hochschule mannheim

Konzeption und Implementierung der Migration verschiedener Web-Frameworks am Beispiel der CHILI/Telemedizinakte und PrimeFaces

Julia Eileen Jungmann

Bachelor-Thesis

zur Erlangung des akademischen Grades Bachelor of Science (B.Sc.)

Studiengang Informatik

Fakultät für Informatik

Hochschule Mannheim

29.11.2016

Betreuer

Prof. Lutz Strüngmann, Hochschule Mannheim

Dipl.- Inform. (FH) Florian Schwind, CHILI GmbH

Jungmann, Julia Eileen:
Konzeption und Implementierung der Migration verschiedener Web-Frameworks am Beispiel der CHILI/Telemedizinakte und PrimeFaces / Julia Eileen Jungmann. –
Bachelor-Thesis, Mannheim: Hochschule Mannheim, 2016. 52 Seiten.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h., dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Hochschule Mannheim öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Mannheim, 29.11.2016

Julia Eileen Jungmann

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Bachelorarbeit unterstützt und motiviert haben.

Zuerst gebührt mein Dank Herr Prof. Lutz Strüngmann, der meine Bachelorarbeit betreut und mich unterstützt hat.

Ebenfalls möchte ich mich bei der CHILI GmbH und dem Leiter der Entwicklungsabteilung, Dr. Heiko Münch bedanken, die diese Arbeit ermöglicht haben.

Besonderer Dank gebührt meinem Betreuer, Florian Schwind, der mir von der Themenfindung bis hin zur fertigen Arbeit jederzeit unterstützend zur Seite stand und immer ein offenes Ohr hatte.

In diesem Zuge möchte ich mich auch bei meinen Arbeitskollegen Jens Poxleitner und Peter Fritz bedanken, die mir mit viel Geduld und Hilfsbereitschaft zur Seite standen. Bedanken möchte ich mich für die zahlreichen Anregungen und Ideen, die maßgeblich dazu beigetragen haben, dass diese Bachelorarbeit in dieser Form vorliegt.

Darüber hinaus möchte ich mich für das Korrekturlesen und die Unterstützung bei meiner guten Freundin Karolin Walter, meinem Freund Helge Eggers, und meiner Mutter Carmen Jungmann bedanken.

Abschließend möchte ich mich bei meiner Mutter und meinen Großeltern bedanken, die mir mein Studium durch ihre Unterstützung ermöglicht haben und stets ein offenes Ohr sowie emotionalen Rückhalt geboten haben.

Mannheim, 29.11.2016

Julia Eileen Jungmann

Abstract

Konzeption und Implementierung der Migration verschiedener Web-Frameworks am Beispiel der CHILI/Telemedizinakte und PrimeFaces

Die vorliegende Bachelorarbeit dient als Leitfaden für die Konzeption und Implementierung der Umstellung eines Softwareprojekts auf ausgewählte Web-Frameworks am Beispiel von PrimeFaces und der CHILI/Telemedizinakte.
Hierfür wurde ein Konzept für die Migration der CHILI/Telemedizinakte erarbeitet, ein passendes Framework ausgewählt und anschließend das etappenweise Vorgehen der Umstellung von JavaServer Faces 1.1 zu JavaServer Faces 2.2 mit der Nutzung von PrimeFaces beschrieben. Besonders wird hierbei auf Probleme und Schwierigkeiten eingegangen, die ein solch großer Versionssprung und ein älteres Backend mit sich bringen. Auch die Umstellung eines Projektes auf das Model-View-Controller-Prinzip wird beschrieben.
Am Ende dieser Ausführung steht die neue CHILI/Telemedizinakte, die auf einem solchen Stand ist, dass die grundsätzliche Migration der Basiskomponenten durchgeführt wurde und sie mit der bestehenden Basis und mit Hilfe dieser Arbeit fertiggestellt werden kann.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Dieses Kapitel gibt einen Eindruck darüber, mit welchem Hintergrund diese Arbeit entstanden ist, was die Zielsetzung ist und wie die Auswahl auf dieses Thema fiel. Außerdem wird eine kurze Vorstellung des Unternehmens gegeben, in welchem diese Arbeit entstand.

1.1 Motivation

Erfahrungen mit diversen Web-Applikationen haben gezeigt, dass es gerade im Web-Bereich sehr wichtig ist auf einem technologisch aktuellen Stand zu arbeiten. Die CHILI/Telemedizinakte (AKTE) bot zwar die gewünschten Funktionalitäten, die angewendeten Web-Technologien stießen aber an ihre Grenzen, was Wartbarkeit und Weiterentwicklung betrifft. Die Schwierigkeit, die Kompatibilität mit neuen Browsern und verschiedenen Endgeräten zu erhalten oder zu schaffen, wächst stetig. Auch im Bereich der Sicherheit waren Verbesserungen erwünscht und möglich. So ergab sich während eines Praxissemesters bei der CHILI GmbH das Interesse die AKTE - welche sich zwar kontinuierlich weiterentwickelt, aber auf einer nicht mehr aktuellen, problembehafteten, Web-Technologie basiert - auf aktuelle Technologien zu migrieren und dieses Vorgehen detailliert auszuleuchten und umzusetzen.

1.2 Zielsetzung

Ziel dieser Bachelorarbeit ist es, ein passendes Konzept der Migration der verschiedenen Web-Frameworks am Beispiel der AKTE und PrimeFaces zu erarbeiten. Der Inhalt soll die Migration für zukünftige Anwendungszwecke vereinfachen und eine Anleitung zur Vorgehensweise für eine komplette Implementierung ähnlicher Projekte bieten, indem die Migration und die damit verbundenen Probleme dargelegt werden. Einen zusätzlichen wichtigen Punkt stellt die Umstellung auf ein konsequentes Model-View-Controller (MVC)-Konzept, wie in Abschnitt 2.13 erklärt, dar, wobei die Logik aus dem Frontend komplett ausgelagert und ins Backend verschoben wird. Im Zuge dessen bietet es sich an, Code-Refactoring zu betreiben. Außerdem soll die Sicherheit der Anwendung gesteigert und eine gute Wartbarkeit erreicht werden.
Entstehen soll ein Leitfaden, der die Probleme und Vorgehensweisen zum Umstellen von älteren Web-Frameworks auf eine aktuellere Version von JavaServer Faces (JSF) und PrimeFaces darstellt. Am Ende dieser Arbeit sollte die Implementierung soweit fortgeschritten sein, dass auf deren Grundlage eine Fertigstellung der AKTE ohne Probleme umgesetzt werden kann.

1.3 Unternehmensvorstellung

Die CHILI GmbH entwickelt Softwarekomponenten für die Teleradiologie, Picture Archiving and Communication Systems (PACS), und andere bildverarbeitende Fachgebiete, wie z.B. Kardiologie oder Pathologie und hat ihren Sitz in Dossenheim bei Heidelberg. Die Kunden der CHILI GmbH sind Praxen, kleine, mittlere und große Krankenhäuser und Universitätskliniken. Nicht nur die bilderzeugenden Fachabteilungen, sondern auch deren Zuweiser setzen CHILI-Systeme ein, um Arbeitsabläufe effizienter und kostengünstiger zu gestalten und die Qualität der ärztlichen Versorgung zu verbessern. Die Software basiert auf den Ergebnissen der wissenschaftlichen Forschung, die gemeinsam mit dem Deutschen Krebsforschungszentrum (DKFZ) in Heidelberg betrieben wird. Momentan sind circa 50 Mitarbeiter bei der CHILI GmbH beschäftigt.

Kapitel 2

Grundlagen

Im Grundlagenkapitel werden alle Einzelheiten, welche zum Verständnis der Arbeit benötigt werden, erklärt und zusammengefasst dargestellt. Ein gewisses technisches Verständnis und Vorkenntnisse in der Materie werden dennoch vorausgesetzt, da eine zu detaillierte Erklärung der Technologien den Rahmen dieser Arbeit sprengen würde.

2.1 CHILI/Telemedizinakte

Die AKTE ermöglicht eine patientenzentrierte und webbasierte Ansicht auf Daten des Systems, mit der Möglichkeit diese zu erstellen oder zu bearbeiten. Alle zu einem Patienten gespeicherten Daten werden in der AKTE visualisiert und verwaltet. Sie beinhaltet Funktionen zur Kooperation zwischen verschiedenen Einrichtungen. Unter anderem Konsilerstellung und -verfolgung, ein Kommunikationssystem und ein patientenzentriertes Diskussionsforum.
Die AKTE ist ein rein webbasiertes Modul auf der Basis von JSF und Apache MyFaces. Die Besonderheit der AKTE ist, dass das Backend auf einem hohen Level konfigurierbar ist und unter anderem als firmeninternes Help-Desk-System, als auch als Dienstplan und Telereadiologieportal zur Erstellung einer Rechtfertigenden Indikation (RI) dient. Zudem greift sie auf ein komplexes Rechte- und Rollenmanagement zurück.
Durch diese Eigenschaften muss das Backend dynamisch und mit vielen Plugins kompatibel sein, was die Komplexität der Migration erhöht. Die Schwierigkeit be-

steht unter anderem darin, alle Funktionen zu erhalten und keine Verschlechterung zu erzielen.

2.2 Java

Java ist eine von Sun Microsystems entwickelte objektorientierte Programmiersprache und eingetragene Marke. 2010 wurde Java von Oracle aufgekauft. [1]
In Schlagworten beschreibt Sun Java als einfach, objektorientiert, verteilt, interpretiert, robust, sicher, architekturunabhängig, portabel, hochperformant, multithreadfähig und dynamisch. Java ist einfach, da die Anzahl von Sprachkonstrukten klein gehalten wurde und es einige Konstrukte bietet, die auch C und C++ verwenden. Es ist darauf ausgelegt, von einem Programmierer schnell erlernt zu werden und bietet weniger Möglichkeiten für schlechte Programmiertechniken als zum Beispiel C oder C++. Ebenso ist Java eine verteilte Sprache, da es Netzwerkverbindungen auf verschiedenen Ebenen unterstützt. [2] Das Backend der AKTE ist komplett in Java geschrieben.

2.3 Entwicklungsumgebung

Als Entwicklungsumgebung wird Eclipse verwendet, welches das gewünschte Syntax-Highlighting mitbringt, eine gut durchdachte Versionskontrolle und eine Kommunikationsmöglichkeit mit dem bei CHILI eingesetzten Concurrent Version System (CVS) bietet. Außerdem wird die von CHILI genutzte Open-Source-Java-Library APACHE ANT von Eclipse unterstützt und kann dort integriert werden. So können aus Eclipse heraus mit Hilfe eines Plugins direkt Jar-Dateien erzeugt oder Applikationen gestartet werden.

2.4 Webserver

Die Hauptaufgabe eines Webservers ist es, mit HTTP Daten zu übertragen, siehe Abbildung 2.1. Ein Beispiel für die Funktion eines Webservers ist die Eingabe einer URL durch den Nutzer. Diese URL wird als HTTP-Request an den Webserver

geschickt. Dieser generiert daraus eine Webseite, welche er als HTTP-Response zurück an den Client schickt und die daraufhin angezeigt wird. [3]

2.5 Servlet-Container

Da von einem Webserver nur statische Webseiten zurückgegeben werden können, wird für dynamische Anfragen eine andere Lösung benötigt. Für diesen Zweck gibt es Servlet-Container. Das Grundprinzip eines Servlet-Containers ist es, Java zu benutzen um serverseitig eine dynamische Webseite zu erstellen. Ein Servlet-Container ist also ein Teil eines Webservers, siehe Abbildung 2.1, der mit den Servlets, siehe Abschnitt 2.6, interagiert und ein Behältnis für diese darstellt. [3]

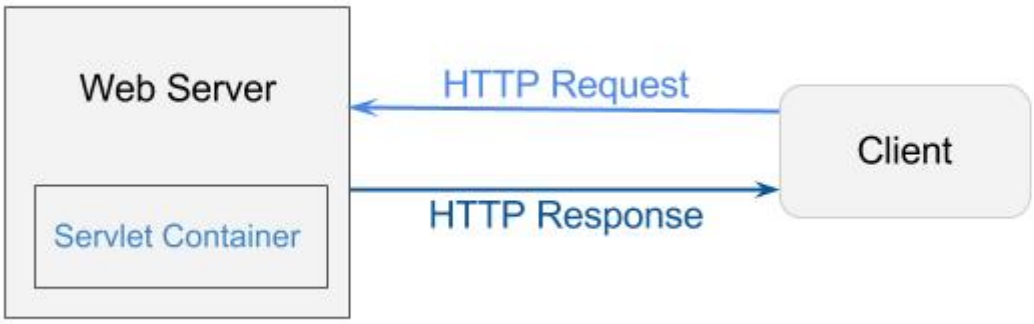

Abbildung 2.1: Zusammenhang von Servlet-Container, Webserver und Client. [3]

2.6 Servlets

Ein Servlet ist eine in Java geschriebene Software, die in einem besonders vorbereiteten Java-Webserver ausgeführt wird und HyperText Markup Language (HTML)-Ergebnisseiten erzeugt. Ein Java-Webserver der Servlets lädt und verwaltet heißt auch Servlet-Container, siehe Abschnitt 2.5. Mit Servlets werden die Fähigkeiten von Webservern insofern erweitert, dass dynamische Informationen generiert werden können. „Servlets sind somit ein wenig mit Applets vergleichbar. Ein Applet ist ein Java-Programm auf der Clientseite (im Browser), während ein Servlet ein Programm auf der Serverseite (im Server) ist. Der Browser ist der Applet-Container, während der Java-Webserver mit Servlet-Schnittstelle einen Servlet-Container dar-

stellt.“ [4]
Hier eine unvollständige Auflistung verschiedener Webserver mit Servlet-Funktionalität: [5]

- **Apache Tomcat.** Ein Open-Source-Produkt der Apache Software Foundation. „Definiert eine Umgebung zur Ausführung von Java-Code auf Webservern, integriert einen Servlet-Container und auch einen HTTP-Server.“
- **Glassfish.** „ Leistungsfähiger Webserver mit Unterstützung für Servlets, CGI-Scripts und PHP, der einen Webcontainer integriert.“
- **Jetty.** „HTTP-Server, der sich u.a. in individuelle Software (Webservices) einbetten lässt.“

