Berufsakademie Sachsen

Staatliche Studienakademie Leipzig

**Analyse des Einsatzes einer Volltextsuche für die Sage HR Suite am Beispiel von Apache Solr**

Bachelorarbeit

zur Erlangung des akademischen Grades eines

„Bachelor of Science“

in der Studienrichtung Informatik

Eingereicht von: Georg Andrássy

Herrmann-Meyer-Straße 11, 04207 Leipzig

Seminargruppe: CS14-1

Matrikelnr.: 5000593

Betreuer: M. o. Science Felix Franck

Sage GmbH

Karl-Heine-Str. 109-111

04229 Leipzig

Leipzig, 29.08.2017

**Inhaltsverzeichnis**

[1 Einleitung 5](#_Toc491759164)

[1.1 Motivation 5](#_Toc491759165)

[1.2 Einordnung der Volltextsuche 7](#_Toc491759166)

[1.3 Einführung in die Sage GmbH und Sage HR Suite 8](#_Toc491759167)

[1.4 Anforderungen an einen Prototyp 9](#_Toc491759168)

[1.5 Thesen 10](#_Toc491759169)

[2 Theoretische Vorbetrachtung 11](#_Toc491759170)

[2.1 Apache Lucene 11](#_Toc491759171)

[2.1.1 Aufbau der Indexdatenbank 12](#_Toc491759172)

[2.1.2 Durchsuchen der Indexdatenbank 15](#_Toc491759173)

[2.1.3 Sortierung der Suchergebnisse 16](#_Toc491759174)

[2.2 Apache Solr 17](#_Toc491759175)

[2.2.1 Möglichkeiten von Solr 17](#_Toc491759176)

[2.2.2 Grenzen von Solr 19](#_Toc491759177)

[2.2.3 Werkzeuge zur Indexierung 19](#_Toc491759178)

[3 Vorüberlegungen zur Umsetzung in der Sage HR Suite 22](#_Toc491759179)

[3.1 Software-Architektur vom HR Portal 22](#_Toc491759180)

[3.2 REST-konforme Web-API als Schnittstelle zur Datenquelle der PW 26](#_Toc491759181)

[3.3 Navigation innerhalb des HR Portals 27](#_Toc491759182)

[3.4 Benutzerberechtigungen 30](#_Toc491759183)

[3.5 Erzeugung eines möglichst eindeutigen Primärschlüssels für Dokumente 33](#_Toc491759184)

[3.6 Überlegungen zur partiellen Indexierung 34](#_Toc491759185)

[4 Implementierung der Volltextsuche 36](#_Toc491759186)

[4.1 Technische Infrastruktur 36](#_Toc491759187)

[4.2 Indexierung der Daten 38](#_Toc491759188)

[4.2.1 Auswahl der zu indexierenden Daten 38](#_Toc491759189)

[4.2.2 Definition einer eigenen Datenquelle für die WebAPI 40](#_Toc491759190)

[4.2.3 Konfiguration des Data Import Handler 41](#_Toc491759191)

[4.2.4 WebAPI-Schnittstelle für Datenanfrage zur Indexierung 44](#_Toc491759192)

[4.3 Benutzereingabe zur Volltextsuche 46](#_Toc491759193)

[4.4 Auswahl und Darstellung der Suchergebnisse 48](#_Toc491759194)

[4.5 Navigation vom Suchergebnis zur entsprechenden Ansicht 50](#_Toc491759195)

[5 Diskussion der Thesen 54](#_Toc491759196)

[5.1 Legitimierung der Volltextsuche in volltextarmen Anwendungen 54](#_Toc491759197)

[5.2 Eignung der Volltextsuche für Navigation im HR Portal 55](#_Toc491759198)

[5.3 Zentrale Steuerung der Navigation 56](#_Toc491759199)

[5.4 HR Portal als Single-Page-Anwendung 57](#_Toc491759200)

[5.5 Verwendung des DIH zum automatisierten Indexieren 58](#_Toc491759201)

[6 Zusammenfassung, Fazit und Ausblick 60](#_Toc491759202)

[6.1 Zusammenfassung 60](#_Toc491759203)

[6.2 Fazit 62](#_Toc491759204)

[6.3 Ausblick 63](#_Toc491759205)

[6.3.1 Automatisierte Schlagwortgewinnung für Indexierung 63](#_Toc491759206)

[6.3.2 Fehlerhafte Sucheingaben und Autovervollständigung 63](#_Toc491759207)

[6.3.3 Übersichtlichere Darstellung der Suchergebnisse 64](#_Toc491759208)

[7 Abkürzungsverzeichnis 65](#_Toc491759209)

[8 Abbildungsverzeichnis 66](#_Toc491759210)

[9 Tabellenverzeichnis 67](#_Toc491759211)

[10 Literaturverzeichnis 68](#_Toc491759212)

[11 Anhang 72](#_Toc491759213)

[11.1 Auszug aus Solr-Schema-Datei für Prototyp 72](#_Toc491759214)

[11.2 Programmcode der beiden DataSource-Klassen 72](#_Toc491759215)

[11.3 Konfigurationsdatei des Data Import Handler für Prototyp 72](#_Toc491759216)

[11.4 Quelltext zur Benutzeroberfläche der Volltextsuche 72](#_Toc491759217)

[11.5 Vollständige Ansicht der Suchergebnisse 73](#_Toc491759218)

[11.5.1 Beispiel für Kalenderübersichtmodul 73](#_Toc491759219)

[11.5.2 Beispiel für Bewerberdokumente 74](#_Toc491759220)

[11.6 Vollständiger Quelltext innerhalb der WebAPI zum Absenden einer Solr-Suchanfrage 74](#_Toc491759221)

[11.7 Kalendermodulansichten 75](#_Toc491759222)

[11.7.1 Hauptansicht 75](#_Toc491759223)

[11.7.2 Ereignisliste 75](#_Toc491759224)

[11.7.3 Ereignisdetails 76](#_Toc491759225)

[11.8 Quelltext zur Navigation innerhalb der Kalenderübersicht 76](#_Toc491759226)

[11.9 Gemessene Leistungsdaten während der Indexierung und der Suche 76](#_Toc491759227)

[Selbstständigkeitserklärung 78](#_Toc491759228)

# Einleitung

## Motivation

Wenn es darum geht, sich als Softwareunternehmen mit einem neuen Produkt gegen seine Konkurrenz durchzusetzen, fällt häufig neben dem reinen Funktionsumfang des Computerprogramms, im Folgenden auch als Anwendung bezeichnet, die Eigenschaft *Benutzerfreundlichkeit* als gewinnbringendes Verkaufsargument auf. Potentielle Kunden werden immer anspruchsvoller und wählerischer, wollen sie keine seitenlangen Benutzeranleitungen mehr studieren, um zum Ziel zu gelangen. Dazu zählt auch, dass gewünschte Informationen schnell, sprich in wenigen Handlungsschritten, erreichbar sind. Je komplexer jedoch eine Anwendung ist, desto schwieriger wird es für dessen Entwickler, die vielen Funktionen möglichst intuitiv bereit zu stellen. Eine natürlich sprachige Volltextsuche verspricht, dieses Problem aufzulösen, in der Hoffnung, dass der Anwender selbst dann die richtige Antwort erhält, wenn er noch nicht einmal die genaue Frage kennt.

Die weltweit populärste und meistgenutzte Volltextsuche stellt das Unternehmen *Google* mit seiner gleichnamigen Web-Anwendung[[1]](#footnote-1) bereit [1]. Diese ermöglicht es, zu beliebigen verschieden sprachigen Suchwörtern die zugehörigen, relevantesten Webseiten vorzuschlagen. Die Sucheingabe ist tolerant gegenüber Schreibfehlern und die Rangfolge der Ergebnisse richtet sich stark nach deren Passgenauigkeit zur Eingabe.

