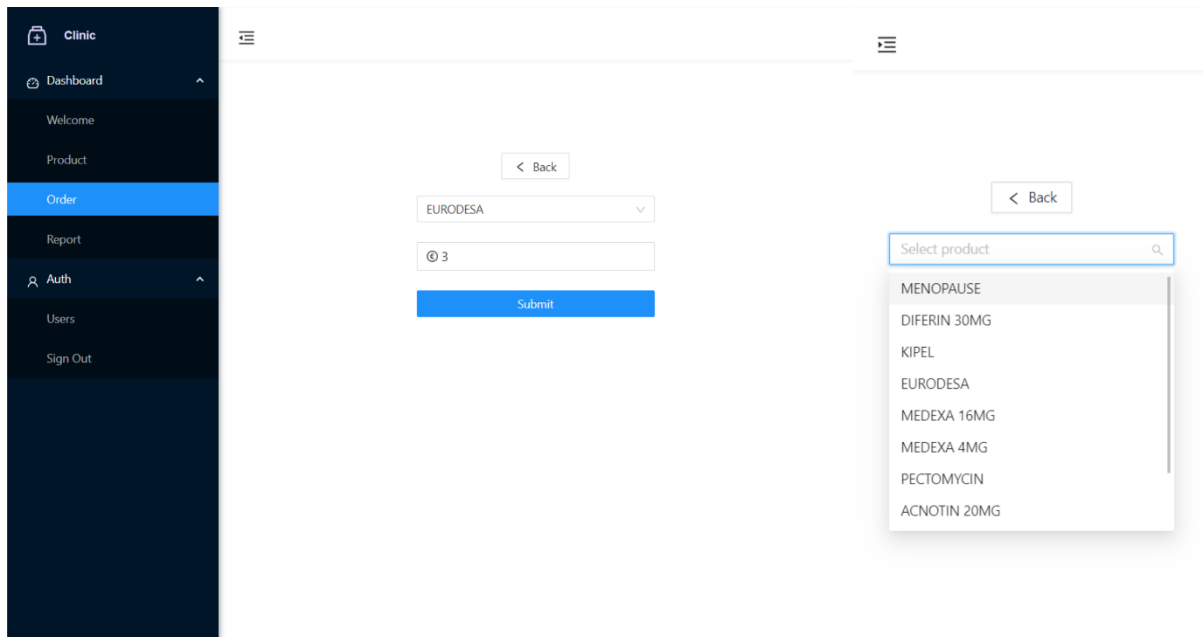


Abbildung 22: Order Detail Create-Edit Page

5.2. Eingesetzte Technologien

Wie im vierten Kapitel schon vorgestellt, werden die Haupttechnologien *Angular*, *Spring Framework*, *PostgreSQL* und *Digital Ocean* verwendet. Einige Vorteile davon sind Codemodularisierung, Erweiterbarkeit und einfaches *Deployment*-Verfahren. Ein rollenbasierendes Authentifikationsverfahren kann im *Frontend* durch *Route Guard*²³ von *Angular* und *Spring Security*²⁴ von *Spring Framework* realisiert werden. Im Prinzip sind alle Technologien quelloffen, außer *Digital Ocean*. Für ein kleines Projekt wie dieses bietet Digital Ocean einen Tarif für einen virtuellen Rechner mit 1GB RAM und 25GB SSD Speicherplatz für 5 Euro/ Monat an²⁵. Die Software kann aber mit Hilfe von Docker durch Multicontainer in irgendeinen anderen Rechner bereitgestellt werden, deshalb sind die anderen gewählten Technologien zur kosten-

²³ <https://angular.io/guide/router>

²⁴ <https://spring.io/projects/spring-security>

²⁵ <https://www.digitalocean.com/pricing>

freien gewerblichen Nutzung erhältlich. Ein einfaches *Deployment*-Verfahren hilft einer Web-basierend-Applikation dabei, weniger *Downtime* und immer die aktuelle Version für alle *Clients* vom *Server* anzubieten.

Der komplette Quellcode wird separat für diese Arbeit in einer CD oder unter github.com/nghiemphan93/clinic zur Verfügung gestellt. Die Endversion des Quellcodes wird gebaut und die Software wird auf einen virtuellen Rechner auf *Digital Ocean* bereitgestellt.

5.3. Testverfahren und Report

Es sind UI-Tests und Integration-Tests für diese Phase der Softwareentwicklung vorzubereiten. Im Rahmen dieser Arbeit können die Unit-Testfälle auf Grund von zeitlichem Umfang nicht implementiert werden.