2.7 JavaServer Pages

JavaServer Pages (JSP) ist eine Technologie um Webseiten mit dynamischem Inhalt zu entwickeln [6, S. 3]. Die von Sun Microsystems entwickelte Web-Programmiersprache dient der dynamischen Erzeugung von HTML- und Extensible Markup Language (XML)-Ausgaben eines Webservers. Java-Code kann so in HTML- oder XML-Seiten eingebettet werden. Um vordefinierte Funktionalitäten einzubinden, werden Tag Libraries genutzt. Die Kompilierung des Codes erfolgt beim Anwendungsstart auf dem Applikationsserver. [7, S. 2]

2.8 JavaServer Faces

JSF ist ein komponentenbasiertes MVC-Framework-Standard zur Entwicklung von grafischen Benutzeroberflächen für Webapplikationen. Es basiert auf Servlets und der JSP-Technik. Hierzu wird ein Servlet-Container benötigt.
JSF benötigt eine Konfigurationsdatei, die faces-config.xml, siehe Abschnitt 2.8.1, in welcher Managed-Beans und Navigationsfälle festgelegt werden können. Zudem kann hier die Lebensdauer der einzelnen Beans angegeben werden. Auf diese wird in Abschnitt 6.4 noch genauer eingegangen. Ebenso muss eine web.xml im WEB-INF Verzeichnis liegen, sodass auf die JSF- Technologie zugegriffen werden kann. Hier kann ein Mapping auf die benötigten Endungen stattfinden.
Der Unterschied von JSF zu anderen, ähnlichen Frameworks besteht darin, dass

dahinter eine Standardspezifikation (JSR-127) steht, welche eine gute Toolunterstützung und eine weitreichende Akzeptanz garantiert. [7, S. 5] [8]

2.8.1 faces-config.xml

Die faces-config.xml ist eine Konfigurationsdatei für JSF. In dieser können Managed-Beans eingetragen werden, sodass sie später über Expression Language (EL) aufrufbar sind. Ebenso kann hier ihre Lebensdauer, wie in Abschnitt 6.4 beschrieben, konfiguriert werden. Benutzerdefinierbare Validierungsnachrichten können hier ebenfalls eingestellt, sowie RESOURCE-BUNDLES definiert werden. Außerdem können VIEW-HANDLER und PHASELISTENER eingetragen werden. Es gibt noch mehr Anwendungsfälle für die faces-config.xml, wie zum Beispiel eine benutzerdefinierte EXCEPTION HANDLER FACTORY die bestimmt werden kann.

2.9 Facelets

Ab JSF 2.0 ist Facelets die standardmäßige View Declaration Language (VDLGE) für JSF. JSP wird in aktuellen Versionen zwar noch unterstützt, aber nicht empfohlen und gilt als überholt. Facelets ist eine leichtgewichtige, aber mächtige Deklarationssprache, die mit Hilfe von HTML-Style-Templates JSF-Ansichten und den Komponentenbaum erstellt. JSF-Ansichten werden normalerweise als Extensible HyperText Markup Language (XHTML)-Seiten erstellt. Einige wichtige Features von Facelets sind die Unterstützung von Facelets-Tag-Libraries, siehe Abschnitt 2.12, zusätzlich zu JSF und der JSP Standard Tag Library (JSTL), die Unterstützung von Expression Language, Templating für Komponenten und Seiten, wie in Abschnitt 6.2 beschrieben, und die Nutzung von XHTML um Webseiten zu erstellen. Facelets haben den Vorteil, dass sie die Wiederverwendung des Codes mit Templating unterstützen. Außerdem bieten sie durch die individuelle Anpassbarkeit eine funktionale Erweiterbarkeit von Komponenten und anderen serverseitigen Objekten, verkürzen die Kompilierungsdauer, führen die EL-Validierung zur Kompilierzeit durch und bieten ein hochperformantes Rendering. Facelets reduzieren also den Zeit- und Entwicklungsaufwand. [9]

2.10 JavaScript

JavaScript (JS) wurde ursprünglich als LiveScript in einen Internet-Browser von Netscape eingebaut, um Einfluss auf HTML-Seiten nehmen zu können. Die Syntax war an Java angelehnt. Aus Gründen des Marketings wurde die Sprache schließlich in JS umbenannt. JS verbreitete sich schnell und es wurde selbstverständlich, dass eine Skriptunterstützung in Browsern gegeben war. [10] Zusammengefasst kann man sagen, „JavaScript ist eine Skriptsprache welche ursprünglich für dynamisches HTML in Webbrowsern entwickelt wurde um Benutzerinteraktionen auszuwerten und Inhalte zu verändern, nachzuladen, oder zu generieren.“ [11]

2.11 Asynchronous JavaScript and XML

„Asynchronous JavaScript and XML (AJAX) bezeichnet ein Konzept der asynchronen Datenübertragung zwischen einem Browser und dem Server.“ [12] Dies ermöglicht es, die Anzeige zu verändern, ohne sie komplett neu zu laden.
AJAX ist keine einzelne Technologie, sondern besteht aus vielen Technologien, von denen jede ihre Aufgabenbereiche hat. AJAX beinhaltet eine an Standards orientierte Darstellung mit Hilfe von XHTML und Cascading Stylesheets (CSS), eine dynamische Anzeige und Interaktion durch das Document Object Model (DOM), Datenaustausch und Datenmanipulation mit XML und Extensible Stylesheet Language Transformations (XSLT), asynchrone Datenabfrage durch XMLHttpRequest (XHR) und JS, welches alles zusammen bringt. AJAX ermöglicht es also, nur bestimmte Daten abzufragen und führt Serverabfragen so durch, dass der Nutzer möglichst wenig davon bemerkt, indem Wartezeiten umgangen beziehungsweise minimiert werden, wie man in Abbildung 2.2 erkennen kann. [13]

2.12 Tag Libraries

Tag Libraries wurden erstellt um Java-Code innerhalb einer JSP-Seite, ohne JS, auszuführen. Tag Libraries erlauben es, HTML-ähnliche Blöcke zu erstellen, welche einer Java-Klasse zugeordnet sind, die dann die Businesslogik ausführt. Es besteht die Möglichkeit, eigene Tag Libraries zu erstellen, welche spezifisch genutzt werden können. Dies ist allerdings nicht unbedingt nötig, da verschiedene Frameworks sehr

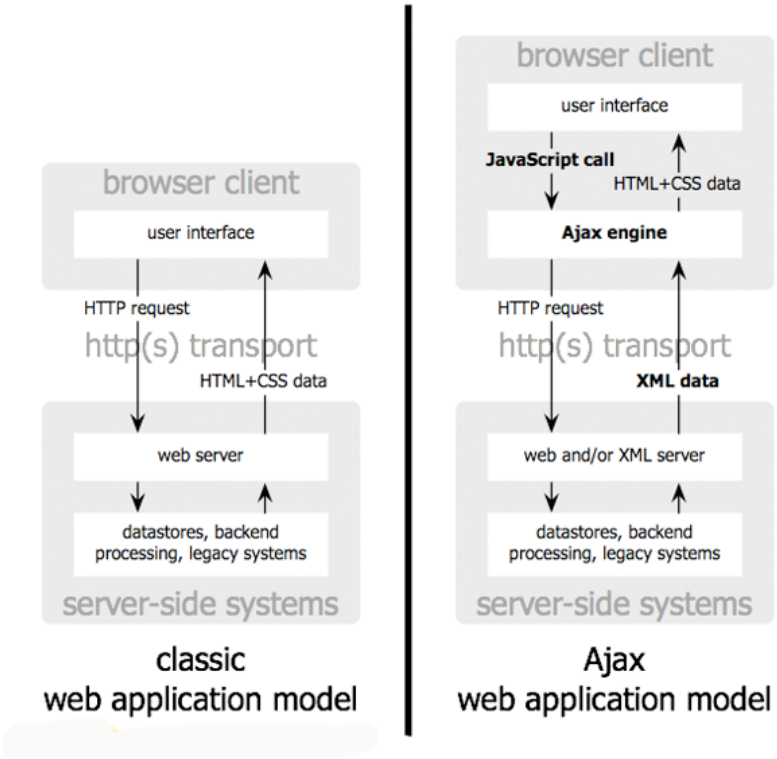

Abbildung 2.2: Ajax im Vergleich [13]

viele vorgefertigte Libraries enthalten, welche eingebunden und verwendet werden können. [14] Von Tag Libraries wird Gebrauch gemacht.

2.13 Model-View-Controller

Beim MVC-Konzept (Abbildung 2.3) sollen Darstellung, Anwendungslogik und Steuerung voneinander getrennt sein, um eine gute Modularisierung zu erhalten und einzelne Komponenten anderweitig wiederverwenden zu können. Im Fall von JSF wird das so genannte MVC 2 umgesetzt. [8, S. 10] [7, S. 5]

- MODEL. Das Model steht für die **Datenhaltung** und repräsentiert den aktuellen Zustand der Anwendung. Hier wird Zugriff auf die Zustandsdaten gewährt und eine Kommunikation mit externen Datenquellen, zum Beispiel einer Datenbank, findet statt. Im Fall der neuen AKTE sind dies die Managed-Beans und Klassen des Projekts. Das Model kann in verschiedenen Anwendungen genutzt werden. Im MVC 2-Konzept der neuen AKTE kann das Model über den Controller die View bearbeiten. Ebenso könnte das OBSERVER-

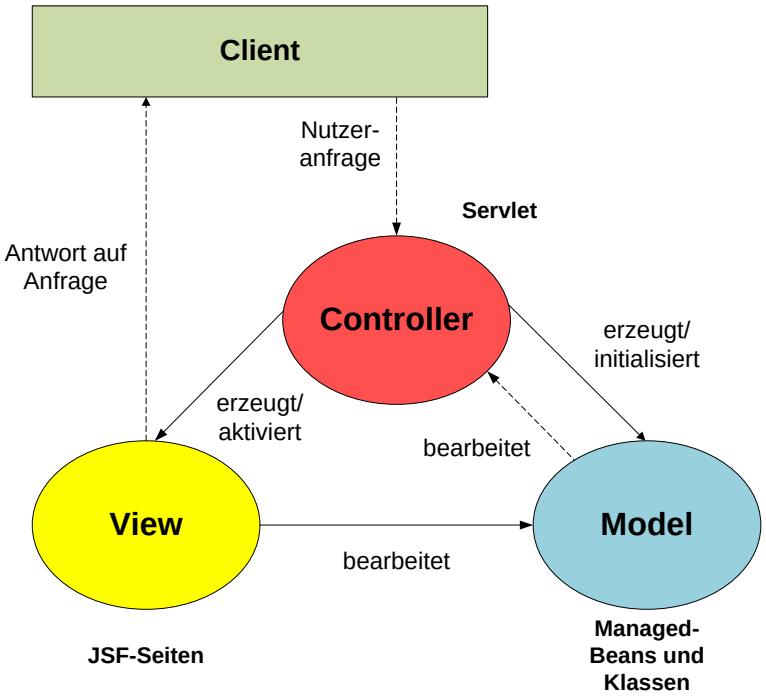

Abbildung 2.3: Model-View-Controller 2 in der neuen AKTE

PATTERN [15, S. 257] realisiert werden. Das könnte von Nutzen sein, wenn zum Beispiel ein anderer Nutzer einen Patienten löscht, dieser aber in der eigenen View noch angezeigt wird. Im aktuellen Model wird diese Möglichkeit allerdings nicht beachtet, da sie noch nicht realisiert ist.

- VIEW. In der View werden Daten des Models visualisiert. In dieser werden **Darstellung** und **Präsentation** geregelt. Im Fall der neuen AKTE repräsentieren die JSF-Seiten die View. Die JSF-Seiten werden über den Controller mit Daten aus dem Model befüllt, beziehungsweise der Controller kommuniziert mit der View, um Änderungen mitzuteilen. Die View bearbeitet das Model wenn Daten sich ändern. Die JSF-Seiten rendern die Antwort auf die Benutzeranfrage.
- CONTROLLER. Der Controller übernimmt die **Steuerung** der Anwendung. In der neuen AKTE dient das FacesServlet als solcher. Der Controller erwartet die Benutzeranfragen und erzeugt und initialisiert dann das Model und die View.

2.14 Contexts and Dependency Injection

Dependency Injection (DI) reglementiert die Abhängigkeit eines Objekts zur Laufzeit. [16] Contexts and Dependency Injection (CDI) unterstützt die Standard-DI für Java. [17] Es bietet eine Architektur, die Java EE Komponenten, wie zum Beispiel Servlets, Enterprise Beans und Java Beans ermöglicht innerhalb des definierten Lebenszyklus einer Applikation (in Abschnitt 6.4 beschrieben) zu existieren. [18]

Einige CDI-Features: [19]

- Typsichere Injektion über Java-Typen
- Fast jedes Java-Objekt kann injiziert werden
- EL-Integration

Da die AKTE einen Apache Tomcat als ServletContainer verwendet, muss CDI manuell integriert werden, dies wäre nicht der Fall, wenn sie auf einem Applikationsserver laufen würde der Java EE 6 unterstützt, da CDI dort bereits integriert und einsatzbereit ist. [7, S. 295]
Der Integrationsvorgang und die Begründung, weswegen er notwendig ist, wird im Abschnitt 6.11 genauer behandelt.

Kapitel 3

Stand der Technik

In diesem Kapitel wird beschrieben, welche Technologien zum Zeitpunkt des geplanten Starts der Migration im zu migrierenden Projekt verwendet werden. Der Ist-Zustand wird erläutert und die verwendeten Technologien aufgeführt und genauer erklärt. Ebenso wird verständlich gemacht, weswegen die vorhandenen Technologien bisher verwendet wurden und in welchem Zusammenhang diese stehen.