Auch eine rein auf die Verwaltung von personalbezogenen Daten fokussierte Software, wie die *Sage HR Suite*, kann womöglich von einer solchen Suchfunktion profitieren. Schließlich handelt es sich bei dem Produkt um eine vielschichtige und äußerst komplexe Anwendung, die ohne eine entsprechende Schulung im Bereich Personalwirtschaft kaum adäquat zu bedienen ist. Was aber wäre, wenn der Benutzer das Programm nach speziellen Informationen *fragen* könnte? Vielleicht möchte er auch eine besonders schwer auffindbare Funktion verwenden und will dazu aber nicht jede Ansicht mühsam aufrufen und durchsuchen. Oder er weiß, wo sich die Funktion befindet, muss dazu aber sehr viele Schaltflächen bedienen, um dorthin zu gelangen. Es wäre möglich, dass eine Volltextsuche hier für Abhilfe sorgen kann. Dies gilt es im Rahmen der vorliegenden Arbeit herauszufinden.

Ziel dieser Arbeit ist es, eine Möglichkeit aufzuzeigen, wie Daten sowie spezielle Funktionsbereiche innerhalb der Sage HR Suite mit Hilfe des Einsatzes einer *natürlichsprachigen*[[2]](#footnote-2) Volltextsuche vereinfacht zugänglich gemacht werden können. Als Ergebnis ist ein Prototyp innerhalb eines ausgewählten Bereichs zu implementieren, der die Machbarkeit der Vollintegration einer solchen Suche beweist. Warum die Wahl des dafür verwendeten Moduls der Sage HR Suite auf einen Bereich innerhalb der zugehörigen Web-Anwendung *Sage HR Portal,* bzw. nur *HR Portal,* fällt, erläutert Punkt 1.3. Dort wird auch auf das Produkt Sage HR Suite näher eingegangen. Es ist außerdem die Frage zu klären, warum eine Volltextsuche sinnvoll ist und welche Alternativen sich darüber hinaus anbieten. Den Abschluss von Kapitel 1 bilden die aufgestellten Thesen, die es im Rahmen dieser Arbeit zu überprüfen gilt.

Die Software zur Volltextsuche selbst zu entwickeln, ist wenig sinnvoll, da bereits eine frei zugängliche Java-Programmbibliothek unter dem Namen *Apache Lucene* erhältlich ist, die sämtliche Basisfunktionen erfüllt und über die Erweiterung *Apache Solr,* kurz als Solr bezeichnet*,* eine benutzerfreundliche Schnittstelle im Web-Browser bietet. Solr ist aber nicht die einzige auf Lucene aufsetzende Technologie, denn mit *ElasticSearch* gibt es eine weitere Open-Source[[3]](#footnote-3) Variante, die Solr in nichts nachsteht. Beide unterscheiden sich in ihrem Funktionsumfang kaum, jedoch ist ElasticSearch mehr auf sich ständig verändernde Daten ausgerichtet, die innerhalb eines Netzwerks verteilt vorliegen. Solr hingegen liefert schnellere Ergebnisse bei eher statischen bzw. wenig veränderlichen, vor allem auf Text basierenden, Daten [2]. Da es keinen klaren Gewinner zwischen beiden Volltextsuchanwendungen gibt und eine Entscheidung getroffen werden muss, fällt diese auf das ein wenig besser dokumentierte Solr [3].

In Kapitel 2.1 wird der Technologiekern von Apache Lucene genauer unter die Lupe genommen und seine Hauptkonzepte zur Indexierung von Daten, zum Durchsuchen der indexierten Daten sowie zur Filterung und Sortierung der Suchergebnisse näher betrachtet. Im daran anschließenden Punkt 2.2 sind die wichtigsten Funktionalitäten von Solr aufgeführt, welche für die Umsetzung in der PW besonders nützlich sind. Außerdem ist es notwendig auf für Solr ungeeignete Anwendungsfälle hinzuweisen. Abschnitt 2.2 schließt dann mit einer genaueren Betrachtung der von Solr zur Verfügung gestellten Werkzeuge zur Indexierung von Daten.

Kapitel 3 beinhaltet die theoretische Vorbetrachtung zur Implementierung der Volltextsuche mit Hilfe von Apache Solr innerhalb des HR Portals. Die praktische Umsetzung in einem eingeschränkten Bereich ist dann in Kapitel 4 beschrieben. Eine Diskussion der aufgestellten Thesen und die Bewertung der Machbarkeit einer vollumfänglichen Umsetzung der Volltextsuche liefert Kapitel 5, während Kapitel 6 die Arbeit zusammenfasst, ein Fazit zieht und einen Ausblick auf mögliche Optimierungen gibt.

## Einordnung der Volltextsuche

Wie der Name bereits vermuten lässt, handelt es sich bei einer Volltextsuche primär um das Durchsuchen von vollständigen Texten bzw. genauer von unstrukturiert und digitalisiert vorliegenden Textdaten. Das Hauptziel stellt die Informationsgewinnung dar [4 S. 6]. Dabei wird gezielt nach Wörtern gesucht, welche eine entsprechende Suchmaschine innerhalb von zum Teil sehr großen Datenmengen, jenseits von TeraByte, findet und die zugehörigen Textdokumente zusammen mit den Stellen, an denen die Wörter vorkommen, liefert.

Die Suchmaschine Google bietet beispielsweise die Möglichkeit, eine riesige Bücherdatenbank nach Wörtern innerhalb der Bücher zu durchsuchen. Die Stärke der Volltextsuche zeigt sich an dieser Stelle, indem innerhalb von Bruchteilen einer Sekunde sämtliche Bücher zurückgegeben werden, die die Suchbegriffe enthalten. Dank der in Abschnitt 2.1 erläuterten Methoden der Suche ist es möglich, eine solche Datengrundlage so vorzubereiten, dass Suchergebnisse in kurzer Zeit lieferbar sind.

Obwohl die Reihenfolge der Ergebnisse ihre Relevanz für das gesuchte Thema sehr gut widerspiegelt, darf nicht davon ausgegangen werden, dass es sich bei der Volltextsuche um eine *intelligente* Technologie handelt. Dahinter steckt ein Algorithmus, der die Häufigkeit des Auftretens von Suchbegriffen zusammen mit verschiedenen Gewichtungen und ihrem räumlichen Abstand im Text berücksichtigt. Mehr dazu ist in Abschnitt 2.1.3 erläutert. Das Programm *kennt* weder Semantik bzw. Bedeutung von eingegebenen Wörtern und Sätzen, noch *versteht* es die gespeicherten Texte und kann sie so der Suchanfrage zuordnen.

Die Google-Suchanwendung ist ein Beispiel für eine Volltextsuche, die bereitgestellte Informationen von öffentlich zugänglichen Webseiten gezielter und schneller finden lässt. Es gibt aber auch Volltextsuchen, die nur auf eine Anwendung und deren Daten spezialisiert sind. Dabei geht es weniger darum, einen möglichst intelligenten Software-Assistenten bereit zu stellen, der bei der Bedienung und beim Verständnis der Anwendung helfen kann. Vielmehr erhält der Anwender die Möglichkeit, ohne große Umwege zu der Stelle im Programm zu gelangen, an der entweder bestimmte Daten zu finden sind oder von wo aus die gewünschte Funktion ausführbar ist.

Bisher war nur von unstrukturierten Volltextdaten die Rede. Es können jedoch auch strukturierte Daten, die z.B. in relationalen Datenbanken vorliegen, durchsucht werden. Im Gegensatz zu den erstgenannten besitzen Daten in einer Datenbank ein Schema, das vorgibt, welchem Datentyp sie angehören. Wenn es nur um das Auffinden von Daten geht, so kann eine reine Datenbankabfrage, z.B. mit Hilfe von SQL, ähnlich gute Ergebnisse wie eine Volltextsuche liefern. Jedoch scheitert sie an fehlerhaften Eingaben oder an ihrer Forderung nach Vollständigkeit, sodass Datensätze, die nur teilweise zu einer Suchanfrage passen, nicht gefunden werden [5].

## Einführung in die Sage GmbH und Sage HR Suite

Sage ist ein weltweites Softwareunternehmen mit seinem Hauptsitz in Newcastle in England. In 23 Ländern werden integrierte Lösungen für Buchhaltungs-, Lohnabrechnungs und Bezahlsysteme angeboten. Der deutsche Vertreter Sage GmbH ist mit ca. 250.000 Kunden im deutschen Mittelstand einer der Marktführer für betriebswirtschaftliche Software und erwirtschaftete im Jahr 2014/2015 über 100 Millionen Euro Umsatz [6].