UI-Tests sind die Prüfung für das *Frontend* in *Angular*, welche manuell ausgeführt wird. Das sichert, dass die UI-Komponenten vom Frontend verhalten, wie der Benutzer erwartet. Responsezeit ist angemessen. In der Wartezeit werden Ladenindikatoren für bessere User-Experience angezeigt. Im Fehlerfall werden Pop-Ups oder Notifikationen auf dem Bildschirm verwendet. Der Benutzer wird benachrichtigt, was schief gegangen ist.

Unit-Tests können erfolgreich laufen aber die ganze Software kann noch nicht richtig funktionieren. Es ist auch gleichzeitig wichtig zu testen, wie die Hauptkomponenten in Integration miteinander oder mit externen Komponenten funktionieren²⁶. Daher werden Integration-Tests für die *Rest-API* in *Spring Boot* eingestellt, um sicher zu stellen, dass das ganze *Backend* wie erwartet funktionieren wird. *Rest-API-Testing* fokussiert sich auf *Request-Body-Validation*, *Response-Code*, *Response-Body* oder *Response-Exceptions*. In diesem Projekt werden Tests für *Requests-Body-Validation* und *Response-Body* organisiert.

Das Ergebnis der Rest-API-Tests wird in Abbildung 23: Ergebnis der Rest-API Integration Tests Abbildung 23 angezeigt. Für die Testfälle wird eine H2 Datenbank verwendet, welche im *RAM* läuft. Die *Controller*-Komponenten bzw. die *CRUD*-Operationen sind hier zu testen.

²⁶ (Khorikow, 2020) Seite 183

Implementierung & Testing

Die *Controller*-Komponenten vom *Backend* wie *BillController*, *ProductController*, *OrderDetailController*, *OrderController*, *ReportController* und *InventoryController* konnten in den Tests erwarteten Daten in *Response-Body* zurückliefern, wenn *Request-Body* mit richtigen Daten aus dem *Frontend* mitgeschickt wurde. Responsezeit war auch angemessen. Test-Code wird zusammen mit dem Quellcode auf der CD oder unter dem GitHub-Link oben zur Verfügung gestellt.












































▼  <default package> 1 sec 608 ms		
▶  BillControllerTest 853 ms		
▶  ProductControllerTest 223 ms		
▶  OrderDetailControllerTest 187 ms		
▶  OrderControllerTest 159 ms		
▶  ReportControllerTest 94 ms		
▶  InventoryControllerTest 92 ms		
▼  BillControllerTest 853 ms	▼  ProductControllerTest 223 ms	▼  OrderControllerTest 159 ms
 shouldCreate() 533 ms	 shouldUpdate() 74 ms	 shouldCreate() 30 ms
 shouldGetAll() 126 ms	 shouldGetAll() 47 ms	 shouldGetAll() 30 ms
 shouldDelete() 85 ms	 shouldCreate() 43 ms	 shouldUpdate() 29 ms
 shouldUpdate() 56 ms	 shouldGetOne() 31 ms	 shouldDelete() 28 ms
 shouldGetOne() 53 ms	 shouldDelete() 28 ms	 shouldGetOne() 21 ms
		 shouldGetOnePdf() 21 ms
▼  InventoryControllerTest 92 ms	▼  ReportControllerTest 94 ms	▼  OrderDetailControllerTest 187 ms
 shouldUpdate() 31 ms	 shouldGetAll() 25 ms	 shouldDelete() 43 ms
 shouldCreate() 24 ms	 shouldGetOne() 19 ms	 shouldGetAll() 43 ms
 shouldGetOne() 19 ms	 shouldUpdate() 19 ms	 shouldCreate() 40 ms
 shouldDelete() 18 ms	 shouldCreate() 17 ms	 shouldUpdate() 36 ms
	 shouldDelete() 14 ms	 shouldGetOne() 25 ms

Abbildung 23: Ergebnis der Rest-API Integration Tests

6. Evaluation

In diesem Kapitel werden die Ergebnisse der Arbeit bewertet, nämlich die Umsetzung, die Architekturen und das Entwicklungsverfahren der Software, bevor die Arbeit mit einem Fazit und Ausblick zusammengefasst wird.