3.1 Java

Das Backend der AKTE ist in Java geschrieben und wird mit Java 6 kompiliert. Die AKTE wird offiziell ab Java 6 unterstützt. Es sind Kundensysteme vorhanden, die mit einer höheren Java-Version nicht mehr funktionieren würden, da sie auf 6 laufen und einen mit einer höheren Java-Version kompilierten Code nicht ausführen könnten. Abwärtskompatibilität ist gegeben, sodass höhere Versionen beim Kunden ebenfalls unterstützt werden. Eine Umstellung auf eine neue Java-Version würde einen großen Aufwand mit sich bringen, da Krankenhaussysteme selten alle auf dem neuesten Stand sind. Erst wenn eine Analyse der Kundensysteme ergibt, dass nur noch vereinzelt Java 6 vorhanden ist und der Großteil schon 8 verwendet, oder die Wartung und der Support von 6 nicht mehr tragbar ist, wird ein Update durchgeführt und die Unterstützung für 6 eingestellt.

3.2 Apache Tomcat

Die genutzte Tomcat-Version liegt bei der Versionsnummer 7.023, da diese die aktuellste Version war, als im Zuge der Umstellung der Server von Java 5 auf Java 7, von Tomcat 5 auf Tomcat 7 aktualisiert wurde. Eine Umstellung auf jede nächsthöhere Version hätte den Aufwand nicht gerechtfertigt, da sehr viele Kompatibilitätsprobleme entstehen können und war nicht notwendig, da mit 7.023 alles wie vorgesehen funktionierte.

3.3 PostgreSQL

Bei CHILI wird PostgreSQL (Postgres) als Datenbanksystem verwendet. Postgres ist ein Open-Source objektrationales Datenbankmanagementsystem, welches seit über 15 Jahren entwickelt wird. Weitgehend entspricht Postgres dem Structured Query Language (SQL)-Standard, dennoch gibt es eine Reihe dokumentierter Funktionalitäten, die Postgresspezifisch sind. Postgres läuft auf allen gängigen Betriebssystemen, Windows und Linux eingeschlossen. [20]

3.4 JavaServer Pages

JSP wird in der aktuellen Version der AKTE verwendet, da es die Möglichkeit mit sich bringt die Anzeige dynamisch zu gestalten. Mit JSP ist es möglich aktive Elemente in den HTML-Code zu integrieren. Zusätzlich zu dynamischen JSP-Tags ist es ab der ersten Version von JSP auch möglich Java-Code in der Oberfläche zu nutzen. Es wird zwar in der meisten Literatur die JSP behandelt empfohlen die Code-Nutzung auf ein Minumum zu beschränken, jedoch wird dies oft nicht eingehalten. [6, S. 4]
Als die AKTE erstellt wurde, wurde JSP gewählt, da es den damaligen Anforderungen entsprach und das MVC-Konzept nicht beachtet wurde.

Nachteile

JSP kann Schwierigkeiten bei Wartbarkeit und Verständlichkeit mit sich bringen, da es schnell dazu verleitet, eine große Menge an Java-Code mit HTML-Tags zu

vermischen. Eine klare Trennung zwischen Anzeige und Logik ist so nicht mehr deutlich zu erkennen. Ein weiterer Nachteil ist, dass durch die späte Kompilierung viele Fehler erst zu Beginn der Laufzeit auffallen. [7, S. 3]

3.5 JavaServer Faces

Die genutzte Version 1.1 von JSF (siehe Abschnitt 2.8) wurde im Mai 2004 verabschiedet und ist die erste BugFix-Version von JSF. In der Version 1.1 wird JSP als standardmäßige VDLGE für JSF genutzt. [21]
JSF 1.1 war zum Zeitpunkt der Programmierung der Akte die aktuellste Version und wurde seitdem nicht mehr aktualisiert. Vermutlich da es einige Kompatibilitätsprobleme gab und die Ansicht besteht, dass ein funktionierendes System nicht geändert werden muss. Zudem sind oft viele Anpassungen notwendig um ein Framework zu aktualisieren, was eine gute Begründung hierfür erfordert. Der Nachteil einer späten Aktualisierung ist allerdings, dass die Anpassungen mit dem Sprung über mehrere Versionsnummern immer aufwendiger umzusetzen sind, sodass der Zeitaufwand und die Hemmschwelle dafür stetig zunehmen.

Nachteile

Im Mai 2006 wurde bereits JSF 1.2 veröffentlicht, welches auch auf der MyFaces-Seite als Mindestversion empfohlen wird. [22] In der bisherigen AKTE wird noch immer JSF 1.1, die erste Bugfix-Version von JSF, verwendet, welche allerdings nicht mehr dem neuesten Stand entspricht. Zudem bieten die SCOPES in JSF 1.1 [23] wenig Möglichkeiten im Vergleich zu JSF 2.2, siehe Abschnitt 6.4.

3.6 Apache MyFaces

„MyFaces ist eine Open-Source-Implementierung der Java-Server-Faces-Spezifikation.“ [24] Die Version 1.1.0 des Frameworks wurde am 11. März 2004 veröffentlicht und benötigt mindestens Java 1.3, JSP 1.2, JSTL 1.0, eine Java Servlet 2.3- und eine JSF 1.1- Implementierung. [22] Eine Alternative zu Apache MyFaces wäre beispielsweise Oracle Mojarra JavaServer Faces, eine andere Implementierung

des JSF-Standards gewesen. Apache MyFaces schneidet allerdings bei der Performance und anderen Eigenschaften besser ab und ist somit die performantere und laut Studie [25, S. 5] auch bessere Alternative.

3.7 MyFaces Tomahawk

Für die Oberfläche der Akte wurden einige Tag Libraries und UI-Komponentensets genutzt, um auf bereits vordefinierte Komponenten zurückzugreifen und somit den Programmieraufwand einzuschränken. Tomahawk ist eine davon und bietet unter anderem einen Kalender, Datei-Upload und Datentabellen zum Gebrauch. Die Version 1.1.14 wurde verwendet, da sie mit der genutzten JSF-Version 1.1 kompatibel ist. [26]

3.8 Abhängigkeiten

Die AKTE beinhaltet einige Abhängigkeiten von anderen Projekten und Bibliotheken, in welchen zum Zeitpunkt der Arbeit ältere Frameworks verwendet werden. Diese gilt es festzustellen und nach Möglichkeit zu beseitigen und bei Bedarf durch andere Frameworks, oder neuere Versionen zu ersetzen. Ebenso gilt es herauszufinden, ob Bibliotheken vorhanden sind, welche in den neueren Bibliotheken bereits inkludiert, oder mittlerweile vom Standard-Java abgedeckt werden und somit obsolet sind.

Kapitel 4

Einzusetzende Technologien und Frameworks

In diesem Teil der Arbeit wird ein Ausblick auf die zu verwendeten Technologien und Frameworks gegeben. Zudem werden diese beschrieben, erklärt Alternativen dazu aufgeführt und erläutert. Ebenso wird herausgearbeitet, wieso die erwähnten Technologien ausgewählt wurden und weswegen etwaige Alternativen ausscheiden.

4.1 JavaServer Faces 2.2

JSF 2.2 ist die aktuelle Version von JSF. Es wird angestrebt diese zu nutzen. Einige wichtige Eigenschaften, die diese Version mit sich bringt, sind nachfolgend aufgeführt.

- Ab Version 2.0 werden mit Facelets und JSP zwei konkrete Implementierungen einer VDLGE unterstützt.
- AJAX wurde in den Standard integriert.
- Eine Dateiupload-Komponente wird ab 2.2 geboten, was die Umsetzung des Datei-Hochladens vereinfacht.
- Die Möglichkeit mit *f:resetValues* Eingabefelder zurückzusetzen, optional auch innerhalb von AJAX-Komponenten.
- Faces-Flows ermöglicht das Zusammenfassen einzelner Komponenten zu Modulen.
  [7, S. 7]

4.1.1 Gegenüberstellung von Facelets und JSP

JSP und JSF wurden für unterschiedliche Einsatzzwecke entworfen. Eine JSP-Anfrage wird, im Gegensatz zu JSF, in einer einzigen Phase bearbeitet, was Probleme mit sich bringen kann. Außerdem besteht die Möglichkeit Java-Code einzubinden. Deswegen wurde ab JSF Version 2.0 komplett auf Facelets gesetzt, welche sich perfekt in den Lebenszyklus von JSF einfügen. [7]

4.2 PrimeFaces

PrimeFaces wurde im Jahr 2008 von PrimeTek veröffentlicht und ist eine Open-Source Komponentenbibliothek für JSF [27], welche Funktionen weit über den JSF-Standard hinaus bietet. Dazu gehören unter anderem Eingabekomponenten, Komponenten zur Darstellung von Listen und Tabellen, sowie Komponenten für Diagramme und GoogleMaps. Ebenso existieren AJAX-Erweiterungen. Um die Standardfunktionen von PrimeFaces zu nutzen, muss lediglich eine Jar-Datei eingebunden werden. Der Fokus von PrimeFaces liegt auf Leichtgewichtigkeit und Performance. [7, S. 309] Aktuell wurde die Version 5.3.1 eingebunden. Je nach Entwicklungsstand bei Beendigung dieser Arbeit, kann noch auf PrimeFaces 6 aktualisiert werden.

4.2.1 Warum PrimeFaces

Neben PrimeFaces standen ICEfaces von ICESOFT und RichFaces von JBoss zur Auswahl. RichFaces schied als erste Option aus, da der angebotene Support für die Zwecke der AKTE unzureichend war und das ganze Auftreten unausgereift erschien. Diese Entscheidung erwies sich als die Richtige, da die Weiterentwicklung für RichFaces im Juni 2016 eingestellt wurde und somit eine Umstellung notwendig geworden wäre. [28]
Beim genaueren Betrachten von ICEfaces und darauf folgender E-Mail-Kommunikation mit ICESOFT, um Informationen zu kommerziellem Gebrauch herauszufinden, schied auch dieses aus, da die benötigten Lizenzen keinen Mehrwert im Gegensatz zu PrimeFaces geboten haben, aber deutlich teurer gewesen wären. Zudem schnitt PrimeFaces bei verschiedenen, selbst durchgeführten Performance-Tests, welche mit Datentabellen und Testimplementierungen erstellt wurden, besser ab. Support

Optionen konnten je nach Bedarf Jahr für Jahr geändert werden, was ebenfalls ein wichtiger Punkt war.
PrimeFaces erwies sich als das beste Framework für die Aktualisierung der AKTE. Bestärkt wurde diese Entscheidung durch gute Bewertungen, die eigene Testimplementierung und ein gutes Rating bei DevRates, siehe Abbildung 4.1 [29]. DevRates ist eine online Gemeinschaft für Entwickler, die Open-Source Bibliotheken benutzen. Dort sind um die 200 der beliebtesten Java, JS, Ruby und mobile Frameworks gesammelt. Hauptsächlich befasst sich die Seite mit Bewertungen der Benutzer und erfasst Ranglisten in verschiedenen Bereichen. [30]

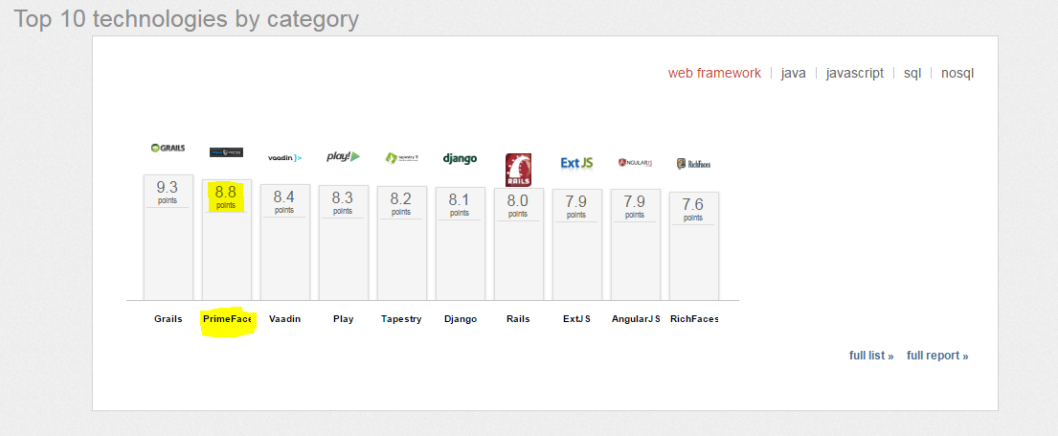

Abbildung 4.1: devrating PrimeFaces 2016 [31]

4.3 OmniFaces

OmniFaces ist eine Hilfsbibliothek für JSF 2, um dessen Standardaufgaben mit der Standard-JSF-API zu erleichtern. OmniFaces fokussiert sich nicht auf die visuelle Darstellung der Komponenten, sondern deren Nützlichkeit und ist somit eine gute Ergänzung zu PrimeFaces. Der Grund, weswegen OmniFaces eingesetzt wird, ist eine angebotene Fehlerbehandlung für AJAX. [32] Eingebunden wird die Version 1.12.1.

4.4 Apache OpenWebBeans

Es existieren mehrere Implementierungen für CDI. Apache OpenWebBeans ist eine Open-Source-Implementierung davon. [33] Zu den bekanntesten zählen aktuell Apache OpenWebBeans und Weld von JBoss. [7, S. 295]

4.4.1 Gegenüberstellung von OpenWebBeans und Weld

Apache OpenWebBeans wurde ausgewählt, da gute Beispiele und Hilfen verfügbar sind [7, S.331] und es sich einfach in eine JSF-Anwendung integrieren lässt [7, S. 295]. Es wurde nach dem downloadbaren Beispielcode zum JSF-Buch [7] vorgegangen. Aus Kompatibilitätsgründen wurde die Version 1.2 ausgewählt. Beim Versuch der Integration von JBoss Weld traten einige schwer identifizierbare Probleme auf und die verfügbaren Leitfäden widersprechen sich oft, oder lassen sich nicht mit den in der AKTE vorhandenen Voraussetzungen und Technologien vereinbaren. Eine Integration mit Tomcat als Servlet-Container erwies sich unter Anderem als problematisch, was vermutlich an Kompatibilitätsproblemen lag.

4.5 Apache Tomcat

Im Zuge der Entwicklung der neuen AKTE wurde Tomcat aufgrund eines Performance-Bugs auf Version 7.069 aktualisiert. Weswegen und welche Probleme dabei aufkamen wird genauer in Abschnitt 6.13 ausgeführt. Eine Aktualisierung des Tomcat war ursprünglich nicht geplant, wurde dann allerdings, aus verschiedenen, triftigen Gründen, doch durchgeführt.