Die Leipziger Niederlassung ist für den Geschäftsbereich *Personalwirtschaft* verantwortlich und entwickelt seit Ende der Neunziger Jahre das zugehörige Hauptprodukt die Sage HR Suite, im Folgenden mit *PW* abgekürzt. Dieses deckt nahezu sämtliche Belange eines Unternehmens bezüglich der Verwaltung von Personaldaten ab. Die wichtigsten Themen innerhalb der PW sind die Personalabrechnung, welche sämtliche Gehälter berechnet, die Zeitwirtschaft, die u.a. die Zeiterfassung von Personal und Projekten ermöglicht, aber auch das Bewerbermanagement und die Personalentwicklung werden u.a. berücksichtigt.

Sämtliche Themen der PW sind innerhalb einer sogenannten Desktop-Anwendung[[4]](#footnote-4) nutzbar. Jedoch gibt es seit ein paar Jahren auch eine zugehörige Web-Anwendung, das HR Portal, welches im Mai 2017 in einer neu überarbeiteten Version erschien. Das HR Portal wird vom Web-Browser ausgeführt und benötigt somit keine lokale Installation auf dem Computer des Benutzers. Es dient als Ergänzung zur PW und soll einen übersichtlichen und schnellen Zugriff auf Personaldaten ermöglichen. Außerdem bietet es neue Funktionen, wie z.B. den modulübergreifenden Kalender, in dem sämtliche arbeitsrelevanten Ereignisse auf einem Blick ersichtlich sind.

In Zukunft könnte das HR Portal weiter ausgebaut werden, mit dem Ziel die PW als gleichwertige Web-Anwendung abzulösen, wodurch eine Installation hinfällig wäre und ein moderneres Preissystem nach dem Subskriptionsverfahren[[5]](#footnote-5) das bisherige Jahreslizenzsystem[[6]](#footnote-6) mit zum Teil teurem Wartungsservice ablösen kann. Dann hat die Hauptzielgruppe der Klein- und mittelständischen Unternehmen es noch leichter, testweise die PW auszuprobieren. Die damit verbundenen neuen Verkaufsmöglichkeiten führen dazu, dass das HR Portal verstärkt weiter entwickelt wird. Aufgrund der vielversprechenden Zukunftsperspektive eignet es sich innerhalb der PW besonders gut als Plattform für die Volltextsuche.

## Anforderungen an einen Prototyp

Um zu zeigen, dass eine Volltextsuche für die PW technisch machbar ist, sei ein Anwendungsprototyp für ein ausgewähltes Modul aus dem HR Portal der PW zu implementieren. Zum einen stellt das Durchsuchen von unstrukturierten Daten, wie Bewerbungsunterlagen oder freie Notizen eine klassische Aufgabe für eine Volltextsuche dar. Zum anderen kann anhand von themenbezogenen Schlagwörtern über die Suchfunktion das Finden von Daten in entsprechenden Benutzeroberflächen beschleunigt werden, sodass eine intuitivere Bedienung möglich wird.

Daher konzentriert sich die Umsetzung eines Prototyps auf zwei Dinge. Zum einen ist es das Ziel, im Bewerbermodul des HR Portals Bewerbungsunterlagen als Volltextdokumente beliebig durchsuchbar werden zu lassen. Es ist gefordert, dass im sämtliche Bewerbungsdokumente von allen Bewerbern erfasst werden. Über eine Weiterleitung muss es dann möglich sein, dass vom Suchergebnis zur Detailansicht des zugehörigen Bewerbers gewechselt werden kann, sodass dort das gefundene Bewerbungsdokument vollständig aufrufbar ist.

Zum anderen werden mitarbeiterabhängige Ansichten zum Resturlaub sowie Erfassungsformulare für Fehlzeiten über die Volltextsuche direkt aufrufbar, sodass innerhalb eines Schritts dorthin navigiert werden kann. Beispielsweise hat man als Vorgesetzter die Möglichkeit, sämtliche Abwesenheitszeiten seiner Angestellten in einer Kalenderansicht für ein gewisses Zeitintervall auf einem Blick zu sehen. Pro Tag können verschiedene Ereignisse, wie z.B. Urlaub, Krankheit, Weiterbildung, eingetragen werden, sodass nachvollziehbar ist, welcher Mitarbeiter, wann zur Verfügung steht. Es muss möglich sein, mit Hilfe der Volltextsuche zu einzelnen Ansichten der Ereigniserfassung zu gelangen. Außerdem bietet die Kalenderübersicht für jeden Mitarbeiter eine ausklappbare Detailanzeige zu seinem Resturlaub und seinen Aufgaben. Auch diese Ansichten sind vom Prototyp zu berücksichtigen.

Darüber hinaus ist ein Konzept zu entwickeln, nach dem das komplexe Benutzerrollensystem innerhalb des HR Portals von der Volltextsuche berücksichtigt wird, sodass Anwender nur zu den Daten über die Suchfunktion gelangen können, für die sie auch die notwendigen Zugangsrechte haben. Eine Umsetzung des Konzepts ist aus Zeitgründen nicht im Rahmen dieser Arbeit vorgesehen.

## Thesen

Folgende Thesen gilt es in der Analyse näher zu untersuchen.

These 1: Auch ohne Volltextdaten ist eine Volltextsuche sinnvoll zum Durchsuchen von Anwendungen wie dem HR Portal.

These 2: Die Volltextsuche eignet sich gut zur vereinfachten Navigation innerhalb des HR Portal.

These 3: Eine zentral gesteuerte Navigation zwischen verschiedenen Ansichten eignet sich gut zur Weiterleitung von Suchergebnissen.

These 4: Erst der Umbau des HR Portals in eine reine *Single-Page-Anwendung* ermöglicht die modulübergreifende Volltextsuche.

These 5: Zum automatisierten Indexieren von Daten der Sage HR Suite ist der *Data Import Handler* von Solr die beste Wahl.

# Theoretische Vorbetrachtung

Bevor die aufgestellten Thesen mit Hilfe eines Prototyps überprüft werden können, gilt es zunächst wichtige theoretische Grundkonzepte der Volltextsuche zu klären, um zu verstehen, was sie leisten kann und was nicht. Da die Entscheidung, welche Technologie verwendet wird, auf Apache Solr gefallen ist, kann als Erstes der eigentliche Kern von Solr – *Apache Lucene* – und seine Grundmechanismen näher beleuchtet werden.

## Apache Lucene

Lucene ist ein Open-Source Projekt der *Apache Software Foundation* und entsprechend unter der Apache Software-Lizenz lizensiert. Es kann als quelloffene Programmbibliothek in der eigenen Anwendung für die Volltextsuche integriert werden und ist ursprünglich in der Programmiersprache Java entwickelt sowie in andere Programmiersprachen, wie z.B. C, C# und Python, übersetzt worden. Der Zweck der Bibliothek ist das Bereitstellen von Funktionen zur Informationsgewinnung, häufig in der englischen Literatur auch unter dem Acronym *IR* zu finden, was für *Information Retrieval* steht [4 S. 6]. Der Begriff Informationsgewinnung wird durch folgendes, aus dem Englischen übersetztes Zitat erklärt:

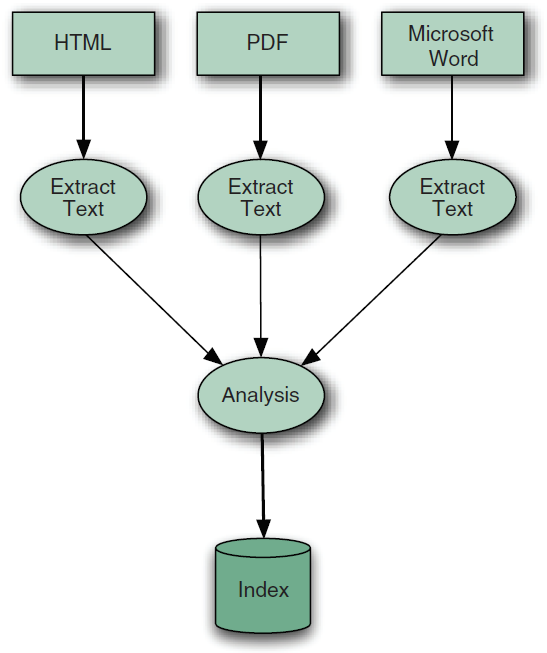
„*Informationsgewinnung ist das Finden von Material (zumeist Dokumente), welches unstrukturiert vorliegt (für gewöhnlich in Textform) und dient der Erfüllung eines Informationsbedarfs aus einer großen Datenansammlung heraus (gespeichert auf Computern)“* [7 S. 11].