Zu der Umsetzung kann auf die Webseite der Software durch einen Webbrowser von Desktop, iOS- oder Android-Geräten zugegriffen werden. Die Software kann lokal oder online getestet werden. Um den Code lokal bauen zu können, muss Folgendes durchgeführt werden:

1. Die Software *Docker* auf dem eigenen Rechner installieren
2. Den Quellcode aus dem GitHub-Link herunterladen oder aus der CD kopieren
3. Unter dem Root-Verzeichnis die *Docker-Compose*-Datei für die Entwicklungsumgebung mit dem Befehl „*docker compose up*“ ausführen

Danach werden drei separate *Docker-Containers* für *Frontend*, *Backend* und Datenbank erstellt und verbunden. Das *Frontend* ist dann erreichbar unter *localhost:4200*. Obwohl wie im Abschnitt Testverfahren und Report im letzten Kapitel beschrieben keine Unit-Tests durchgeführt wurden, wird die Korrektheit der Kommunikation zwischen *Frontend* und *Backend* immer noch gesichert dank der *Integration-Tests* für die *RestAPI*. Die implementierten Funktionalitäten können nun durch die UI-Komponenten getestet werden. Die Online-Version der Software ist zugreifbar unter <http://159.89.23.178>. Antwortzeit aus dem virtuellen Rechner von *Digital Ocean* war angemessen.

Abhängig von der Plattform wird die Software in einem responsiven Layout, wie in Mock Up GUIs beschrieben, angezeigt. Für fünf bis zehn Benutzer kann die Software ohne Probleme verwendet werden. Das Teilsystem für den Medikamentenhandel wurde wie angefordert zunächst implementiert. Mit der bestehenden Software kann der Benutzer sich anmelden, Medikamente, andere Benutzer, Bestellungen und Bestellungswünsche nach Konzeption anlegen, modifizieren und löschen, vorausgesetzt, dass der Benutzer die geeignete Rolle besitzt. Geschützte Routen in *Frontend* und *Backend* können nur durch authentifizierte Benutzer mit geeigneten Rollen zugegriffen werden. Suche- und Filterfunktionalitäten wurden auch realisiert. Zu den Nachteilen zählen keine automatisierten Unit-Testfälle. Bisher kann der Code

immer noch manuell aus dem Master Branch von GitHub auf den virtuellen Rechner von *Digital Ocean* in Multicontainer gepullt und ausgeführt werden. Ein automatisiertes *Deployment*-Verfahren von *GitHub* auf *DigitalOcean* wird auch erwünscht.

Die gewählte Architektur dieser Software war eine Variante von *Layered Architecture* und *Microservices Architecture*. Zu dieser Architektur hat es dabei geholfen, den Code in Modulen bzw. Komponenten und Ebenen zu organisieren. Die Entwicklungserfahrung mit der Architektur in *Frontend* mit *Angular* und in *Backend* mit *Spring Boot* war angenehm. Da *Angular* und *Spring Boot* die Funktionalität *Dependency Injection* bietet²⁷, können Module in den anderen Modulen injiziert werden, ohne Instanzen davon jedes Mal zu erstellen und verwenden. Ein anderer Vorteil von dieser Architektur ist, neue Funktionalitäten als weitere Module in das bestehende System einzubinden. Hier in diesem Fall wurde das Patientensystem noch nicht implementiert, welches später mit dieser Architektur als ein neues Modul entwickelt und verbunden wird.

Das Entwicklungsverfahren hat auch geholfen, Schritt für Schritt die Software aus den Anforderungen der Kunden bis zum Endprodukt zu entwickeln. Dieses Verfahren dauert aber lang, bis die Benutzer eine Demo sehen können. Unter anderem hat es auch gezeigt, dass die Fern-Kommunikation in der Zusammenarbeit mit Kunden nicht effektiv war. Terminvereinbarung war wegen der sechsstündiger Zeitzonendifferenz zwischen Deutschland und Vietnam fast unmöglich. Übrigens soll die Demo in der Zukunft so schnell wie möglich implementiert werden. Danach kann es weiter durch Feedback der Nutzer aus der Demo verbessert werden, da die Benutzer nicht immer vom Anfang wissen, was sie wollen. In diesem Fall, nachdem die Benutzer die bestehende Software verwenden, wird Folgendes gewünscht:

- Die Kunden als eine separate Tabelle mit Beziehung auf die Bestellungstabelle erstellen
- Kaufhistorie der Kunden analysieren
- Erstellung der Bestellungswünsche auf nur einer Seite für schnelle Bearbeitungszeit

²⁷ (Carnell, 2017) Seite 5, (Wilken, 2018) Seite 66

7. Fazit und Ausblick

Im Rahmen dieser Arbeit wurde die konzeptionelle Analyse der Software für das Patienten- und Medikamentenverwaltungssystem durch UML Diagramme erstellt. Insbesondere wurden die Architekturen davon durch die Haupttechnologien *Angular*, *Spring Boot* in eine ausführbare Software umgesetzt und in einen virtuellen Rechner auf *Digital Ocean* bereitgestellt. Dazu gehört die Erreichbarkeit auf die Software von iOS, Android und Desktop Web durch nur ein responsives Layout.