4.6 Alternative Technologien

Die Entscheidung, bei einem JSF Framework zu bleiben und Apache MyFaces weiter zu nutzen, ergab sich unter Anderem mit der Wahl des Konzepts der Migration, welches in Kapitel 5 genauer beschrieben wird.
Man hätte ebenso Mojarra als JSF Framework nutzen, oder auf ein JS Framework wie zum Beispiel AngularJS umsteigen können. Warum MyFaces und nicht Mojarra weiter genutzt wird, wurde bereits in Abschnitt 3.6 ausgeführt.

Ein JS Framework schied aus, da es für die Zwecke der AKTE kaum wartbar, und der Aufwand für die neue Oberfläche enorm gewesen wäre; Die in MyFaces genutzten Komponenten hätten nicht weiter genutzt werden können, sondern selbst programmiert werden müssen. Außerdem wäre das Umsetzen des MVC Konzepts aus Abschnitt 2.13 mit einem JS Framework nicht möglich gewesen und es wäre weiterhin Code im Frontend zu finden gewesen. Des Weiteren war das vorhandene Backend auf die Kommunikation mit JSF ausgelegt und hätte durch ein JS-Framework somit komplett umgebaut werden müssen.

Kapitel 5

Konzepte

Hier wird auf die zur Auswahl stehenden Konzepte für die Migration mit Vor- und Nachteilen eingegangen und diese werden beschrieben und erklärt. Zudem wird ausgeführt auf welche Vorgehensweise die Wahl gefallen ist und aus welchen Gründen die Entscheidung auf diese gefallen ist.

5.1 Neuprogrammierung

Der vorhandene Code wird bei der Neuprogrammierung komplett verworfen und die AKTE wird von Grund auf, mit vollständigem Funktionsumfang, neu programmiert. Nach Fertigstellung der neuen Version wird die bisherige AKTE durch diese ersetzt werden.

Pro

Der Vorteil einer Neuprogrammierung liegt darin, dass man bei einem neuen Frontend keine Rücksicht auf etwaige Kompatibilitätsprobleme nehmen muss und auch, dass im Backend kein obsoleter, nicht genutzter oder schlecht programmierter Code mitgezogen wird. Die Logik kann man dabei erneut überdenken und da alles von Grund auf neu erstellt wird, eventuell verbesserte Algorithmen finden.

Contra

Gegen eine Neuimplementierung spricht, dass sie sehr aufwendig ist und viel Zeit benötigt, was die Gefahr von schlechten Veränderungen mit sich bringt. Nach der Fertigstellung einer Neuprogrammierung muss zudem ein zusätzlicher Zyklus eingeplant werden, in welchem Änderungen, die in der Entwicklungszeit an der bestehenden AKTE vorgenommen wurden, eingebaut und angepasst werden können.
Da eine Neuprogrammierung eine lange Entwicklungszeit voraussetzt, gibt es eine große Zeitspanne in der in beiden Projekten zeitgleich entwickelt wird. Dies beansprucht viele Personentage und ist somit sehr kostenintensiv. Zusätzlich besteht bei einer Neuerstellung das Risiko einer Verschlechterung oder einer Änderung des Umfangs der Funktionen. Nutzer könnten Probleme haben, das Produkt noch als solches zu erkennen.

5.2 Teilmigration

Bei einer Teilmigration wird die Modernisierung an der aktuellen AKTE nach und nach vorgenommen und in kleineren Aktualisierungsschritten ausgeliefert.

Pro

Der Vorteil der Teilmigration liegt eindeutig darin, dass nach der Fertigstellung nichts aktualisiert, beziehungsweise nachträglich aus dem bisherigen Projekt hinzugefügt werden muss, was sich während des Migrationsvorgangs geändert hat, da es weiterhin nur ein Projekt gibt. Außerdem wird der Anwender langsam an die neue Version gewöhnt wund es ist weniger kostenintensiv, da nur an einem Projekt gearbeitet werden muss.

Contra

Nachteile die sich bei einer Teilmigration auswirken sind, dass die Bibliotheken, die für das neue Framework benötigt werden, mit denen des neuen Frameworks kompatibel sein müssen. Auch muss man mit zwei verschiedenen JSF-Versionen arbeiten und entwickeln. Bei einer Teilmigration ist die Gefahr der Seiteneffekte

der teilmigrierten, ausgelieferten Versionen sehr hoch und die Fehleranfälligkeit in einem kritischen Bereich. Nach jedem Migrationsschritt muss komplett neu getestet werden, was sehr zeitaufwendig und zum Teil auch nicht möglich ist.

5.3 Vollmigration

Im Falle einer Vollmigration wird zunächst die neue AKTE mit allen Modernisierungen auf einem separaten Entwicklungszweig entwickelt und nach Fertigstellung dieser, komplett auf die neue Version umgestellt.

Pro

Bei einer Vollmigration ist ein positiver Aspekt, dass keinerlei Seiteneffekte auf die dem Kunden auszuliefernde Software zu erwarten sind, da erst ausgeliefert wird, wenn alles komplett implementiert und getestet ist. Der Kunde bekommt in der Entwicklungszeit nur Auslieferungen der bestehenden, funktionierenden Software. Obsolete Bibliotheken können zum Beispiel ohne Risiko gelöscht werden, da die dadurch nicht mehr benutzbaren Komponenten ebenfalls nicht mehr benötigt werden und aus der neu entstehenden Software vor Fertigstellung nichts ausgeliefert werden muss. Ein weiterer Vorteil ist, dass eine Refaktorisierung am bestehenden Code unbedenklich durchgeführt werden kann, da keine stetig funktionierende Software verfügbar sein muss.

Contra

Es muss stetig an zwei verschiedenen Versionen der AKTE gearbeitet werden, da auch in der alten Version noch Kundenwünsche und Fehlerbehebungen durchgeführt werden. Dabei handelt es sich um einen komplexen Vorgang, wobei beide Versionen auf einem aktuellen Stand gehalten werden müssen und die Entwicklung dennoch Fortschritte verzeichnen muss.

5.4 Finales Vorgehen

Eine Neuprogrammierung ist im Fall der AKTE nicht sinnvoll, da eine solche sehr aufwendig ist und die Funktionen der AKTE sich weder grundlegend ändern sollen, noch komplett neue Funktionen geplant sind. Die Migration soll aufgrund einer Modernisierung stattfinden. Das Backend ist weitestgehend zu erhalten und lediglich mit geringen Verbesserungen und Neuerungen an das neue Frontend angepasst werden. Vor diesem Hintergrund ist eine Neuprogrammierung unwirtschaftlich, da die Dauer und das Risiko, die sie mit sich bringt, nicht gerechtfertigt werden kann.

Die Möglichkeit der Teilmigration scheidet aus, da in der neuen Version der AKTE andere oder neuere Bibliotheken genutzt werden sollen, welche mit den alten Bibliotheken und dem alten Code nicht kompatibel sind. Beide Bibliotheken zeitgleich zu nutzen und die Software komplett funktionsfähig zu halten ist durch die Inkompatibilität der Bibliotheken nicht möglich, da gar keine Funktionalität mehr gegeben wäre. Somit ist auch eine Auslieferung in Teilschritten nicht realisierbar.

Aus Gründen der geringsten Komplexität der vorgestellten Möglichkeiten und zugunsten der Stabilität der bestehenden Software die beim Kunden genutzt und weiterentwickelt wird, wird nach der *Vollmigration auf einem separaten Entwicklungszweig* vorgegangen. Trotz der Schwierigkeit der sich weiterentwickelnden, alten Version der AKTE ist geplant, diese Möglichkeit umzusetzen, da alle Funktionalitäten beibehalten werden können, aber dennoch Änderungen am Backend ohne riskante Seiteneffekte möglich sind. Das genauere Vorgehen wird in Kapitel 6 ausführlich behandelt.

Kapitel 6

Durchführung der Migration

Das Migrationskapitel behandelt die praktische Durchführung der Vollmigration auf einem separaten Entwicklungszweig und erläutert die Schritt-für-Schritt-Vorgehensweise und Ideenfindung. Probleme und zugehörige Lösungswege werden ebenfalls behandelt und ausgeführt.

6.1 Abhängigkeiten und Bibliotheken

Zu Beginn wurden die benötigten Bibliotheken eingebettet. Hierfür wurden zunächst die vorhandenen Bibliotheken mit den aktuellsten noch kompatiblen Versionen ersetzt, oder, falls nicht mehr benötigt, entfernt.
Tomahawk beispielsweise sollte nicht mehr eingesetzt werden und konnte entfernt werden. Zu Beginn des Vorgehens blieben noch einige Bibliotheken im Projekt, deren weitere Nutzung noch nicht feststand. Diese sollten mit der ersten stabilen, lauffähigen Version der AKTE erneut auf Nutzung überprüft werden, um dann unnötige Bibliotheken entfernen zu können und Seiteneffekte auszuschließen.

Da die Entscheidung auf PrimeFaces als zu nutzendes Framework gefallen ist, wurde das aktuellste PrimeFaces-Jar hinzugefügt und in die build.xml integriert.
Wie in Abschnitt 6.11 beschrieben, sollte mit Flows gearbeitet werden. Die Integration von Flows sollte mit Hilfe von OpenWebBeans geschehen, weswegen die maximal unterstützte Version von OpenWebBeans und die damit verbundenen Abhängigkeiten eingebunden wurden. Für die AJAX-Fehlerbehandlung sollte auf OmniFaces gesetzt und die zugehörigen Jars ebenfalls integriert werden.
Die Anzahl der Bibliotheken stieg mit dem Vorankommen der Migration, da die

Nutzung einiger Komponenten zu Beginn noch nicht feststand und sich im Laufe der Entwicklung Kompatibilitäten änderten.

Hier eine Auflistung der wichtigsten integrierten Bibliotheken:

- MyFaces-bundle Version 2.2.8
- PrimeFaces Version 5.3.10
- OmniFaces Version 1.12.1
- OpenWebBeans mit allen benötigten Bibliotheken (el22, impl, jsf, resource, spi, web) Version 1.2.0

6.2 Templating

Nachdem durch Einbindung der benötigten Bibliotheken die technischen Voraussetzungen erfüllt waren, wurde das Prinzip für Layout und Design überdacht. In der alten Version der AKTE musste in jeder JSP-Datei explizit eine Datei für den HEADER und eine Datei für den FOOTER eingebunden werden. Da PrimeFaces ein gutes Konzept für Templating bietet, war es naheliegend die PrimeFaces Standardlösung zu verwenden und über das *ui:insert*-Tag, aus der Facelets-Tag-Library, dynamische Inhalte zu bestimmen.

Hierzu wurden im TEMPLATE selbst, welches die grundlegende Seitenstruktur festlegt, mit *ui:insert* ein oder mehrere Bereiche über *name* definiert, in welche zur Laufzeit der so benannte Teil aus der aufgerufenen Zieldatei eingefügt wird. Verdeutlicht wird dies in Abbildung 6.1.

Die Zieldatei selbst braucht keine eigenes Grundgerüst, da sie mit *ui:composition* über das *template*-Attribut mit dem gewählten TEMPLATE verbunden wird. Alles was im TEMPLATE mit *ui:insert* definiert ist, wird mit dem Inhalt aus *ui:define* der Zieldatei ersetzt. Allerdings nur, wenn das *name*-Attribut den gleichen Wert hat. Dies ist in Abbildung 6.2 nachvollziehbar.
Es besteht die Möglichkeit, im TEMPLATE einen Standardinhalt für definierte Container anzugeben der angezeigt wird, wenn die aufgerufene Datei keinen überschreibenden Inhalt enthält. Dieser wird innerhalb des entsprechenden Tags im TEMPLATE eingefügt.

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:p="http://primefaces.org/ui">

<h:head>
  <title>CHILI/Telemedizinakte</title>
  <link rel="stylesheet" type="text/css" href="style.css" ></link>
</h:head>

<h:body>

  <p:layout fullPage="true">

    <p:layoutUnit position="north">
      <!-- Das Menü wird ins Template inkludiert -->
      <ui:include src="#{menuBean.menuToInclude}" />
    </p:layoutUnit>


    <p:layoutUnit position="center">
      <ui:insert name="content">
        Hier erscheint was in der aufgerufenen Datei als "content"
        definiert wurde.
        Ohne definierten content, erscheint dieser Text.
      </ui:insert>
    </p:layoutUnit>

    <p:layoutUnit position="south">
      <ui:insert name="footer">
      Hier erscheint was in der aufgerufenen Datei als "footer"
      definiert wurde.
      Ohne definierten footer, erscheint dieser Text.
      </ui:insert>
    </p:layoutUnit>

  </p:layout>

</h:body>
</html>
```

Abbildung 6.1: Template Beispiel

```
<ui:composition template="templates/template.xhtml"
  xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html">

<!-- Inhalt der den Bereich mit dem Namen "content" ersetzt -->
  <ui:define name="content">

    <h:panelGrid id="grid">
        <h:outputText value="Ich bin der Inhalt"/>
    </h:panelGrid>