Zur Gewinnung von Informationen müssen Daten zunächst so aufbereitet werden, dass sie für einen Computer leichter zu durchsuchen sind. Lucene kann dazu Daten aus Texten extrahieren und in einer eigenen Indexdatenbank so ablegen, dass sie schneller gefunden werden, als wenn der gesamte Text nach ihnen durchkämmt werden muss. Dabei ist es völlig unerheblich, ob es sich um Textdaten aus einer Webseite, einer Email oder einem digitalen Dokument handelt. Auch das Format der Datei kann unterschiedlich sein, z.B. PDF, XML oder als Microsoft Word-Dokument [4 S. 7].

Der Prozess des Extrahierens von Daten und Aufbaus einer Indexdatenbank wird auch als *Indexierung* bezeichnet und im folgenden Abschnitt detaillierter ausgeführt werden. Lucene verwendet dazu intern eine dokumentenorientierte *NoSQL*-Datenbank. Der Begriff *NoSQL* bedeutet übersetzt *Nicht nur SQL* und bezeichnet ein Datenbanksystem, welches u.a. nicht auf Relationen bzw. Tabellen basiert, wie es bei relationalen Datenbanken der Fall ist. NoSQL-Datenbanken sind entweder schemafrei oder haben schwache Einschränkungen, was die Definition der Datentypen betrifft. Außerdem sind sie auf eine Verteilung von großen Datenmengen auf unterschiedliche Computer und eine horizontale Skalierbarkeit ausgerichtet, wodurch weitere Rechner dynamisch eingebunden werden können, die einen Teil der Last bzw. Daten übernehmen [8 S. 2f.].

Dokumentenorientierte NoSQL-Datenbanken speichern Daten in strukturierten Dateiformaten, wie z.B. *JSON*, welches Daten als Schlüssel-Wert-Paare in JavaScript-Objekten ablegt [8 S. 8].

### Aufbau der Indexdatenbank

Die Grundidee hinter der Indexierung beruht darauf, dass die Daten selbst als Schlüsselwörter dienen, um zugehörige Dokument zu finden, in dem diese Schlagwörter enthalten sind. Dieses Konzept wird als *invertierter Index* bezeichnet. Um die Indexdatenbank aufbauen zu können, werden drei wesentliche Schritte benötigt, die auf **Abbildung 1** als Pfeile dargestellt sind. Prinzipiell werden dabei aus der Originalquelle Wörter extrahiert und in einer Lucene eigenen Dokumentdatenstruktur abgelegt, um die darin enthaltenen Daten analysieren zu können.

Ziel der Analyse ist es, Schlagwörter, sogenannte *Token*, zu gewinnen. Sie bilden zusammen mit dem jeweiligen Lucene-Dokument, im Folgenden nur noch als *Dokument* bezeichnet, Schlüssel-Wert-Paare, welche in der Indexdatenbank gespeichert werden [4 S. 34].

Ein Dokument definiert alle Daten, die zur Identifizierung eines Suchergebnisses gehören. Gespeichert werden die Daten in Feldern verschiedenen Typs. Die sogenannte *Schema*-Datei speichert sämtliche Informationen zu den definierten Feldern. Beispielsweise kann darin festgelegt sein, dass ein Feld nur für die Token-Analyse indexiert, jedoch nicht für die Dokumentwiederherstellung dauerhaft in der Indexdatenbank abgespeichert werden. Dies hilft, das Datenvolumen des Indexes so klein wie möglich zu halten, um ihn schneller durchsuchen zu können und Suchergebnisse nur mit für die Anwendung relevanten Daten auszustatten [7 S. 124]. Ein Beispiel für Felddefinitionen innerhalb der Schema-Datei ist im Anhang 11.1 zu finden. Mehr zur Definition von Dokumenten ist in Abschnitt 4.2.3 am Beispiel des Prototyps erläutert.

Abbildung 1: Indexierungsprozess von Lucene [4 S. 34]

In den Solr-Versionen vor Version 4.0 hieß diese Datei *schema.xml* und konnte manuell geändert werden. Ab 4.0 gibt es mehrere Möglichkeiten für den Dateinamen, je nachdem ob es sich um die SolrCloud-Anwendung, in der es keine Schema-Datei lokal gibt. Sonst kann das Schema über eine eigene *API*[[7]](#footnote-7) oder über die Weboberfläche von Solr bearbeitet werden, aber auch weiterhin manuell, was jedoch nicht von der offiziellen Dokumentation empfohlen wird [9]. Im Schema werden folgende Konfigurationsdaten zur Indexierung gespeichert.

Tabelle 1: Zu definierende Eigenschaften einer Solr-Schema-Datei

|  |  |  |
| --- | --- | --- |
| **Schema-Eigenschaft** | **XML-Tag-Name** | **Beschreibung** |
| Feld | field | Speichert ein Datenelement eines Feldtyps |
| Feldtyp | fieldType | Definiert den Datentyp des Feldes (u.a. string, int, date, boolean, double, binary) |
| Dynamisches Feld | dynamicField | Gibt ein beliebiges Präfix oder Suffix an, sodass alle Felder, deren Name damit versehen wurde, betrachtet werden (z.B. „\*\_s“ für alle Felder des Typs string) |
| Kopierfeld | copyField | Definiert von welchem Quellfeld in welches Zielfeld der Wert kopiert werden soll |
| Eindeutiger Schlüssel | uniqueKey | Wert, der ein Dokument eindeutig kennzeichnet, sodass darauf schnell zugegriffen werden kann |
| Analysierer | analyzer | Java-Klasse zum Analysieren von Daten als Vorbereitung zur Definition von Token (kann auch aus Tokenbilder und Filter zusammengesetzt werden) |
| Tokenbilder | tokenizer | Java-Klasse zum Definieren von Token aus zuvor extrahierten Daten |
| Filter | filter | Java-Klasse zur Nachbearbeitung von Token |

**Abbildung 2** stellt den Zusammenhang zwischen Dokument, Feldern, Analysierern, Token und Index während des Indexierungsprozesses dar. Dabei ist noch unerheblich, woher die Daten genau kommen und wie sie extrahiert und in Dokumenten gespeichert werden. Dies wird genauer in Kapitel 2.2.3 und am Beispiel in Kapitel 4.2 betrachtet. Zunächst kann von vorhandenen Dokumenten mit in Feldern gespeicherten Daten ausgegangen werden. Die Felder müssen nun einzeln nach indexierbaren Token analysiert werden.

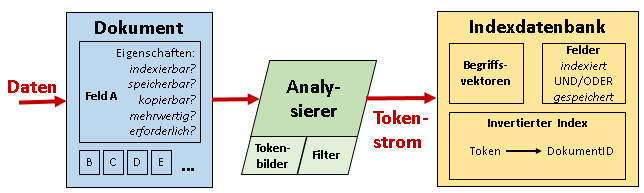


Abbildung 2: Indexierungsvorgang mit Zusammenhang zwischen Dokument, Feldern, Analysierer und Indexdatenbank

Während der Analyse der Felder werden viele verschiedene sprachabhängige Aufgaben abgearbeitet. Pro Aufgabe ist eine spezifische Java-Klasse vom Typ *Analyzer* definiert. Es gibt zahlreiche von Lucene bereitgestellte Analyzer-Klassen, die dafür sorgen, dass u.a. irrelevante Wörter, wie z.B. Präpositionen oder Füllwörter, sowie Satzzeichen und unnötige Leerzeichen entfernt oder alle Buchstaben zu Kleinbuchstaben konvertiert werden. Sollte für eine Spezialanforderung keine Klasse von Lucene bereitgestellt werden, ist es dank der modularen Software-Architektur[[8]](#footnote-8) möglich, eine eigene zu programmieren und zur Programmbibliothek hinzuzufügen. Die so extrahierten Wörter sind am Ende für die Bildung von Token und damit für den Aufbau des invertierten Index verwendbar [4 S. 35].