Zuerst wurden die Anforderungen der Stakeholder in Form von kleinen Texten beschrieben und analysiert. Danach wurden ein Anwendungsfalldiagramm, seine Beschreibungstabellen und Aktivitätsdiagramme als Werkzeuge genutzt, um diese Anforderungen der Zielbenutzer zu erheben.

Nachdem die Anforderungen analysiert worden sind, wurde angefangen, die Datenbankmodelle zu gestalten, um festzustellen, welche Daten im System zu speichern sind. Als Hauptwerkzeug wurde ein *Entity-Relationship-Diagramm* verwendet, um das Datenmodell zu modellieren. Darüber hinaus wurde ein Datenklassendiagramm vorgestellt, wobei dieses fast eins zu eins ähnlich zu dem *Entity Relationship-Diagramm* war. Das spielt die Rolle als eine Schnittstelle zwischen der Datenbank und der objektorientierten Programmierwelt.

Im nächsten Abschnitt wurden verschiedene statische Modelle mit Hilfe von Komponentendiagrammen erstellt, um die wichtigsten Bausteine der Software zu strukturieren, nämlich das *Frontend* und *Backend*. Dazu wird die *RestAPI* in Tabellenform beschrieben, welche das Kommunikationsverhalten zwischen des Frontends und Backends feststellt. Danach wird die Architektur der Softwarebereitstellung beschrieben. Es werden hier die Entwicklungsumgebung bzw. Produktionsumgebung mit Hilfe von zwei Verteilungsdiagrammen analysiert.

Das nächste Kapitel stellt die Haupttechnologien in diesem Projekt vor. Das sind *Angular*, *Spring Boot*, *PostgreSQL*, *Docker* und *Digital Ocean* auszuwählen. Es werden die Vorteile jeder Technologie und ihr Preismodell vorgestellt.

Das nächste Kapitel über Implementierung fängt mit den Mock-Up-GUIs an. Hier werden plattformabhängige UI-Komponenten für jede Sicht erstellt, um die Benutzeroberfläche der Software zu visualisieren. Bis zu diesem Zeitpunkt können die Konzepte bzw. Architekturen

der Software durch die gewählten Technologien realisiert werden. Das Kapitel schließt mit der Beschreibung und dem Ergebnis des Testverfahrens der Umsetzung ab. Hier sind UI- und Integrations-Tests durchzuführen.

Obwohl im nächsten Kapitel über Evaluation der Umsetzung, der Architektur und des Entwicklungsverfahrens festgestellt wurde, dass es noch Stellen gibt, die verbessert werden können, hat die Umsetzung fast alle Anforderungen erfüllt und stellt das Entwicklungsverfahren eine generische Lösung für weitere Funktionalitäten.

Diese Arbeit hat aufgezeigt, wie Anforderungen eines echten Businessanwendungsfalls mit methodischen Werkzeugen analysiert wurden, wie Softwarearchitekturen mit UML Diagrammen modelliert wurden und wie das konzeptionelle Design mit den neuen Technologien umgesetzt wurde. Die Erfahrungen mit dieser methodischen Entwicklungsanalyse und den gewählten Technologien in dieser Arbeit können dabei helfen, nicht nur weitere Funktionalitäten für dieses Projekt nach Bedarf zu implementieren, sondern auch andere Projekte in der Zukunft mit der gelernten Methodik zu verwirklichen.

Literaturverzeichnis

- Booch, Grady; Rumbaugh, James; Jacobson, Ivar;. (2005). *Unified Modeling Language User Guide, The, 2nd Edition*. New Jersey: Addison-Wesley Professional.
- Braude, E., & Bernstein, M. (2011). *Software Engineering Modern Approaches* (2nd Ausg.). Long Grove: Waveland Press, Inc.
- Bruegge, B., & Dutoit, A. (2010). *Object-Oriented Software Engineering Using UML, Patterns, and Java™* (3rd Ausg.). New Jersey: Pearson Education, Inc.
- Carnell, J. (2017). *Spring Microservices in Action*. Shelter Island: Manning Publications Co.
- Daniels, P., & Atencio, L. (2017). *RxJS in Action* (1st Ausg.). New York: Manning Publications Co.
- DigitalOcean, L. (1. August 2021). *DigitalOcean*. Von DigitalOcean: <https://www.digitalocean.com/> abgerufen
- Docker, I. (1. August 2021). *Docker*. Von Docker: <https://www.docker.com/> abgerufen
- Gomaa, H. (2011). *Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures*. New York: Cambridge University Press.
- Google. (1. August 2021). *Angular*. Von Angular: <https://angular.io/> abgerufen
- Group, T. P. (1. August 2021). *PostgreSQL*. Von PostgreSQL: <https://www.postgresql.org/> abgerufen
- Khorikow, V. (2020). *Unit Testing: Principles, Practices and Patterns*. Shelter Island: Manning Publications Co.
- Koelsch, G. (2016). *Requirements Writing for System Engineering*. Herndon: Apress.
- Kongress. (25. 08 2021). *Bibliothek der Gesetze* . Von Thu Vien Phap Luat: <https://thuvienphapluat.vn/van-ban/The-thao-Y-te/Luat-kham-benh-chua-benh-nam-2009-98714.aspx> abgerufen
- Laplante, P. A. (2018). *Requirements Engineering for Software and Systems*. Boca Raton: CRC Press.