  </ui:define>

</ui:composition>
```

Abbildung 6.2: Zieldatei die in das Template eingebunden wird

Die gewünschten Anforderung, ein einziges TEMPLATE in die aufzurufende Datei zu integrieren, in welche diese dann eingebettet wird, wurde mit diesem Standardkonzept erfüllt. Zunächst wurde auf dieser Grundlage weiter gearbeitet.

6.3 Managed-Beans

Managed-Beans, im nachfolgenden auch Beans genannt, sind besondere Java-Klassen die in der faces-config.xml definiert sind. Sie enthalten Daten und Funktionen für die Kommunikation mit dem Frontend. Bevor die erste Seite mit Inhalt aus dem Backend angezeigt werden konnte, musste die Kommunikation mit dem Backend eingerichtet werden.
Dies funktioniert bei JSF über EL und die faces-config.xml. Die Bean muss in der faces-config.xml mit ihrer Lebensdauer, auch SCOPE genannt, eingetragen werden, um dann über EL auf ihre Variablen und Funktionen zugreifen zu können.
Genau wie in der bisherigen AKTE muss die Bean keine weiteren Voraussetzungen erfüllen, als einen Konstruktor ohne Parameter, mit der Sichtbarkeit *public* zu besitzen.
Demnach war es, was den Backend-Zugriff betrifft, nicht notwendig, etwas am alten Backend zu ändern.

In der bisherigen faces-config.xml waren die Beans bereits eingetragen. Die nutzbaren SCOPES von JSF 2.2 und JSF 1.1 unterscheiden sich allerdings, siehe Tabelle 6.1. Deshalb war es sinnvoller, diese neu zu bestimmen.

Scope-Name	Verfügbar ab	Lebensdauer
None	JSF 1.0	Referenz
Request	JSF 1.0	HTTP-Request
Session	JSF 1.0	Dauer der Anwendersession
Application	JSF 1.0	Programmlaufzeit
Custom	JSF 2.0	Frei wählbar
Flash	JSF 2.0	Zwei HTTP-Requests
View	JSF 2.0/JSF 2.2	Bildschirmanzeige
Flow	JSF 2.2	Definitionsabhängig

Tabelle 6.1: Scopes in den verschiedenen JSF-Versionen [34]

Zudem machte es Sinn, sich bei jeder Bean erneut Gedanken darüber zu machen, wie lange ihre Lebensdauer sein sollte. Eine Übersicht über einige SCOPES und deren Eintragung in der faces-config.xml wird in Abschnitt 6.4 gegeben.

6.4 Standard-Bean-Scopes

Für jede Bean muss bestimmt werden wie lange ihre Instanz gültig ist. Je nach Anwendungsbereich einer Bean variiert der richtige SCOPE. Hier eine Übersicht über einige der SCOPES die in JSF 2.2 unterstützt werden und deren Lebensdauer, welche in Abbildung 6.3 grafisch dargestellt ist.

- NONE-SCOPE. Die Bean wird bei jedem einzelnen Aufruf neu erstellt. Dieser Bean-SCOPE wird allerdings kaum benutzt, da keinerlei Datenhaltung gegeben ist.
- REQUEST-SCOPE. Wählt man REQUEST-SCOPE, existiert die Bean die Dauer der HTTP-Anfrage über.
- VIEW-SCOPE. Eine Bean mit VIEW-SCOPE wird so lange erhalten, wie man sich in derselben Ansicht befindet. Beim Wechseln der Ansicht wird auch die Bean erneut erzeugt.
- SESSION-SCOPE. Hier kann so lange auf die gleiche Instanz der Bean zugegriffen werden, wie der Nutzer mit der Anwendung verbunden ist und eine Session besteht.
- APPLICATION-SCOPE. Eine APPLICATION-SCOPED-Bean bleibt für die gesamte Lebensdauer der Anwendung mit der gleichen Instanz für alle Nutzer erhalten.

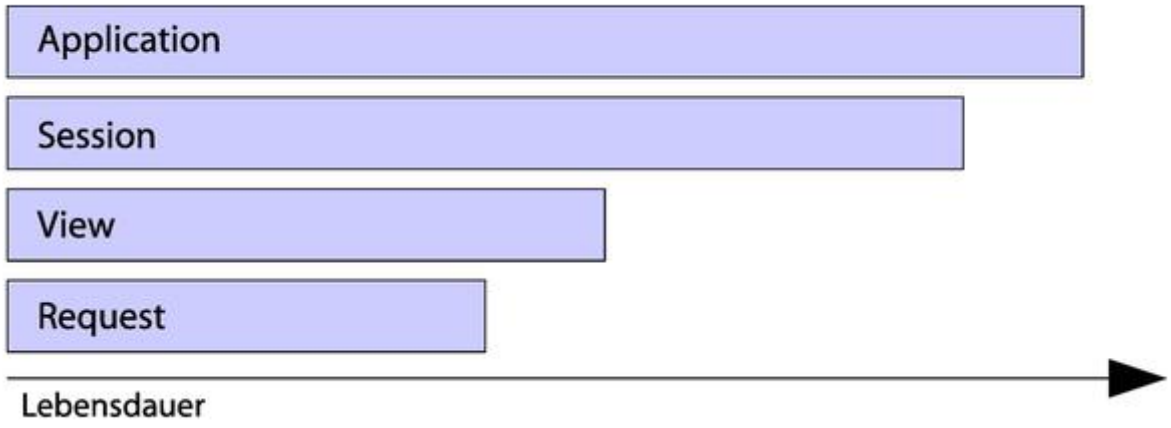

Abbildung 6.3: Vergleich der Lebensdauer der verschiedenen Scopes [36, S. 33]

- FLOW-SCOPE. Die Lebensdauer einer FLOW-SCOPED-Bean wird vom Anwender definiert. Solange man im definierten Bereich navigiert, bleibt die Bean erhalten. Diese Phase wird genauer in Abschnitt 6.11 behandelt.
- CUSTOM-SCOPE. Eine Anwenderspezifische Implementierung muss vorhanden sein, welche die Lebensdauer der Bean kontrolliert. CUSTOM-SCOPED wird allerdings kaum genutzt. [35]
- FLASH-SCOPE. Für den FLASH-SCOPE gibt es keine dazugehörige Annotation, das es sich eher um ein Container-Objekt handelt, als um einen richtigen SCOPE. In diesem kann ein Wert oder Objekt über eine Weiterleitung hinaus erhalten werden. [34]

Ein Beispiel der Konfiguration einer Bean in der faces-config.xml ist in Abbildung 6.4 zu sehen. [36, S. 31-33]

Die Instanz einer Bean sollte nur so lange erhalten bleiben, wie auf ihre Daten zugegriffen wird. Zu beachten ist vor allem, sie nicht länger als notwendig am Leben zu erhalten um Ressourcen in Bezug auf den Speicher zu sparen. Eine Ausnahme besteht, wenn das Erstellen im Konstruktor teuer, bezogen auf die Performance ist, da zum Beispiel viele performancelastige Initialisierungen stattfinden, die sich ständig wiederholen würden.

```
<managed-bean>
  <managed-bean-name>searchWindowBean</managed-bean-name>
  <managed-bean-class>
  de.chili.medizinakte.beans.SearchWindowBean
  </managed-bean-class>
  <managed-bean-scope>view</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>filemanagerBean</managed-bean-name>
  <managed-bean-class>
  de.chili.medizinakte.beans.FileManagerBean
  </managed-bean-class>
  <managed-bean-scope>view</managed-bean-scope>
</managed-bean>
```

Abbildung 6.4: Bean Scopes in der faces-config.xml

6.5 Migration von JSP zu JSF

Der Grundgedanke der Vollmigration ist, das bestehende Backend zu verwenden und ein neues Frontend aufzusetzen.
Bei genauem Betrachten der zu migrierenden JSP-Seiten traf man auf einen problematischen Sachverhalt. In der bisherigen AKTE wurden Logik und Anzeige nicht sauber getrennt. Die einzelnen JSP-Seiten enthalten teilweise mehr ausführbaren Code als JSP-Tags und sind damit sehr kompliziert aufgebaut und schwer verständlich. Eine Trennung von Logik und Ansicht, wie im MVC-Konzept in Abschnitt 2.13 beschrieben, war bisher nicht gegeben und sollte entwickelt werden.

Bevor man den Inhalt der einzelnen JSP-Seiten in JSF-Seiten übertragen konnte, musste also der vorhandene Java-Code in das Backend übertragen werden, um die bisherigen Funktionalitäten beizubehalten und dennoch ein MVC-Konzept zu erzielen. Hierzu mussten neue Beans, beziehungsweise Klassen angelegt werden. Anschließend konnte die JSF-Seite, welche auf die neue oder ergänzte Bean zugreifen sollte, erstellt werden.

Ein gutes Beispiel für den Umfang und die Unübersichtlichkeit der JSP-Seiten bietet die JSP-Datei des Logins, siehe Abbildung 6.5. Da die Login-JSP-Datei fast komplett aus ausführbarem Java-Code und nur wenigen JSP-Tags besteht, wurde eine *LoginBean* angelegt. Der Code aus der JSP-Datei konnte so in diese und in die bereits vorhandene *Login*-Klasse übertragen werden. Der *LoginBean* wurde die Lebensdauer einer VIEW zugeordnet.

Durch das Auslagern des Codes in die Bean wurden die Funktionen der bisher 487 Zeilen langen Login-JSP-Datei, welche schwer verständlich ist, in eine im Vergleich kürzere und verständlichere Login-JSF-Datei (Abbildung 6.6) übertragen.
In der neu erstellten JSF-Datei ist die komplette Logik im Backend zu finden. Sie befasst sich, wie geplant, nur noch mit der Ansicht und kommuniziert mit der *LoginBean*.
Um alle bisherigen Funktionalitäten beizubehalten, wie einen Login über eine URL mit Parametern, und keinen Code mehr im Frontend zu belassen, wurden noch einige zusätzliche Klassen implementiert. Bereits jetzt war abzusehen, dass der Vorgang, ein neues Frontend auf das bestehende Backend zu setzen, komplizierter und langwieriger ist, als bisher angenommen. Jede zu migrierende JSP-Seite erforderte Anpassungen im Backend.

6.6 Die Startseite mit PrimeFaces und JavaServer Faces 2.2

Nach dem Festlegen des grundlegenden Design-Prinzips und der Datenzuordnung zum Benutzer durch das Einloggen über die *LoginBean*, sollte nun die erste Inhaltsseite der AKTE mit Hilfe der neuen Technologien angezeigt werden. Da die Patientenanzeige den Kern der AKTE darstellt und diese auch als die Standardstartseite nach dem Login konfiguriert ist, wurde diese als erste umgesetzt.

Zunächst wurde ein PrimeFaces *dataTable* erstellt und anschließend, befüllt mit einer Liste von Beispieldaten aus einer Bean, angezeigt. Mit Hilfe der PrimeFaces ShowCases konnte die Grundstruktur für einen *dataTable* problemlos erstellt werden.
Nachdem die Anzeige mit Beispieldaten funktionierte, sollten die dynamischen Daten der Patienten aus dem bereits vorhandenen Backend und der Datenbank genutzt werden. Hierbei fiel auf, dass es nicht möglich war, das komplette Backend unverändert beizubehalten und nur fehlende Klassen und Beans zu ergänzen. Einige der *Arrays* mussten in Listen umgewandelt und viele Funktionen und deren Rückgabewerte angepasst werden.
Zudem wurde entschieden, im Zuge der Migration, gleich einige Modernisierungen des Backend-Bereichs zu vollziehen, um unter anderem *Generics* [37] einzuführen und *Arrays*, wenn möglich und sinnvoll, durch Listen zu ersetzen.

```
if (! Mandant.isConfigValid()) {
    Mandant.initObject();
    if (! Mandant.isConfigValid()) {
        throw new Exception("Invalid configuration or configuration file not found.");
    }
}

String loginTitle = null;
boolean useMandantSelectorPage = false;
Element globalProperties = Mandant.getGlobalConfigRoot().getChild("globalproperties");
if (globalProperties != null) {
    Element loginTitleElement = globalProperties.getChild("loginTitle");
    if (loginTitleElement != null) {
        loginTitle = loginTitleElement.getTextTrim();
    }

    Element useSelectorElement = globalProperties.getChild("usemandantselector");
    if (useSelectorElement != null) {
        useMandantSelectorPage = "true".equalsIgnoreCase(useSelectorElement.getTextTrim());
    }
}
else {
}
if ( (loginTitle == null) || (loginTitle.length() == 0)) {
    loginTitle = "CHILI/Telemedizinakte";
}

  String selectedClient = null;
  if (request.getParameter("mandant") != null) {
      selectedClient = request.getParameter("mandant");
  }
  else if (request.getParameter("selectedclient") != null) {
      selectedClient = request.getParameter("selectedclient");
  }

  User user = null;
  String errorMessage = null;
  MedizinakteObject medizinakteObject = null;
  String clientHost = request.getRemoteAddr();

  String username = null;
  String realusername = null;
  String password = null;
  String ticket = null;

  if ( request.getParameter("username") != null ) {
      username = request.getParameter("username").toLowerCase();
      if (request.getParameter("password") != null) {
          password = request.getParameter("password");
      }
      else if (request.getParameter("ticket") != null) {
          ticket = request.getParameter("ticket");
      }
  }

  //Datei geht noch weiter....
```

Abbildung 6.5: Kleiner Ausschnitt aus der Login-JSP-Seite

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:h="http://java.sun.com/jsf/html" xmlns:f="http://java.sun.com/jsf/core"
  xmlns:p="http://primefaces.org/ui">

<f:view>
  <h:head>
    <script type="text/javascript">
      if(top != self) top.location.replace(location);
    </script>
    <f:facet name="first">
      <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    </f:facet>

    <title>#{loginBean.loginTitle}</title>

    <link rel="stylesheet" type="text/css" href="medizinakte.css" />
    <link rel="shortcut icon" type="image/x-icon" href="image/akte_icon.ico" />
  </h:head>

  <h:body styleClass="login" onload="document.getElementById('fmLogin:username').focus();">
    <p:layout fullPage="true">
      <p:layoutUnit position="center">
        <h:form id="fmLogin">
          <ui:include src="#{loginBean.loginBody}" />
        </h:form>
      </p:layoutUnit>
    </p:layout>
  </h:body>
</f:view>


</html>
```

Abbildung 6.6: Neue Login-Seite in JSF

Eine wichtige Erkenntnis bei der Verwendung des vorhandenen Backends war, dass es wenig Sinn machte das bestehenden Backend so zu belassen. Es erwies sich als sinnvoll die Möglichkeit zu nutzen, das Backend gemeinsam mit der Modernisierung des Frontends genauer zu betrachten, um nicht mehr aktuelle Mechanismen zu ersetzen oder zu verbessern. Um die komplette Seite der Patientenanzeige mit den neuen Technologien darzustellen, waren weitere Umbaumaßnahmen der Funktionen und Datentypen notwendig, da die bisherigen Datenstrukturen nicht zu den geforderten Typen des PrimeFaces *datatable*s passten. Nach dem Umbau konnten die Patientendaten erfolgreich angezeigt werden.

6.7 Problematisches Neuladen der Seite

Nachdem ein Login und das Anzeigen einer ersten Seite möglich waren, sollte eine Struktur für das Menü erstellt werden, um die Ansicht zu wechseln. In der alten AKTE werden die Menüpunkte als eine Liste von Links dargestellt. Dieses Konzept wurde verworfen, da PrimeFaces mit seiner Menü-Komponente eine bessere Lösung bietet.