Doch wie sieht dieser Index genau aus? Grundsätzlich besteht er aus einem oder mehreren Segmenten. Alle Segmente stellen jeweils einen eigenständigen Index dar, der eine Teilmenge der zu indexierenden Dokumenten beinhaltet. Mit jeder Operation auf dem Index, z.B. durch Hinzufügen eines neuen Eintrags oder Löschen eines bestehenden, wird ein neues Segment erzeugt. Jedes Segment setzt sich aus mehreren Dateien zusammen, die verschiedene Indexeigenschaften abbilden, wie z.B. *Begriffsvektoren*, gespeicherte Felder und den invertierten Index [4 S. 36ff.].

Ein Begriffsvektor, bzw. *term vector* auf Englisch, besteht aus einer Reihe von Token, die einem gespeicherten Feld zugeordnet sind. Der invertierte Index beinhaltet dazu die Zuordnung von einem Token zu allen Dokumenten, die den Token enthalten. Mit Hilfe dieser speziellen Indexdateistruktur können schnell und effizient zu den eingegebenen Suchwörtern Dokumente durchsucht und nach Rekonstruktion der gespeicherten Originaldaten als Ergebnis zurückgegeben werden.

### Durchsuchen der Indexdatenbank

Um den Lucene-Index durchsuchen zu können, muss zunächst eine gültige Abfrage erzeugt werden, die in einem *Query*-Objekt verpackt ist. Dies kann auf zwei verschiedenen Wegen erfolgen. Entweder wird die *API* von Lucene angesprochen und eine von mehreren *Query*-Klassen genutzt, wie z.B. *TermQuery* zur Definition einer Anfrage mit nur einem Suchwort. Oder eine Instanz der speziellen *QueryParser*-Klasse kümmert sich um die Erzeugung einer gültigen Abfrage. Der Unterschied zwischen den beiden Möglichkeiten ist, dass der QueryParser mit Hilfe einer Analyzer-Instanz die vom Nutzer eingegebenen Suchwörter auf die gleiche Art und Weise transformiert, wie es beim Indexieren geschehen ist. Dadurch ist ein Vergleich mit den im Index gespeicherten Token möglich. Ansonsten muss bei der Verwendung der Query-Klassen entsprechend darauf geachtet werden, dass die Eingabewörter im Index gefunden werden können und gegebenenfalls eine separate Konvertierung implementiert werden [4 S. 76ff.].

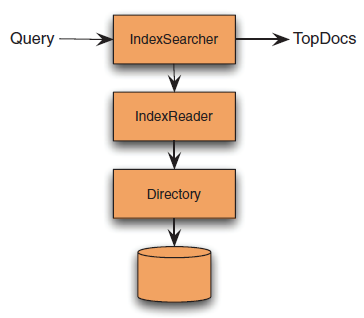
Mit Hilfe der Query-Klassen bzw. des QueryParsers ist ein Query-Objekt instanziierbar, welches von einer Instanz der *IndexSearcher*-Klasse ausgewertet werden kann. Als Ergebnis gibt der IndexSearcher ein TopDocs-Objekt zurück, welches Informationen zu den gefundenen Dokumenten bereitstellt, u.a. auch deren Identifikationsnummern, um sie aus den im vorherigen Abschnitt beschriebenen Indexdateien rekonstruieren zu können. Für den Zugriff auf den Index nutzt der IndexSearcher ein *IndexReader*-Objekt, welches die zum Zeitpunkt seiner Instanziierung vorhandene Momentaufnahme des Indexes ausliest und für die Durchsuchung zur Verfügung stellt. Auf **Abbildung 3** sind die Beziehungen der einzelnen für die Suche verwendeten Klassen abgebildet. Die *Directory*-Klasse ist eine Schnittstelle von Lucene, mit deren Hilfe der IndexReader auf die Indexdateien zugreifen kann.

Abbildung 3: Beziehung zwischen den Hauptklassen von Lucene zum Durchsuchen des Indexes

Als Ergebnis werden jedoch nicht zwangsläufig sämtliche gefundenen Dokumente angegeben, da dies unter Umständen zu viele sein können, um alle gleichzeitig im Computerspeicher vorgehalten zu werden. Daher liefert die TopDocs-Klasse nur eine angegebene maximale Anzahl von Dokumentenschlüsseln, die nach einer definierten Sortierbedingung geordnet sind. Um weitere Dokumente zu erhalten, wird empfohlen, die gesamte Suchabfrage mit einer höheren maximalen Anzahl an Ergebnissen zu wiederholen, anstatt von vornherein eine hohe Zahl zu erfragen, da die meisten Benutzer ihr gewünschtes Ergebnis bereits auf der ersten Seite finden und die Rekonstruktion von gefundenen Dokumenten deutlich länger dauert, je mehr Dokumente zurückgegeben werden sollen. Eine seitenweise Darstellung der nächsten, z.B. zehn, Ergebnisse ist somit performanter, als alle bereits in eine Liste zu laden [4 S. 84].

### Sortierung der Suchergebnisse

Die standardmäßig genutzte Sortiereigenschaft zur Ordnung der Suchergebnisse ist ein spezifischer Bewertungsmaßstab, der sogenannte *Score*. Er gibt an, wie relevant das Dokument bezüglich der Suchanfrage ist. Aus einer komplexen Berechnung verschiedener Faktoren ergibt sich der Score als Dezimalzahl. Je größer diese ist, desto relevanter ist das betrachtete Dokument.

In die Berechnung des Score fließen mit ein

* die *Begriffsfrequenz*, welche die Auftrittshäufigkeit eines Suchbegriffs in einem Dokument angibt,
* die inverse *Dokumentfrequenz*, die die Anzahl der zugehörigen Dokumente darstellt,
* der für bestimmte Dokumente und Dokumentfelder zuvor definierte Verstärkungsfaktor, auch *boost* genannt, welcher eine höhere Wertigkeit als Dokumente bzw. Felder mit gar keinem oder geringerem boost ausdrückt,
* der sogenannte *Koordinationsfaktor*, der sich aus der Anzahl der im jeweiligen Dokument vorkommenden, unterschiedlichen Suchbegriffe ermitteln lässt,
* der Normalisierungswert für die Länge eines Dokumentfeldes, also wieviel Begriffe ein Feld beinhaltet, wobei kürzere Felder einen größeren Faktor ergeben,
* und der Normalisierungswert für die Suchanfrage, der sich aus der Summe der Quadrate der einzelnen Gewichtungen pro Suchbegriff ergibt, da in der Anfrage einzelne Suchwörter mit einem eigenen Verstärkungsfaktor angegeben werden können [4 S. 87f.].

Lucene bietet mit der *explain*-Methode der IndexSearcher-Klasse die Möglichkeit das Ergebnis der einzelnen Faktoren sich anzeigen lassen zu können, sodass das Zustandekommen des Relevanzwerts für ein Dokument nachvollziehbar ist [4 S. 88]. Neben der Sortierung nach dem Relevanzwert können die gefundenen Dokumente auch nach einem oder mehreren enthaltenen Feldwerten oder nach deren Indexnummer geordnet werden [4 S. 156ff.]

Es sei an dieser Stelle noch erwähnt, dass zur weiteren Spezialisierung einer Volltextsuche für die eigene Anwendung neben den bereits von Lucene mitgebrachten Hilfsklassen auch eigene Klassen programmiert werden können, die lediglich von einer entsprechenden abstrakten Basisklasse erben müssen. Im nächsten Abschnitt wird Solr vorgestellt, das eine separate Zugriffsschicht für Lucene darstellt und dessen Funktionen leichter zugänglich werden lässt.

## Apache Solr

Apache Lucene ist wie bereits erwähnt eine reine Programmbibliothek. Um sie für die eigene Anwendung zur Volltextsuche nutzen zu können, muss entweder die Anwendung in einer der Sprachen, in denen Lucene existiert, entwickelt worden sein oder die Suche muss in eine unabhängige Komponente ausgelagert werden, die über standardisierte Schnittstellen aufrufbar ist. Eine solche Komponente stellt Apache Solr dar. Genauer gesagt handelt es sich dabei um eine Web-Anwendung, deren serverseitiger Quelltext in Java programmiert ist und die Lucene-gestützte Volltextsuche leichter konfigurierbar und ausführbar werden lässt, ohne eine eigene Suchanwendung entwickeln zu müssen [7 S. 13].