-
- Lauret, A. (2019). *The Design of Web APIs* (1st Ausg.). New York: Manning Publications Co.
- Madden, N. (2020). *API Security in Action*. Shelter Island: Manning Publications Co.
- Martin, R. (2018). *Clean Architecture: A craftsman's guide to software structure and design* (1st Ausg.). Pearson Education, Inc.
- Palmer, J., Cohn, C., Giambalvo, M., & Nishina, C. (2018). *Testing Angular Applications* (1st Ausg.). New York: Manning Publications Co.
- Paradigm, V. (2019). *Visual Paradigm*. Abgerufen am 1. November 2019 von <https://circle.visual-paradigm.com/docs/>
- Pohl, K., & Rupp, C. (2015). *Requirements Engineering Fundamentals*. Santa Barbara: rockynook.
- Richards, M. (2015). *Software Architecture Patterns*. Sebastopol: O'Reilly.
- Robert, M. C. (2018). *Clean Architecture*. USA: Prentice Hall.
- Rozanski, N., & Woods, E. (2005). *Software Systems architecture*. Upper Saddle River: Addison-Wesley.
- Sommerville, I. (2016). *Software Engineering* (10th Ausg.). Harlow: Pearson Education Limited.
- Van Deusen, S., & Seemann, M. (2019). *Dependency Injection: Principles, Practices and Patterns* (1st Ausg.). New York: Manning Publications Co.
- VMware, I. (1. August 2021). *Spring*. Von Spring: <https://spring.io/> abgerufen
- Weisfeld, M. (2019). *The Object-Oriented Thought Process* (5th Ausg.). Pearson Education, Inc.
- Wiegers, K., & Beatty, J. (2013). *Software Requirements*. Redmond: Microsoft Press.
- Wilken, J. (2018). *Angular in Action* (1st Ausg.). New York: Manning Publications Co.

Appendix

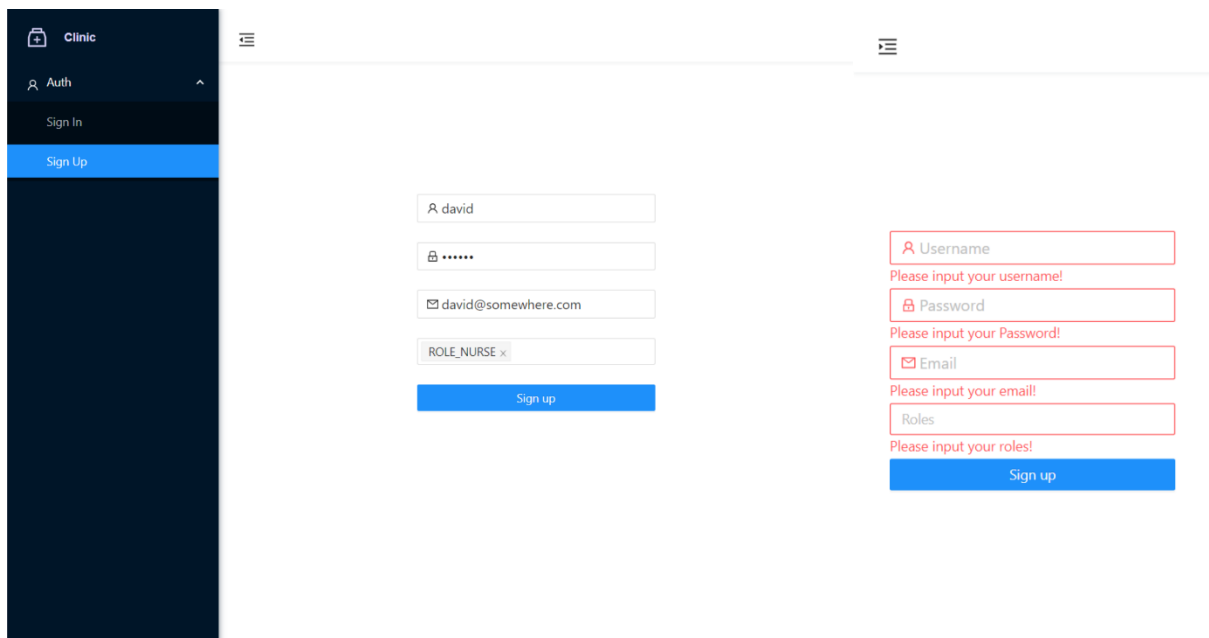
Endpunkte	http	Rückgabewert	Rollen
/auth/login	POST	User mit JWT-Token	Public
/auth/logout	POST		Public
/auth/register	POST		Public
/api/users/{userId}	PUT	User	Manager
/api/users/{userId}	DELETE		Manager
/api/patients	GET	List<Patient>	Alle
/api/patients/{patientId}	GET	Patient	Alle
/api/patients	POST	Patient	Alle
/api/patients/{patientId}	PUT	Patient	Alle
/api/patients/{patientId}	DELETE		Alle
/api/appointments	GET	List<Appointment>	Alle
/api/appointments/ {appointmentId}	GET	Appointment	Alle
/api/appointments	POST	Appointment	Alle
/api/appointments/ {appointmentId}	PUT	Appointment	Alle
/api/appointments/ {appointmentId}	DELETE		Alle
/api/visits	GET	List<Visit>	Manager, Doctor
/api/visits/{visitId}	GET	Visit	Manager, Doctor
/api/visits	POST	Visit	Manager, Doctor
/api/visits/{visitId}	PUT	Visit	Manager, Doctor