Da das Menü der bisherigen AKTE viele Punkte und Unterpunkte enthält und zudem noch konfigurierbar ist, wurde ein programmatisches Menü umgesetzt. Im Zuge dessen wurde die *MenuBean*, deren Lebensdauer die Session über bestehen sollte, erweitert. Im Konstruktor der *MenuBean* wird das *MenuModel*, welches in der Anzeige aufgerufen wird, erstellt und befüllt. Beim Befüllen des Menüs fiel die Aufmerksamkeit auf die Rechtekontrolle.
Die Anzeige der Menüpunkte wird im Backend, beim Erstellen des Menüs, gesteuert. Es wurde implementiert, dass vor dem Einfügen eines Menüpunktes in das *MenuModel* das entsprechende Recht abgeprüft und der Punkt bei fehlendem Recht nicht in das *MenuModel* übertragen wird. Die Navigation zu anderen Seiten erforderte eine allgemeine Rechteverwaltung, welche unter Punkt 6.8 behandelt wird.

Als die Navigation über das Menü möglich war, fiel ein unangenehmer Seiteneffekt auf: Da beim Klick auf einen Menüpunkt eine neue JSF-Seite aufgerufen wurde, und diese das TEMPLATE erneut einband, blinkte die komplette Ansicht durch das erneute Laden. Da dieser optische Effekt nicht erwünscht ist, wurde nach einer Lösung gesucht, um nur den Seiten-Content, welcher in Abbildung 6.7 zu sehen ist, zu erneuern.

Zunächst wurde ein ein festes TEMPLATE aufgerufen, welches seine Inhalte inkludiert. Für den Hauptinhalt wurde eine dynamisches *ui:include* innerhalb eines PANELS erstellt, welches dann über AJAX erneut geladen wurde. Diese Variante musste allerdings wieder verworfen werden, da ein dynamisches Erstellen eines INCLUDES mit den verwendeten Tomcat- und EL-Versionen nicht möglich war, was darauf zurückzuführen war, dass der Komponentenbaum schon erstellt wurde, als die Variable aus dem Backend noch nicht abgerufen werden konnte.

Nach einiger Recherchearbeit und dem Abgleich mit renommierten Herstellerseiten, welche PrimeFaces nutzen, unter anderem BMW [38], fiel auf, dass auch dort ein Blinken der ganzen Seite beim Navigieren zu verzeichnen ist. Da in der bisherigen AKTE eine Navigation ohne ein komplettes Neuladen der ganzen Seite durchgeführt wird und keine Verschlechterung erzielt werden darf, wurde die Möglichkeit einer iFrame-Nutzung erarbeitet. Die Umsetzung dieser und die Idee dahinter wird in Abschnitt 6.10 behandelt.

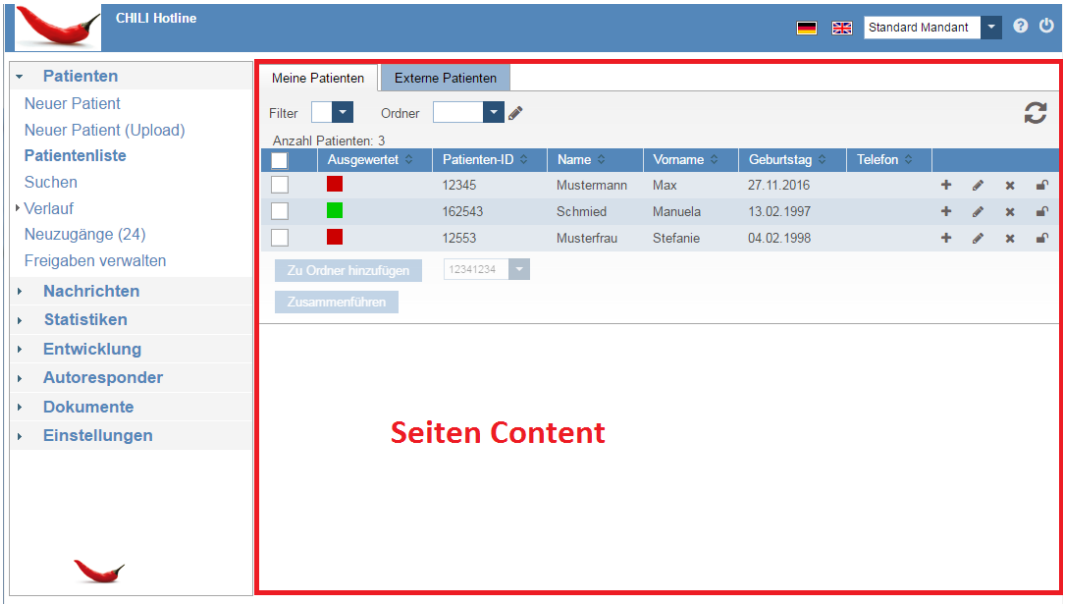

Abbildung 6.7: PrimeFaces Seite mit Menü und Content

6.8 Rechteüberprüfung

Bisher wird in jeder JSP-Datei im HEADER eingebunden mit welchen Rechten sie dem Benutzer angezeigt werden darf.
Dieser Mechanismus sollte ersetzt werden, da die Fehleranfälligkeit sehr hoch ist.

Es kann passieren, dass man das Einfügen des entsprechenden Rechts vergisst oder sich dabei verschreibt. Die Folgen wären verheerend, da es Nutzern möglich wäre, über einen direkten URL-Aufruf auf Seiten zu gelangen, die sie nicht sehen dürfen. Da es sich in der AKTE unter Anderem um sensible Patientendaten handelt, wäre ein solcher Fehler nicht vertretbar.

So entstand die Idee, eine Klasse mit einer MAP zu halten, in der verschiedene Ansichten an Rechte geknüpft sind. Beim Vergessen des Einfügens einer JSF-Seite in die MAP, wird die entsprechende Seite nicht angezeigt und man erhält eine Fehlermeldung. Somit wird einem unautorisierten Seiten-Aufruf durch Vergessen des Einfügens der Rechteabhängigkeit oder durch ein fehlendes Recht, vorgebeugt.

Dies wird in der Klasse *NavigationRightsGuard* gesteuert, welche speziell für diese Aufgabe erstellt wurde und die Session überdauert. In dieser Klasse ist es möglich, durch die Übergabe des NAVIGATION CASES oder dem Namen der JSF-Seite, zu erfahren, ob der momentan eingeloggte Benutzer die Rechte für die Anzeige der angeforderten Seite besitzt. Es kam die Frage auf, wann die Rechteüberprüfung stattfinden muss. Klar war, dass die Überprüfung durchgeführt werden soll, bevor die Seite angezeigt wird. Für diesen Zweck wurde ein PHASELISTENER implementiert, der in Abschnitt 6.9 beschrieben wird.

6.9 JSF-Lebenszyklus und PhaseListener

Der JSF-Lebenszyklus und seine Phasen ist an den einer HTTP-Anfrage gebunden, siehe Abbildung 6.8. Um festzustellen, in welcher Phase eine Rechteüberprüfung Sinn macht, müssen die einzelnen Phasen verstanden werden. Hier die Phasen im Detail:

- RESTORE VIEW (Ansicht wiederherstellen). Facelets baut aus dem zugrundeliegenden Dokument den Komponentenbaum auf und rendert ihn in einem zweiten Durchlauf. Die VIEWS werden in einer FACESCONTEXT-Instanz gespeichert.
- APPLY REQUEST VALUES (Request Parameter anwenden) Der Komponentenbaum wird bearbeitet und die eingetragenen Werte werden den einzelnen Komponenten zugewiesen. Bevor der Wert ins Datenmodell zurückgeschrieben wird, muss er validiert und konvertiert werden. Dies geschieht standardmäßig in der PROCESS VALIDATIONS-Phase. Setzt man das *immediate-*

Attribut auf *true*, kann dieser Schritt schon in die APPLY REQUEST VALUES-Phase vorgezogen werden.

- PROCESS VALIDATIONS (Konvertierung und Validierung durchführen). Der aus der vorherigen Phase ausgelesene Wert wird hier validiert und konvertiert, falls das *immediate* Attribut nicht auf *true* gesetzt wurde.
  Schlägt die Validierung in dieser Phase fehl, wird der Lebenszyklus angepasst und die Antwort wird bereits gerendert, ohne die anderen Phasen zu durchlaufen.
- UPDATE MODEL VALUES (Modell aktualisieren). In dieser Phase werden die Daten, falls Validierung und Konvertierung erfolgreich waren, über die SETTER in das Modell übertragen. Nach dieser Phase sind die entsprechenden Werte in die Bean überführt.
- INVOKE APPLICATION (Applikation ausführen). In dieser Phase werden Aktionen ausgeführt. Zum Beispiel *actions* oder *actionListener*, welche kurz vor den *actions* aufgerufen werden und somit die Navigation bestimmen.
- RENDER RESPONSE (Antwort rendern). Hier „wird der Komponentenbaum gerendert und die Ausgabe wird als Antwort der JSF-Anfrage zum Client geschickt“ [7, S. 46].
  Außerdem wird der Komponentenbaum für weitere Anfragen auf dieselbe Ansicht gespeichert.

[7, S. 41-47]

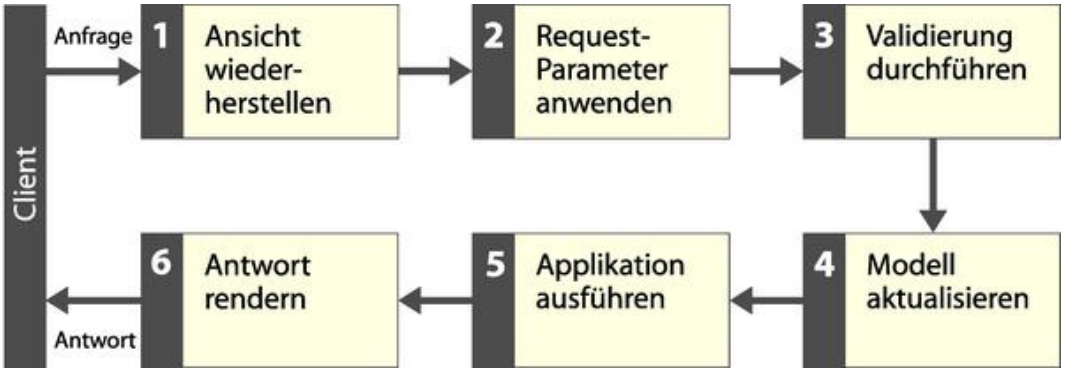

Abbildung 6.8: Der Lebenszyklus einer HTTP-Anfrage [7]

Die einzelnen JSF-Phasen bieten PHASE-EVENTS, die vor und nach jeder Phase ausgelöst werden. Dieser werden als *beforePhase* und *afterPhase* bezeichnet. Da jeder PhaseListener das Interface *javax.faces.event.PhaseListener* implementieren

muss, sind drei Methoden definiert, über die man die Aktionen in den jeweiligen Events steuern kann und die Phase, für die Ereignisse behandelt werden, als Konstante zurückbekommt. [7]

Diese sind

- *void afterPhase(PhaseEvent event)*
- *void beforePhase(PhaseEvent event)*
- *PhaseId getPhaseId()*

Beim genaueren Durchgehen der einzelnen Phasen fiel der Entschluss, die Rechteüberprüfung bereits vor der RESTORE VIEW-Phase durchzuführen, da dann keine unnötigen Komponentenbäume zusammen gebaut, oder bereits Werte validiert und konvertiert werden. Es wurde eine *RightsPhaseListener*-Klasse geschrieben, welche wiederum die *PhaseListener*-Klasse implementiert und in der RESTORE VIEW-Phase aufgerufen wird.
In diesem LISTENER wird nun in der Methode *beforePhase* von der RESTORE VIEW-Phase über den *NavigationRightsGuard* überprüft, ob der entsprechende Nutzer die Rechte für die angeforderte JSF-Seite besitzt. Besitzt er diese Rechte nicht, wird er ausgeloggt und mit einem Hinweis auf die Login-Seite weitergeleitet, oder er erhält eine Fehlermeldung.

6.10 Templating mit einem iFrame

Bei einem Menüaufruf wird die JSF-Datei mit dem gewünschten Seiteninhalt referenziert. Die JSF-Datei bindet das TEMPLATE ein, welches das Layout enthält. Das heißt, bei einem Wechsel der JSF-Seite wird das TEMPLATE und somit das ganze Menü neu eingebunden und es ist sichtbar, wie die ganze Seite neu lädt. Dies ist das erwartete Verhalten von PrimeFaces-Templating. Wie bereits erwähnt, ist dies auch bei anderen Nutzern der Fall.
Das dynamische Inkludieren ist nicht umsetzbar, da der Komponentenbaum dafür zu früh erstellt wird und die EL noch nicht interpretiert wurde. Es schien auf PrimeFaces-Ebene keine Alternative für das Auftreten des Neuladens der ganzen Seite zu geben, wenn man auf das Templating von PrimeFaces zurückgreifen wollte. Was gewünscht war, ist ein festes TEMPLATE mit einem dynamischen Seiteninhalt.

Diese Anforderung ließ sich am einfachsten durch ein iFrame umsetzen. Ein iFrame stellt einen eigenständigen Bereich mit eigener URL, hier innerhalb des TEMPLATES, dar. Da der Komponentenbaum nicht vom iFrame beeinflusst wird, da seine Inhalte losgelöst vom Rest der Seite existieren, war es nun möglich das iFrame über ein *ui:include* fest in das bestehende TEMPLATE einzubinden und die SOURCE dynamisch über eine Bean zu generieren, um so die verschiedenen Inhaltsseiten darzustellen, siehe Abbildung 6.9.

Die Seiteninhalte waren nun eigenständige Seiten und wurden auch als solche implementiert. Als nächstes wurde der Bereich des iFrames, welcher in ein PANEL integriert wurde, über AJAX neu geladen. Zuvor wurde die SOURCE, je nach Wunsch-Navigationsziel, angepasst. Eine Hauptinhalts-JSF-Seite wurde erstellt, welche das Menü, das iFrame und andere Seiteninhalte integriert und die Haupt- und Anzeigeseite darstellt. Ergänzend zu ihr wurde eine *ContentBean* eingeführt, die sich mit der kompletten Verwaltung des Hauptinhalts beschäftigt. Dazu gehört unter Anderem das Aktualisieren des *src*-Attributs für das iFrame und das Aktualisieren des PANELS, welches das iFrame beinhaltet.