Über die Web-Benutzeroberfläche können direkt von der Definition der zu indexierenden Dokumenten bis zur Darstellung von Suchergebnissen sämtliche Teilschritte der Volltextsuche nachvollzogen werden. Und dank der gut dokumentierten *REST-konformen Web-API[[9]](#footnote-9)* existiert eine standardisierte Schnittstelle, um aus dem Computernetzwerk auf den Solr-Server via HTTP[[10]](#footnote-10)-Anfragen zugreifen zu können. Weitere Möglichkeiten und Funktionen von Solr werden im folgenden Abschnitt erläutert.

### Möglichkeiten von Solr

Mit einer Volltextsuchplattform, wie z.B. Solr, ist es möglich, dass nicht nur Informationen gefunden werden, die erwartet wurden, sondern auch solche, von denen man keine Ahnung hatte, dass sie existieren. Ein Beispiel macht es deutlicher.

Ein Mitarbeiter der Personalabteilung sucht mit Hilfe einer fiktiven Suchfunktion im HR Portal nach einem Bewerber, der bereits Erfahrungen in der Leitung eines Projekts hat. Er verwendet als Suchanfrage die Eingabe „*Bewerbung* *Projekt Leitung Projektleiter“* und erhält eine nach Relevanz geordnete Liste von Ergebnissen, die diese Schlagwörter irgendwo enthalten. Er geht die einzelnen gefundenen Bewerbungen durch und findet an zehnter Stelle eine, die ihn besonders beeindruckt, obwohl schnell klar wird, dass der Bewerber noch keine direkten Erfahrungen als Projektleiter hat, dafür aber in der Leitung von Seminaren. Außerdem ist er ehrenamtlich in der Leitung eines Vereins vertreten. Die sozialen Fähigkeiten sowie der Rest der Bewerbung haben den Personalmitarbeiter dennoch überzeugt, den Bewerber zu einem Gespräch einzuladen.

Das Beispiel zeigt, dass die Volltextsuche abhängig von den gewählten Suchbegriffen Resultate erzielen kann, die mit einer herkömmlichen Datenbankabfrage nicht möglich sind. Es reicht, dass bereits ein Suchwort, selbst wenn es falsch geschrieben wurde, in die Suchergebnisliste aufgenommen werden kann. Die in Abschnitt 2.1.3 erläuterte Sortierung der Ergebnisse nach Relevanzkriterien ermöglicht dennoch, dass die besten Ergebnisse zuerst zu sehen sind. Darüber hinaus können scheinbar weniger relevante Dokumente zusätzlich gefunden und somit entdeckt werden, was unter Umständen auch zu weiterem Informationsgewinn führen kann.

Außerdem unterstützt Solr den Suchenden je nach Vorkonfiguration mit vielen Hilfsfunktionen, wie z.B.

* die Autovervollständigung der Suchbegriffe,
* die sprachabhängige Fehlerkorrektur der Eingabewörter,
* die Kategorisierung der Suchergebnisse, sodass diese weiter gefiltert werden können,
* die flexible Gewichtung von zu durchsuchenden Daten für eine bessere Relevanz der Suchergebnisse
* und das ausschnittartige Anzeigen von gefundenen Sucheingaben innerhalb der im Index gespeicherten Daten [7 S. 7f.].

Mit Solr ist es zudem möglich, auf einem Server mehrere Indexdatenbanken anzulegen, die als Solr-Kerne bezeichnet werden. Über die Angabe des eindeutigen Kernnamens können diese dann von verschiedenen Anwendungen gleichzeitig für die Suche genutzt werden. Außerdem können die Kerne dynamisch zur Laufzeit miteinander verschmolzen bzw. in zwei oder mehrere aufgeteilt werden - siehe dazu die *CoreAdmin API* aus der offiziellen Dokumentation von Solr [9].

Darüber hinaus ist es seit der Solr-Version 4.0 möglich, über die Erweiterung *SolrCloud* mehrere Solr-Server in einem Verbund mehrerer Rechner anzuordnen, sodass der Gesamtindex verteilt wird. Die einzelnen Indexteile werden als *Shards* bezeichnet werden und stellen voneinander unabhängige Teilmengen der indexierten Dokumente dar. Somit können sehr große Datenmengen durchsucht werden, die nicht auf einem Server genug Speicherplatz haben. Auch zu diesem Thema sei auf die Dokumentation unter dem Abschnitt *SolrCloud* verwiesen [9]. Trotz der vielen Möglichkeiten von Solr gibt es auch Fälle, bei denen sich eine Implementierung für die eigene Anwendung nicht eignet.

### Grenzen von Solr

Erzielt die Volltextsuche Millionen von Resultaten, dauert die Bereitstellung der zugeordneten Daten äußerst lange, da deren Datenquellen aus der Indexdatenbank einzeln rekonstruiert werden müssen. Genauso ineffizient ist eine tiefere Analyse der gefundenen Daten, wenn z.B. sehr viele Beziehungen zwischen Dokumenten bestehen, wie es bei relationalen Datenbanken der Fall ist. Nicht umsonst fordert die Definition eines Dokuments eine flache Struktur. Es soll verhindern, dass das Zwischenspeichern von großen Teilen des Indexes solange notwendig ist, bis das gewünschte Dokument zusammengesetzt wurde [7 S. 9]. Für diese Aufgabe sind relationale Datenbanken deutlich besser optimiert [5].

Eine weitere Einschränkung beim Umgang mit Solr ist die fehlende Möglichkeit, feingranulare Benutzerrechte auf Dokumente vergeben zu können. Dies muss außerhalb von Solr geschehen [7 S. 9].

### Werkzeuge zur Indexierung

Der in Abschnitt 2.1.1 beschriebene Aufbau der Indexdatenbank stellt eine Grundvoraussetzung für die Volltextsuche dar und benötigt einen großen planerischen Aufwand, der entscheidend für die Beantwortung einer Suchanfrage ist. Schließlich wird dabei definiert, welche Daten für die Suche zu berücksichtigen sind und wie viele Informationen der Suchende letztendlich erhält. Solr ermöglicht das automatisierte Indexieren auf zwei verschiedenen Wegen. Entweder können definierte Dokumente über die REST-Schnittstelle von Solr an den Server gesendet werden, welcher mit Hilfe einer Instanz des *IndexHandler* die Indexierung vornimmt.

Unterstützt werden dabei als Datenquelle die Dateiformate XML, CSV und JSON. Weitere Dateiarten, wie z.B. PDF, Microsoft Word-Dokumente oder HTML-Dateien können von einem dafür entworfenen *Framework*[[11]](#footnote-11) von Apache namens *Tika* ausgelesen und indexiert werden, wie Abschnitt *Uploading Data with Solr Cell using Apache Tika* aus der Solr-Dokumentation ausführlich beschreibt [9]. Zur Verwendung des IndexHandler sind sämtliche zu indexierenden Dokumente bereits vorher anzulegen, sodass sie direkt an den Solr-Server geschickt werden können. Insofern eignet sich dieser Weg besonders gut zum kurzfristigen Indexieren von Daten.

Das zweite Werkzeug zur Indexierung nennt sich *Data Import Handler*, das im Folgenden als DIH abgekürzt wird. Es bietet ein automatisiertes Abfragen von verschiedenen Datenquellen sowie das Anlegen und anschließende Indexieren von Dokumenten. Der klassische Anwendungsfall des DIH stellen strukturierte Datenquellen dar, wie relationale Datenbanken, die über SQL-Anfragen ausgelesen werden [7 S. 146]. Dies gilt aber nur, wenn für den Zugriff auf den Datenbank-Server ein JDBC-Treiber[[12]](#footnote-12) existiert. Aber auch lokal gespeicherte Dateien, die sich im XML-Format befinden oder reinen Volltext enthalten sind als Datenquelle denkbar, genauso wie Daten, die über eine *URL*[[13]](#footnote-13) innerhalb des Computernetzwerks erreichbar sind.