/api/visits/{visitId}	DELETE		Manager, Doctor
/api/visits/{visitId}/ prescriptions	GET	List<Prescription>	Manager, Doctor
/api/visits/{visitId}/ prescriptions/{prescriptionId}	GET	Prescription	Manager, Doctor
/api/visits/{visitId}/ prescriptions	POST	Prescription	Manager, Doctor
/api/visits/{visitId}/ prescriptions/{prescriptionId}	PUT	Prescription	Manager, Doctor
/api/visits/{visitId}/ prescriptions/{prescriptionId}	DELETE		Manager, Doctor
/api/products	GET	List<Product>	Alle
/api/products/{productId}	GET	Product	Alle
/api/products	POST	Product	Alle
/api/products/{productId}	PUT	Product	Alle
/api/products/{productId}	DELETE		Alle
/api/orders	GET	List<Order>	Alle
/api/orders/{orderId}	GET	Order	Alle
/api/orders	POST	Order	Alle
/api/orders/{orderId}	PUT	Order	Alle
/api/orders/{orderId}	DELETE		Alle
/api/orders/{orderId}/ orderDetails	GET	List<OrderDetail>	Alle
/api/orders/{orderId}/	GET	OrderDetail	Alle

orderDetails/{orderId}			
/api/orders/{orderId}/ orderDetails	POST	OrderDetail	Alle
/api/orders/{orderId}/ orderDetails/{orderId}	PUT	OrderDetail	Alle
/api/orders/{orderId}/ orderDetails/{orderId}	DELETE		Alle
/api/bills	GET	List<Bill>	Alle
/api/bills/{billId}	GET	Bill	Alle
/api/bills/{billId}/pdf	GET	Bill als Pdf-Datei	Alle
/api/bills	POST	Bill	Alle
/api/bills/{billId}	PUT	Bill	Manager
/api/bills/{billId}	DELETE		Manager
/api/reports	GET	List<Report>	Manager
/api/reports/{reportId}	GET	Report	Manager
/api/reports/{reportId}/pdf	GET	Report als Pdf-Datei	Manager
/api/reports	POST	Report	Manager
/api/reports/{reportId}	PUT	Report	Manager
/api/reports/{reportId}	DELETE		Manager
/api/userActions	GET	List<UserAction>	Manager
/api/userActions/ {userId}	GET	UserAction	Manager
/api/userActions	GET	UserAction	Manager
/api/userActions/	POST	UserAction	Manager