Dieses Prinzip erschien zunächst praktisch und schlüssig. Bedenken bestanden allerdings, was die langfristige iFrame-Unterstützung der verschiedenen Browser angeht. Darüber sind keine genaueren Angaben zu finden und die Gefahr bestand, eine andere Lösung finden zu müssen, falls iFrames nicht mehr unterstützt werden. Zudem traten relativ schnell Komplikationen auf, was die Kommunikation des iFrames mit dem PARENT-WINDOW angeht. Das PARENT-WINDOW kennt seinen iFrame-Inhalt nicht und kann somit keinen direkten Einfluss darauf nehmen.
Eine Schwierigkeit war unter Anderem das richtige Hervorheben von Menüpunkten, wenn sich innerhalb des iFrames die Ansicht änderte. Dies wurde über die *MenuBean* mit Hilfe von JS umgesetzt. Somit musste die ganze Kommunikation des iFrames und seines PARENT-WINDOWS über Beans, Variablen und JS ablaufen. Dies war zum Teil schlecht wartbar und unnötig kompliziert.

Da diese Lösung nicht vollkommen zufriedenstellend war, sollte zu einem späteren Zeitpunkt erneut nach weiteren Möglichkeiten recherchiert werden.

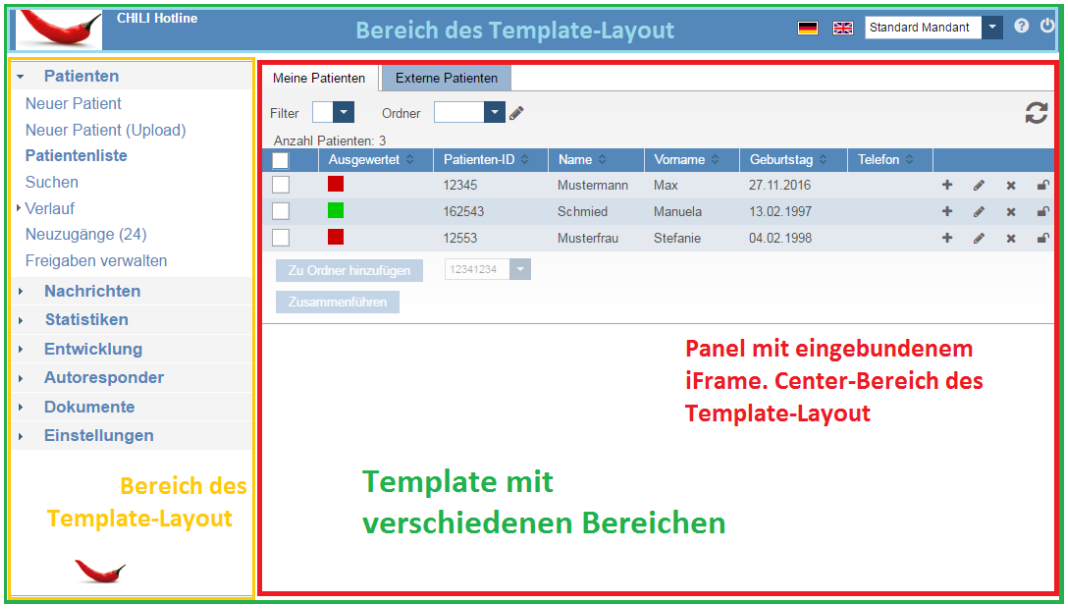

Abbildung 6.9: iFrame im Template-Bereich

6.11 Die Einführung von Faces Flows

Nachdem ein Routinevorgehen für das Erstellen von JSF-Dateien mit der Vorlage der vorhandenen JSPs erarbeitet war, konnte damit begonnen werden alle JSP-Dateien in JSF-Seiten zu migrieren. Beans, welche die Session über ihre Daten halten sollen, wurden in der faces-config als SESSION-SCOPED definiert und Beans, deren Daten nur für eine Ansicht von Bedeutung sind, als VIEW-SCOPED. Mit diesem Verfahren wurden die JSP-Seiten migriert, bis ein Fall auftrat, in welchem eine Bean über mehrere Ansichten hinweg bestehen sollte, jedoch auf keinen Fall die ganze Session über.

Dies trat beim Implementieren der Such-Funktion ein. Die sogenannte *SearchpatientsBean* dient unter anderem dazu, Patienten zu suchen oder Suchvorlagen für eine wiederholte Suche zu speichern. Um eine Suchvorlage zu speichern, gibt man zunächst die Daten in die Suchfelder ein, um dann über den Speicher-Button zur nächsten Seite und auch zur nächsten VIEW zu gelangen, in welcher man den Namen und die Sichtbarkeit der Vorlage bestimmen kann. In beiden VIEWS ist der Zugriff auf dieselben Daten notwendig und somit auch ein Bestehen der Bean über beide VIEWS hinweg.

Die Bean die Session über zu erhalten wäre wiederum unangemessen, da die Daten nach dem Verlassen der Suche nicht mehr benötigt werden.

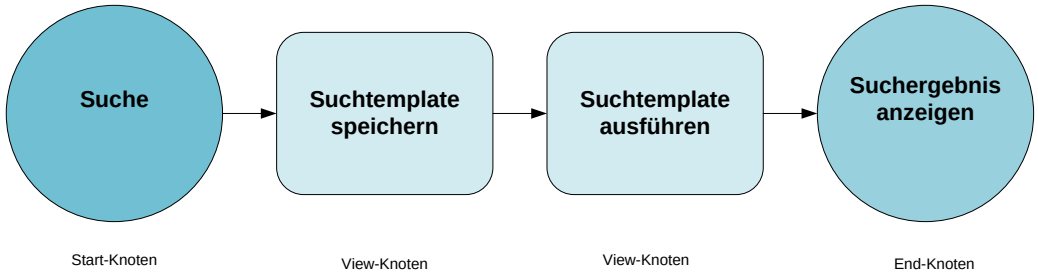

Abbildung 6.10: Beispiel eines Ablaufs im Flow searchform

Seit JSF 2.2 sind Faces-Flows für JSF spezifiziert. In Flows können zusammenhängende Seiten zu einer Gruppe zusammengefasst werden, sodass sie vom Einstiegspunkt ab in einer bestimmten Reihenfolge bis zum Ausstiegspunkt abgearbeitet werden können. Im Gesamten werden sie wie eine einzige View behandelt. So werden die Daten der Bean bis zum Ausstiegspunkt erhalten. Genau dieser Vorgang bietet sich bei der Suche an. In Abbildung 6.10 wird der mögliche Ablauf des Flows für die Suche dargestellt.

Bevor der Flow definiert und benutzt werden konnte, mussten einige Vorbereitungen getroffen werden.
Wie schon im Grundlagenkapitel erwähnt, muss CDI bei einem Servlet-Container (Abschnitt 2.5) wie Tomcat, zumindest wenn die Version kleiner ist als Tomcat 8, manuell integriert werden.
Der Grund hierfür ist, dass die JSP-Servlet-APIs von Tomcat 7 nicht den APIs von Java EE 7 entsprechen, sondern mit Java EE 6 kompilieren. Ab Tomcat Version 8 wäre eine manuelle Integration nicht mehr notwendig gewesen. [39]
Ein Update auf Tomcat 8 ist vorerst wegen der verwendeten Java-Version allerdings nicht möglich. Das Vorgehen zur Integration von CDI wird im Abschnitt 6.12 beschrieben.

Um einen Flow zu definieren wird eine XML-Datei mit dem Namen des Flows benötigt. In dieser werden Start-, View- und Endknoten des Flows definiert. Die zugehörige Bean wird über Annotationen an ihren Flow gebunden und kann so in den JSF-Dateien, genauso wie die anderen Beans, referenziert werden.
Um den Flow zu starten, reicht es nun, über die Flow-ID die JSF-Datei aufzurufen, welche als Startknoten definiert ist. Dies erreicht man zum Beispiel, indem man im *action*-Attribut die ID des Flows einträgt. Der Flow wird nun automatisch gestartet und man gelangt zur richtigen JSF-Datei. Beendet wird er, sobald man den Endkno-

```
<listener>
 <listener-class>
    org.apache.webbeans.servlet.WebBeansConfigurationListener
  </listener-class>
</listener>
```

Abbildung 6.11: Listener in der web.xml

ten erreicht. Wichtig ist, dass alle zum Flow gehörigen JSF-Dateien innerhalb eines Ordners mit dem Namen des Flows bereit liegen.

6.12 CDI-Integration

Zunächst war geplant, sich an das Vorgehen der Anleitung von www.jsf2.com [40] zu halten. Deswegen wurde JBoss Weld, eine Open-Source-CDI-Implementierung, heruntergeladen, die Jar integriert und die entsprechenden Definitionen in der context.xml und der web.xml eingetragen. Auch nach etlichen Versuchen konnten mit obiger Anleitung keine Erfolge erzielt werden und die ganze Anwendung nicht mehr gestartet werden. Nachdem sich auch an anderer Stelle keine Lösung auftat und Recherchen ergaben, dass es eventuell Kompatibilitätsprobleme mit der vorhandenen Tomcat-Version geben konnte, wurde JBoss Weld verworfen.

Da viele Entscheidungen am JavaServer Faces 2.2 Buch [7] orientiert getroffen wurden, fiel auch hier die Entscheidung nach dem Codebeispiel, welches mitgeliefert wird, vorzugehen. Aufgrund dieser Entscheidung wurde Apache OpenWebBeans (4.4) integriert. Zunächst wurde die aktuellste Version von OpenWebBeans, nämlich OWB-1.7.0 (Stand 16.09.2016), heruntergeladen und mit den benötigten Jar-Dateien integriert. Leider konnte damit keine funktionierende Einrichtung der CDI-Technologie erreicht werden und es gab diverse Probleme mit der Kompatibilität von einigen EL-Versionen und Jars. Trotz vieler Recherchen und dem Analysieren aller Log-Ausgaben konnten die exakten Problemquellen nicht identifiziert und gelöst werden. Es wurde auf OWB-1.2.0 gewechselt, da mit dieser Version ein Minimalbeispiel aus dem JSF-Buch [7] funktionierte und man mit den benötigten, zugehörigen Jars das CDI aktivieren konnte. Zusätzlich zu den Jars musste lediglich ein LISTENER, wie in Abbildung 6.11 zu sehen, in die web.xml eingetragen und eine leere beans.xml-Datei in den META-INF-Ordner gelegt werden. CDI konnte so erfolgreich integriert werden.

6.13 Tomcat-Update

Zunächst war kein Tomcat-Update vorgesehen. Beim Aufbau einer größeren Datentabelle, deren Werte im Frontend durch EL referenziert wurden, traten extreme Laufzeitprobleme auf und es dauerte signifikant lange bis die Datentabelle angezeigt wurde.

Die Analyse dieses Problems ergab, dass der CLASSLOADER bei jedem Aufruf durch EL erneut aufgerufen und bestimmte referenzierte Objekte nicht gepuffert wurden. Dies geschah aufgrund eines Tomcat-Bugs der bisher verwendeten Version [41].

Aus diesem Grund entschied man sich für ein Update auf die höchstmögliche Versionsnummer. Durch die vielen Abhängigkeiten des Tomcats war eine recht umfassende Recherche nötig, um die passende Versionsnummer zu finden. Die verschiedenen Abhängigkeiten ließen sich weitestgehend beim Kompilieren feststellen. Letztendlich erwies sich 7.0.69 als die höchstmögliche kompatible Version, in welcher der Performance-Bug behoben war.

Zusätzlich zur Geschwindigkeitsverbesserung der Anzeige, brachte das Update des Tomcats einen weiteren positiven Nebeneffekt mit sich. Nun war das Nutzen dynamischer Includes durch die ebenfalls angepasste EL-Version möglich. Dadurch konnte die bisherige iFrame-Lösung ersetzt werden, wie in Abschnitt 6.14 erläutert.

6.14 Dynamisches Inkludieren anstatt iFrame

Nachdem der Tomcat und mit ihm auch die genutzte Version der EL upgedatet wurden, war es möglich die Notlösung der iFrame-Nutzung zu verwefen.
Da die höhere Version des Tomcats und das Löschen einer älteren EL-Version nun ein dynamisches *ui:include* ermöglichen, konnte das vorhandene iFrame mit der zuvor geplanten Lösung aus Abschnitt 6.7 ersetzt werden.

Das iFrame wurde also entfernt und stattdessen ein festes TEMPLATE angezeigt, dessen Content bei einer Navigation über AJAX upgedatet wird. Die SOURCE des *ui:include*-tags wird dynamisch aus der *ContentBean* geholt und je nach Navigationsziel, vor dem Update angepasst.

Somit konnte eine saubere Lösung ohne iFrame nach JSF-Prinzip umgesetzt werden. Die Seiten-Kommunikation konnte so wieder vereinfacht werden und ein Aufblinken der Seite, war dennoch nicht zu sehen.

Kapitel 7

Resultat

Am Ende dieser Arbeit steht eine Version der neuen AKTE die mit JSF 2.2 und PrimeFaces lauffähig ist und den erarbeiteten Anforderungen genügt.
Das MVC-Konzept wurde umgesetzt und der übernommene Code überarbeitet und an aktuelle Technologien angepasst, obsoleter Code wurde entfernt. Außerdem wurden Bibliotheken ergänzt, welche unter anderem ein AJAX EXCEPTION HANDLING (Abschnitt 4.3) sowie das Nutzen von Flows ermöglichen.

Des Weiteren wurde ein neues Rechtemanagement sowie ein strukturiertes Templating für die Akte erarbeitet und umgesetzt. Ein dynamisches Menü und eine sortierbare Datentabelle mit DAG'N'DROP-Funktion wurden integriert.
Ein Großteil der in der bisherigen AKTE vorhandenen Funktionen wurden bereits mit den neuen Technologien implementiert.