Sollten die von Solr bereits unterstützten Datenquellen nicht zur eigenen Software-Umgebung passen, ist es dazu noch möglich, eine komplett eigene Datenquelle zu verwenden. Dafür muss eine Java-Klasse erstellt werden, die von der abstrakten Basisklasse *DataSource* erbt. Dies ist, so viel sei bereits an dieser Stelle vorweg genommen, für die Implementierung des Prototyps notwendig und wird in Abschnitt 4.2.2 beschrieben.

Nachdem vom DIH die Datenquelle ausgelesen wurde, übernimmt ein zugehöriges Java–Objekt der Klasse *EntityProcessor* das Anlegen von Dokumenten nach der in der DIH-Konfigurationsdatei festgelegten Definition. Zum Schluss sendet das EntityProcessor-Objekt die erstellten Dokumente an den Solr-Server, welcher für die eigentliche Indexierung verantwortlich ist. Gestartet werden kann der gesamte Prozess entweder über die Weboberfläche von Solr oder über die REST-Schnittstelle von Solr durch Aufruf der entsprechenden URL mit dem relativen Pfad */dataimport* – ausgehend von der Basisadresse des Solr-Server. Dabei stehen zwei Arten des Datenimports zur Verfügung: der vollständige und der partielle.

Für den vollständigen Import ist das Kommando *full-import* mit anzugeben, sodass alle Daten, die in Feldern definiert sind, abgefragt und gespeichert werden, wobei über die zusätzliche Angabe *clean* der Index zuvor komplett gelöscht werden kann. Sonst überschreibt der Vorgang den bisherigen Index. Solr legt dabei immer einen Zeitstempel zum angelegten Index an, wodurch der partielle Import erst möglich ist. Letzterer lässt sich über das Kommando *delta-import* ansteuern. Ein Beispiel für den Aufruf des DIH über die lokale URL mit dem Kommando zum teilweisen Import sieht folgendermaßen aus.

http://localhost:8983/solr/dih/dataimport?command=delta-import

Wie zuvor erwähnt, verwendet der partielle Import dazu einen Zeitstempel, der mit einem weiteren Zeitstempel aus der Datenquelle verglichen wird, sodass nur neuere Daten seit dem letzten Änderungszeitpunkt im Index aktualisiert werden. Dies kann den Indexierungsprozess drastisch reduzieren, worauf in Hinblick auf den zu entwickelnden Prototypen in Abschnitt 3.6 näher eingegangen wird. Wie der DIH zu konfigurieren ist, sei in Abschnitt 4.2.3 am Beispiel des Prototyps gezeigt. Eine ausführliche Beschreibung der DIH-Optionen und –Parameter ist in der offiziellen Dokumentation von Solr im Abschnitt *Indexing and Basic Data Operations* unter *Uploading Structured Data Store with the Data Import Handler* zu finden [9].

# Vorüberlegungen zur Umsetzung in der Sage HR Suite

Nach der Festlegung des Programmbereichs innerhalb der PW mit dem HR Portal und der beiden Module darin, *Bewerber* und *Kalender*, sind zunächst ein paar technologische Vorüberlegungen zu unternehmen, um entscheiden zu können, wie die einzelnen Schritte bis zur fertigen Volltextsuche mit Hilfe von Solr umgesetzt werden können.

Es gilt folgende Fragen zu klären:

* Wie ist das HR Portal aufgebaut?
* Woher kommen die dort angezeigten Daten?
* Wie funktioniert die Navigation zu den einzelnen Benutzeroberflächen?
* Wie können die verschiedenen Benutzerrollen von der Volltextsuche berücksichtigt werden?

Darüber hinaus ergaben sich während der Implementierung zwei weitere Fragen:

* Wie kann ein indexiertes Dokument eindeutig gekennzeichnet werden, damit es beim erneuten Indexieren aktualisiert wird?
* Ist es möglich, nur die zuletzt geänderten Daten neu zu indexieren, um die Dauer des Indexierungsprozesses verkürzen zu können?

Diese Fragen werden in den folgenden Unterkapiteln der Reihenfolge nach behandelt.

## Software-Architektur vom HR Portal

Beim HR Portal der PW handelt es sich um eine *ASP.NET MVC*[[14]](#footnote-14) – Web-Anwendung, welche der MVC-Software-Architektur folgt. Sie unterteilt jede Benutzeroberfläche bzw. Ansicht einer Anwendung in die drei Bereiche *Model* (M), *View* (V) und *Controller* (C). Das Model repräsentiert die Daten, welche von der zugehörigen Ansicht, auch View genannt, für den Anwender dargestellt werden. Der Controller fungiert als Vermittler zwischen den anderen beiden Bereichen.

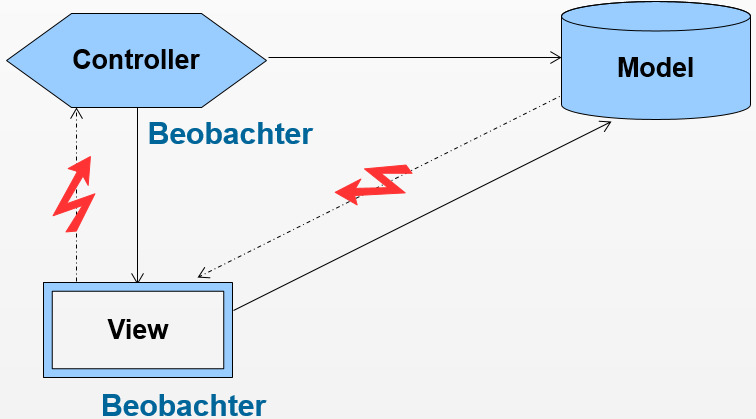
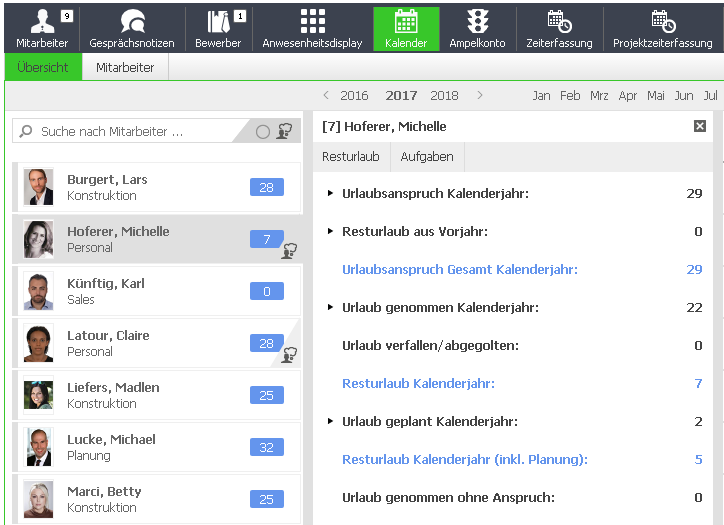
Ziel von MVC ist eine Trennung der Bereiche, die teilweise unabhängig voneinander entwickelt und getestet werden können. Die Beziehungen zwischen Model, View und Controller sind auf **Abbildung 4** anschaulich dargestellt. Durchgezogene Pfeillinien geben an, dass ein Bereich Kenntnis vom zugeordneten Bereich hat, aber nicht umgekehrt. Daraus wird ersichtlich, dass das Model völlig unabhängig von den anderen beiden Bereichen ist.

Abbildung 4: Schematische Darstellung des Architekturmusters MVC (durchgezogene Linie: direkte Beziehung; gestrichelte Linie: Benachrichtigung über Ereignis [roter Blitz])

Die View *kennt* das Model und sobald sie von diesem benachrichtigt wird, dass sich angezeigte Daten im Model geändert haben, fragt sie diese ab, um sich zu aktualisieren. Sobald sich durch Benutzereingaben die View-Daten ändern, sendet sie eine Ereignisnachricht an jede Instanz, die sich dafür angemeldet hat – in diesem Fall nur der Controller. Daraufhin fragt der Controller die geänderten Daten von der View ab und gibt diese an das Model weiter, sodass es sich aktualisiert. Somit ist lediglich der Controller abhängig von beiden anderen Bereichen [10 S. 380ff.].