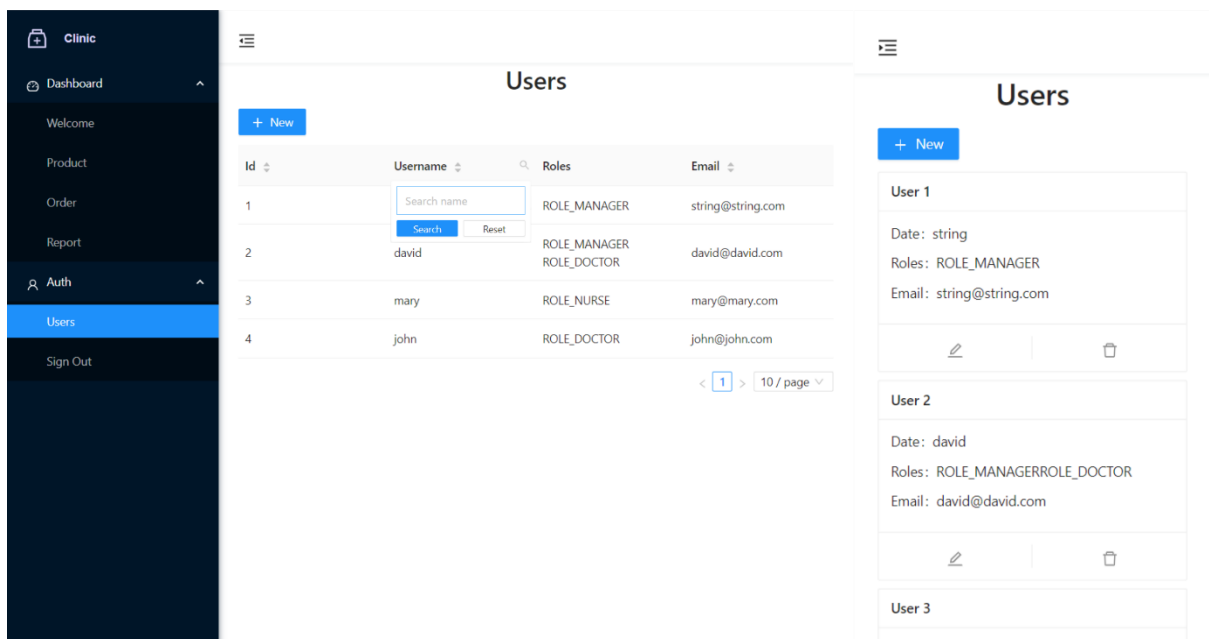
{userActionId}			
/api/userActions/ {userActionId}	<i>PUT</i>	<i>UserAction</i>	<i>Manager</i>
/api/userActions/ {userActionId}	<i>DELETE</i>		<i>Manager</i>

Tabelle 5: Beschreibung der-RestAPI zwischen Backend und Frontend



The image shows a 'Sign Up' page for a 'Clinic' application. On the left is a dark sidebar with a 'Clinic' header and a menu containing 'Auth', 'Sign In', and 'Sign Up' (which is highlighted in blue). The main content area is white and contains two forms. The left form is a 'Sign Up' form with fields for 'Name' (containing 'david'), 'Password' (masked with dots), 'Email' (containing 'david@somewhere.com'), and 'Role' (a dropdown menu with 'ROLE_NURSE' selected). A blue 'Sign up' button is at the bottom. The right form is a 'Sign Up' form with fields for 'Username', 'Password', 'Email', and 'Roles'. Each field has a red border and a red error message below it: 'Please input your username!', 'Please input your Password!', 'Please input your email!', and 'Please input your roles!'. A blue 'Sign up' button is at the bottom.

Abbildung 24: Sign Up Page



The image shows a 'Users' page for a 'Clinic' application. On the left is a dark sidebar with a 'Clinic' header and a menu containing 'Dashboard', 'Welcome', 'Product', 'Order', 'Report', 'Auth', 'Users' (which is highlighted in blue), and 'Sign Out'. The main content area is white and contains two views of the 'Users' table. The left view is a table with columns 'Id', 'Username', 'Roles', and 'Email'. It has a '+ New' button and a search bar. The right view is a 'Users' page with a '+ New' button and a list of users. Each user entry shows their name, roles, and email, along with edit and delete icons.

Id	Username	Roles	Email
1	Search name	ROLE_MANAGER	string@string.com
2	david	ROLE_MANAGER ROLE_DOCTOR	david@david.com
3	mary	ROLE_NURSE	mary@mary.com
4	john	ROLE_DOCTOR	john@john.com