Durch die schlechte Dokumentation von JSF, auf welche im Fazit (Kapitel 8) genauer eingegangen wird, und Wechselwirkungen, die einige PrimeFaces Komponenten mit sich bringen, könnten weitere Schwierigkeiten auftreten, auf die man im Laufe dieser Arbeit noch nicht gestoßen ist.

Die Grundlagen sind erarbeitet und die zu beachtenden Umstände bekannt. Die AKTE ist auf ihrem aktuellen Stand bereits als Grundlage für zwei Projekten im Einsatz.

Kapitel 8

Fazit

Das Ziel dieser Arbeit war es, das Konzept für die Migration verschiedener Web-Frameworks am Beispiel der CHILI/Telemedizinakte und PrimeFaces zu entwickeln und dieses dann umzusetzen. Bei der Recherche nach einem Migrations-Konzept konnte zügig ein Passendes herausgearbeitet werden, da sich die meisten Alternativen als nicht umsetzbar oder unwirtschaftlich erwiesen. Somit blieb eine Vollmigration auf einem separaten Entwicklungszweig als einzig praktikable Möglichkeit.

Im Allgemeinen ist zu sagen, dass die Wahl des Migrations-Konzeptes keine große Hürde darstellt, wenn Ziele, Technologien und Grundlagen klar definiert sind. Im Fall der AKTE erwies sich die Wahl der Vollmigration, unter den gegebenen Umständen, auch im Nachhinein als die Richtige.

Bei der praktischen Umsetzung der Migration ist vor allem aufgefallen, dass die teils fehlende und schlechte Dokumentation von JavaServer Faces und PrimeFaces einige Komplikationen mit sich bringt. Des Öfteren konnte keine logische Erklärung für bestimmte Verhaltensweisen des Frameworks gefunden werden und einiges, wie zum Beispiel das Umsetzen eines Flows, musste durch Prototyping etlicher Möglichkeiten herausgefunden werden. Selbst detaillierte Beschreibungen und Anleitungen aus Büchern führten teilweise zu keinem funktionierenden Ergebnis, da die Wechselwirkungen der verschiedenen Technologien und deren Versionen immens sind und in der Literatur nicht immer beachtet werden. Es war sehr schwer festzustellen wo die Kompatibilitätsprobleme lagen und ob es überhaupt an diesen lag, wenn ein Fehler auftrat. Die beste Vorgehensweise war, ein funktionierendes Beispiel mit ähnlichen Technologien und Abhängigkeiten zu finden, und dieses als Grundlage zu verwenden.

Direkt zu Beginn der Migration stellte sich heraus, dass der zeitlich geschätzte Aufwand für das Übertragen der JSP- in JSF-Dateien wesentlich unterschätzt wurde, da viele Funktionen neu überdacht und somit teilweise neu programmiert werden mussten. Ebenso erwies sich das Auslagern des Codes aus den JSP-Dateien ins Backend als eine zeitaufwändige Arbeit, da hierfür viele neue Klassen und Managed-Beans erforderlich waren.

Auch das zunächst einfach wirkende Konstrukt der verschiedenen SCOPES, auch Lebensdauer genannt, stellte eine große Herausforderung dar, welche ebenfalls auf eine nicht ausreichend vorhandene Dokumentation zurückzuführen ist. Einiges der Programmierarbeit musste doppelt gemacht werden, da in der Literatur verschiedene Dinge, wie zum Beispiel die wirkliche Lebensdauer der Managed-Beans im Backend nicht erwähnt werden. Neue Managed-Beans wurden zwar entsprechend ihres SCOPES erstellt, die vorhandenen allerdings nie verworfen, was zu schlechter Performance und einer Vielzahl an vorhandenen Managed-Beans führte.

Ebenso ist aufzuführen, dass die vielseitige Konfigurierbarkeit der AKTE eine hohe Komplexität der Migration mit sich bringt. Es existiert keine genaue Auflistung der verschiedenen Konfigurationsmöglichkeiten oder der vorhandenen Konfigurationen beim Kunden, sodass die Gefahr besteht, einige Funktionen zu vergessen, was erst im Produktiveinsatz beim Kunden auffallen würde. Ein umfangreiches Testen wird so erschwert.

Die größte Hürde der Migration erwies sich darin, den großen Versionssprung von JSF 1.1 zu JSF 2.2 umzusetzen. Da der Java-Code hinsichtlich der intern genutzten Datentypen und Rückgabewerte der Funktionen ebenfalls auf JSF 1.1 und JSP ausgelegt war, erwies es sich oft als aufwendig diesen umzubauen und es musste viel Backendprogrammierung vorgenommen werden, um alles auf JSF 2.2 zu migrieren.

Mit dem jetzigen Stand der AKTE und den im Zuge dieser Arbeit erlangten Erkenntnissen sollte es möglich sein, das Projekt erfolgreich abzuschließen.

Abkürzungsverzeichnis

AJAX	Asynchronous JavaScript and XML
AKTE	CHILI/Telemedizinakte
CDI	Contexts and Dependency Injection
CSS	Cascading Stylesheets
CVS	Concurrent Version System
DI	Dependency Injection
DKFZ	Deutschen Krebsforschungszentrum
DOM	Document Object Model: Spezifikation einer Schnittstelle für den Zugriff auf HTML oder XML Dokumente.
EL	Expression Language
HTML	HyperText Markup Language
JS	JavaScript
JSF	JavaServer Faces
JSP	JavaServer Pages
JSTL	JSP Standard Tag Library
Konsil	Der Begriff findet häufig im Krankenhaus Anwendung, wenn von einem Arzt einer anderen Fachrichtung eine Empfehlungen zur Diagnostik oder Therapie angefordert wird. Diese wird als Konsil bezeichnet. [42]
MVC	Model-View-Controller
PACS	Picture Archiving and Communication Systems: Ein PACS „ist in der Medizin ein Bildarchivierungs- und Kommunikationssystem auf der Basis digitaler Rechner und Netzwerke.“ [43]
Postgres	PostgreSQL
RI	Rechtfertigende Indikation: Eine rechtfertigende Indikation beschreibt warum eine Untersuchung oder Behandlung notwendig ist und wird vom Arzt oft dokumentiert.

SQL Structured Query Language

VDLGE View Declaration Language: „Eine Seitendeklarationssprache ist eine Syntax , um Ansichten beziehungsweise Seiten für JSF zu deklarieren.“ [7, S. 73]

XHR XMLHttpRequest: Programmierschnittstelle für JS um Daten über das HTTP-Protokoll zu übertragen.

XHTML Extensible HyperText Markup Language: XHTML ist eine Erweiterung von HTML. Im Gegensatz zu HTML genügen XHTML Dokumente den Syntaxregeln von XML.

XML Extensible Markup Language

XSLT Extensible Stylesheet Language Transformations: Sprache für die Transformation von XML Dokumenten in andere Formate.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] Wikipedia. Java (programmiersprache). https://de.wikipedia.org/w/index.php?title=Java_(Programmiersprache)&oldid=159935896, 2016. [Online; Zugriff 23. November 2016].

[2] Henry Gosling, James und McGilton. *The Java Language Environment, A White Paper*. Sun Microsystems Computer Complany, 1995.

[3] Program Creek. What is servlet container? http://www.programcreek.com/2013/04/what-is-servlet-container/, 2013. [Online; Zugriff 16.September 2016].

[4] Christian Ullenboom. Java ist auch eine insel. http://openbook.rheinwerk-verlag.de/javainsel/, 2011. [Online; Zugriff 27. September 2016].

[5] ITWissen.info. Servlet. http://www.itwissen.info/definition/lexikon/Servlet.html, 2016. [Online; Zugriff 26. September 2016].

[6] Hans Bergsten. *JavaServer Pages*. O'REILLY, 3. auflage edition, 2004.

[7] Michael Kurz and Martin Marinschek. *JavaServerFaces 2.2 - Grundlagen und erweiterte Konzepte*. dpunkt.verlag, 3. auflage edition, 2014.

[8] Jonas Jacobi and John R. Fallows. *Pro JSF and Ajax*. Apress, 2006.

[9] ORACLE. What is facelets? https://docs.oracle.com/cd/E19798-01/821-1841/giepx/index.html, o.J. [Online; Zugriff 22. August 2016].

[10] Christian Wenz. *JavaScript*. Rheinwerk Verlag GmbH, 2001-2002.

[11] Wikipedia. Javascript. https://de.wikipedia.org/w/index.php?title=JavaScript&oldid=158080824, 2016. [Online; Zugriff 19. Oktober 2016].

[12] Wikipedia. Ajax (programmierung). https://de.wikipedia.org/w/index.php?title=Ajax_(Programmierung)&oldid=157225267, 2016. [Online; Zugriff 21. Oktober 2016].

[13] Jesse James Garrett. Ajax: A new approach to web applications. https://web.archive.org/web/20080702075113/http://www.adaptivepath.com/ideas/essays/archives/000385.php, 2005. [Online; Zugriff 21. Oktober 2016].

[14] Microsoft Corporation. Tag libraries. https://msdn.microsoft.com/en-us/library/aa478990.aspx, 2003. [Online; Zugriff 19. Oktober 2016].

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Entwurfsmuster*. ADDISON-WESLEY, 1. auflage edition, 1996.

[16] Wikipedia. Dependency injection. https://de.wikipedia.org/w/index.php?title=Dependency_Injection&oldid=155878979, 2016. [Online; Zugriff 23. August 2016].

[17] Wikipedia. Contexts and dependency injection. https://de.wikipedia.org/w/index.php?title=Contexts_and_Dependency_Injection&oldid=154634280, 2016. [Online; Zugriff 23. August 2016].

[18] Andy Gibson. Contexts and dependency injection. https://netbeans.org/kb/docs/javaee/cdi-intro.html, o.J. [Online; Zugriff 14. November 2016].

[19] MarkStruberg. Tutorial – introduction to cdi – contexts and dependency injection for java ee (jsr 299). https://jaxenter.com/tutorial-introduction-to-cdi-contexts-and-dependency-injection-for-java-ee-jsr-299-104536.html, 2012. [Online; Zugriff 14. November 2016].

[20] PostgreSQL. About. https://www.postgresql.org/about/, 2016. [Online; Zugriff 26. September 2016].

[21] The Apache Software Foundation. Core jsf-1.1. https://myfaces.apache.org/core11/index.html, o.J. [Online; Zugriff 24. Oktober 2016].

[22] The Apache Software Foundation. Myfaces core 1.1. http://myfaces.apache.org/core11/index.html, o.J. [Online; Zugriff 17. August 2016].

[23] David Geary and Cay Horstmann. *core - JavaServer Faces*. Sun Microsystems Press, 4. auflage edition, 2004.

[24] Wikipedia. Apache myfaces. https://de.wikipedia.org/w/index.php?title=Apache_MyFaces&oldid=137888418, 2015. [Online; Zugriff 17. August 2016].

[25] Thomas Asel. *Auswirkung großer Komponentenbäume auf die Performance von JSF Implementierungen*. Orientation in Objects, 2012.

[26] The Apache Software Foundation. Tag library information. https://myfaces.apache.org/tomahawk-project/tomahawk/tagdoc.html, o.J. [Online; Zugriff 24. Oktober 2016].

[27] PrimeTek Informatics. Ready for primetime. http://www.primetek.com.tr/, 2015. [Online; Zugriff 30. August 2016].

[28] redhat. Richfaces, the next-generation jsf component framework by jboss! http://richfaces.jboss.org/, 2016. [Online; Zugriff 30. August 2016].

[29] devrates. Primefaces. http://devrates.com/project/show/89675/PrimeFaces, o.J. [Online; Zugriff 30. August 2016].

[30] Grails. Devrates. https://grails.org/website/69, o.J. [Online; Zugriff 25. Oktober 2016].

[31] devrates. Top 10 technologies by category. http://devrates.com/stats/top?tagName=web+framework, o.J. [Online; Zugriff 30. August 2016].

[32] OmniFaces. What is omnifaces? http://showcase.omnifaces.org/, o.J. [Online; Zugriff 30. August 2016].

[33] Claudia Fröhling. Apache openwebbeans 1.2.2 wird schneller. https://jaxenter.de/apache-openwebbeans-1-2-2-wird-schneller-1457, 2014. [Online; Zugriff 30. August 2016].

[34] Tobias Lönnies. Ein kurzer Überblick über scopes in java ee7. http://blog.triona.de/development/jee/ein-kurzer-uberblick-uber-scopes-in-java-ee7.html, 2014. [Online; Zugriff 18. November 2016].

[35] BalusC. How to choose the right bean scope? - introduction. http://stackoverflow.com/questions/7031885/how-to-choose-the-right-bean-scope, 2011. [Online; Zugriff 18. November 2016].

[36] Michael Kurz and Martin Marinschek. Javaserverfaces 2.2 - grundlagen und erweiterte konzepte. http://jsfatwork.irian.at/book_de/introduction.html, o.J. [Online; Zugriff 31. August 2016].

[37] ORACLE. Generic types. https://docs.oracle.com/javase/tutorial/java/generics/types.html, o.J. [Online; Zugriff 06.September 2016].

[38] PrimeFaces. Who uses primefaces. http://www.primefaces.org/whouses, o.J. [Online; Zugriff 06.September 2016].

[39] stackoverflow. How to install and use cdi on tomcat? http://stackoverflow.com/questions/18995951/how-to-install-and-use-cdi-on-tomcat, 2013. [Online; Zugriff 29. Oktober 2016].

[40] coreservlets.com. Configuring tomcat 7 and 8 to use cdi. http://www.jsf2.com/using-cdi-and-jsf-2.2-faces-flow-in-tomcat/, 2016. [Zugriff 16. September 2016].

[41] Apache Konstantin Kolinko. Bug 52998 - performance issue with expressionfactory.newinstance(). https://bz.apache.org/bugzilla/show_bug.cgi?id=52998, 2012. [Online; Zugriff 18. November 2016].

[42] Wikipedia. Konsil. https://de.wikipedia.org/w/index.php?title=Konsil&oldid=153357757, 2016. [Online; Zugriff 19. August 2016].

[43] Wikipedia. Picture archiving and communication system. https://de.wikipedia.org/w/index.php?title=Picture_Archiving_and_Communication_System&oldid=153700264, 2016. [Online; Zugriff 22. September 2016].