< 1 > 10 / page

Users

+ New

User 1

Date: string
Roles: ROLE_MANAGER
Email: string@string.com

User 2

Date: david
Roles: ROLE_MANAGERROLE_DOCTOR
Email: david@david.com

User 3

Abbildung 25: Users Page

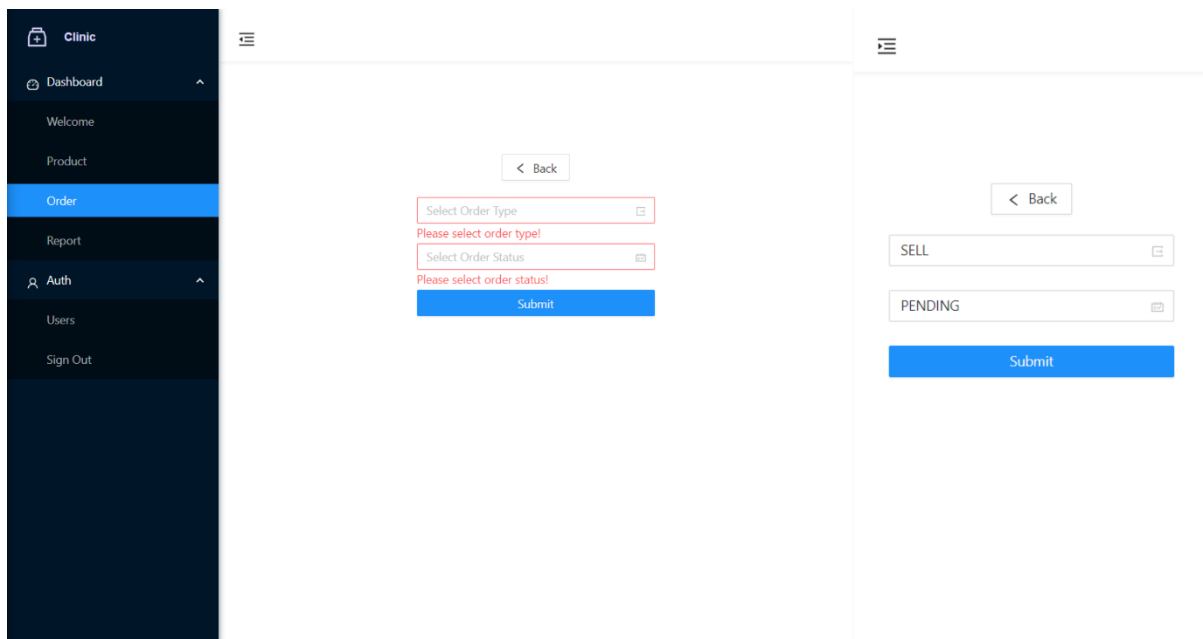


Abbildung 26: Order Create-Edit-Detail Page

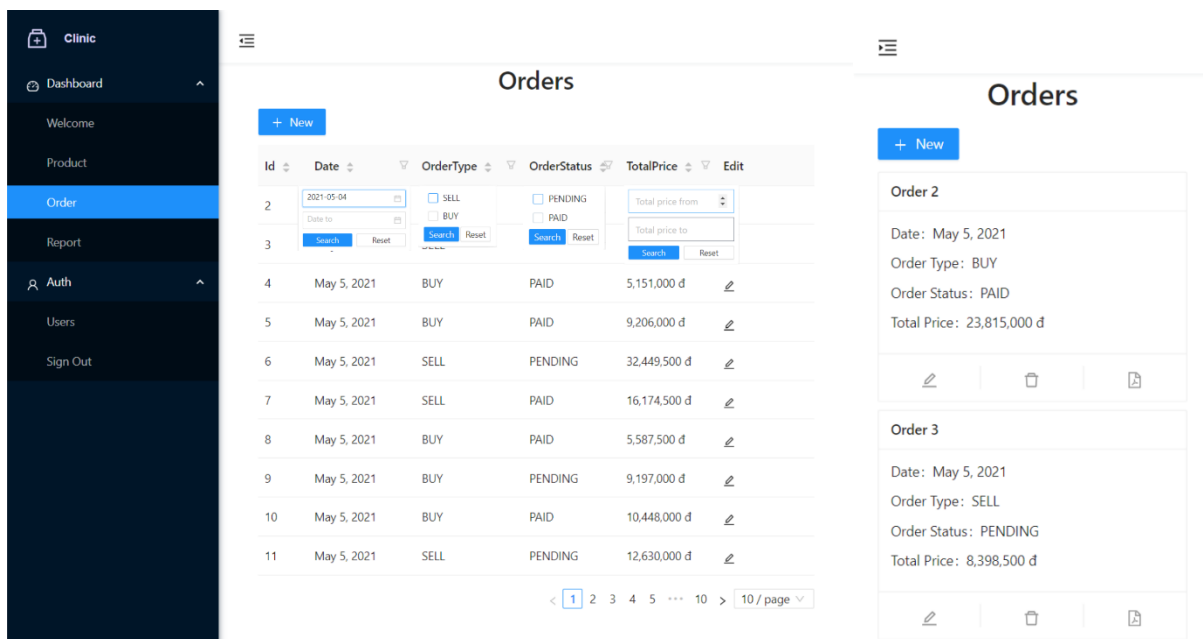


Abbildung 27: Orders Page

Eidesstattliche Erklärung

„Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht.“

Ort, Datum

Unterschrift