Weil das ASP.NET MVC-Framework für Web-Anwendungen konzipiert ist, verwendet es das MVC- Muster in leicht abgewandelter Form. Nachdem ein *Client*[[15]](#footnote-15) eine REST-konforme HTTP-Anfrage an die URL-Adresse des zu einer View zugehörigen Controllers[[16]](#footnote-16) gesendet hat, schickt dieser sämtliche zu der View zugehörigen HTML[[17]](#footnote-17)-, CSS[[18]](#footnote-18)- und JavaScript[[19]](#footnote-19) -Dateien in einer HTTP-Antwort zurück an. Daraufhin kann der Web-Browser des Client die View laden und darstellen. Der Controller stellt somit die Netzwerkschnittstelle der Anwendung zwischen Client und Server dar und sorgt neben dem Ausliefern der View an den Client auch für die Änderung des auf dem Server befindlichen Models. Der Unterschied zum klassischen MVC-Muster besteht darin, dass die View keine Ereignisnachricht vom Model erhalten kann, wenn sich Model-Daten geändert haben. In der Client-Server-Architektur ist keine Möglichkeit vorgesehen, dass der Server ohne Anfrage eine Nachricht an den Client schicken kann. D.h. die Model-Änderungen werden erst nach erneuter Client-Anfrage von der neu ausgelieferten View übernommen [11].

Jedoch hält sich das HR Portal nicht strikt an die Konventionen des ASP.NET MVC-Framework, da nicht jede View über einen eigenen Controller vom Server geladen wird. Einige Module innerhalb des HR Portals ähneln eher einer *Single-Page-Application[[20]](#footnote-20)*, kurz *SPA*, die sich insofern von ASP.NET MVC-Anwendungen unterscheidet, als dass nur beim Aufruf der Hauptwebseite sämtliche Views vom Server geladen werden und anschließend lediglich reine Anwendungsdaten mit Hilfe von JavaScript und *AJAX*[[21]](#footnote-21)zwischen Client und Server ausgetauscht werden. Dies ermöglicht, dass deutlich weniger Daten über das Netzwerk versendet werden müssen, da nach dem erstmaligen Laden der Web-Anwendung nur die sich verändernden Teile der Webseite gesendet werden.

Beispielsweise stellt die Kalenderübersicht des HR Portals eine reine SPA dar, deren sämtliche Dateien über einen Controller-Aufruf komplett vom Server an den Client geschickt werden. Wenn dann nach Auswahl eines Mitarbeiters die Übersicht zu seinem Resturlaub in einer eingeblendeten View auftaucht, dann lag diese zuvor bereits im Browser geladen aber unsichtbar vor und wurde mit Hilfe von JavaScript sichtbar. Nur die Daten zum Resturlaub werden beim Klick auf die Arbeitnehmerschaltfläche nachgeladen und sofort angezeigt, wie auf **Abbildung 5** zu sehen ist.

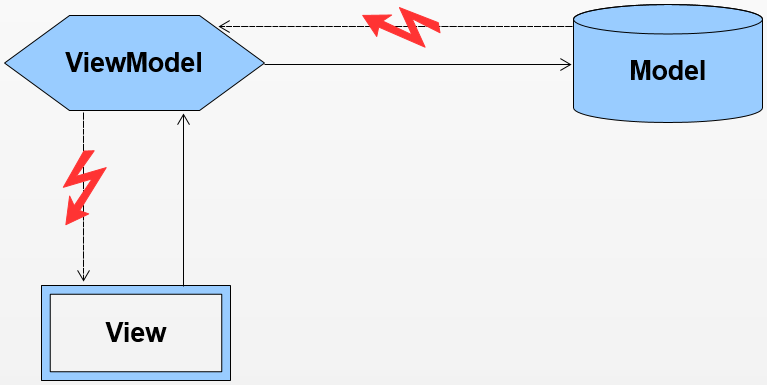
Die Darstellung der Benutzeroberflächen erfolgt durch ein weiteres proprietäres JavaScript-Framework mit dem Namen *Kendo UI*. Dieses richtet sich nach einem anderen Architekturmuster, welches sich *MVVM* nennt. Wie das MVC-Muster definiert MVVM auch eine Model-Schicht für Datenmodelle, eine View-Schicht für die reine Anwendungsansicht. Anstelle eines Controllers kümmert sich jedoch eine sogenannte *ViewModel*-Komponente um die Kommunikation zwischen View und Model. Wie auf **Abbildung 6** zu sehen ist, speichert das ViewModel die in der View gerade angezeigten Daten und aktualisiert entweder das Model, wenn in der View Daten geändert wurden, oder die View, wenn sich die Daten auf dem Server geändert haben [12]. Letzterer Fall ist, wie bereits zuvor erläutert, im HR Portal nicht umsetzbar, sofern eine REST-Schnittstelle zwischen Client und Server verwendet wird.

Abbildung 5: Bildschirmausschnitt zum Aufruf der Resturlaubansicht in Kalenderübersicht des HR Portals. Die gesamte Ansicht zum Menüpunkt Übersicht ist eine SPA.

Abbildung 6: Schematische Darstellung des Architekturmusters MVVM (durchgezogene Linie: direkte Beziehung; gestrichelte Linie: Benachrichtigung über Ereignis [roter Blitz])

Das MVVM-Muster verwendet zudem ein Konzept, welches den automatisierten Datenabgleich zweier miteinander verbundener Objekteigenschaften als Ziel hat. Das Konzept nennt sich *Data* *Binding*. In MVVM stellt das ViewModel Eigenschaften bereit, an die sich die zugehörige View binden kann. Ändert sich diese Eigenschaft in der View, z.B. durch eine neue Benutzereingabe, so wird von der View ein Ereignis ausgelöst, dass die zugehörige ViewModel-Eigenschaft mit den Eingabedaten abgleicht. Das Data Binding wird an entsprechender Stelle von der View definiert. Da diese Verbindung bidirektional sein kann, ist es möglich, dass auch die View geänderte Daten vom ViewModel abrufen kann [13].

Während beim MVC-Muster noch eine geringe Abhängigkeit der View vom Model besteht, indem sie sich nach einer Benachrichtigung die Daten vom Model selbständig abholen muss, wird im MVVM-Muster die View vom Model komplett unabhängig gehalten. Sie muss jede Daten-Anfrage direkt vom ViewModel ausführen lassen, sodass die vollständige Trennung von Model und View gewährleistet ist und beide beliebig ausgetauscht werden können. Ein weiterer Vorteil von MVVM im Vergleich zu MVC ist die Möglichkeit, pro View verschiedene ViewModel bereitstellen zu können, da diese nur *bindungsfähige* Eigenschaften anbieten und nicht wissen, wer sich daran bindet. Es müssen lediglich die Namenskonventionen der ViewModel-Eigenschaften berücksichtigt werden, um keine Anpassungen in der View notwendig werden zu lassen [12].

Nachdem für das HR Portal das Zusammenspiel zwischen Views, Controllers, ViewModels und Models grundlegend geklärt ist, stellt sich die Frage, woher denn genau die angefragten Model-Daten kommen? Schließlich stellt ein Model nur ein abstraktes Konstrukt dar, das gespeicherte Daten repräsentiert. Die Art der Datenquelle wird dabei außer Acht gelassen.

## REST-konforme Web-API als Schnittstelle zur Datenquelle der PW

Für die gesamte PW existiert eine einzige relationale Datenbank, die auf einem Microsoft SQL Server verwaltet wird. Neuere Anwendungen, wie das HR Portal, greifen jedoch nicht direkt auf diese Datenbank zu, sondern indirekt über eine extra dafür entwickelte REST-konforme Web-API, kurz WebAPI. Sie fungiert als wohl dokumentierte Schnittstelle für den Zugriff einerseits auf PW-Daten, andererseits aber auch auf die PW-Geschäftslogik, die unterschiedlichste Berechnungsprozesse der PW zugänglich macht. Die Ausnahme stellt u.a. die Lohnabrechnung dar, die teilweise noch ausschließlich in Microsoft Access[[22]](#footnote-22) entwickelt wurde. Erst später erfolgten bei Sage GmbH in Leipzig Neuentwicklungen von Desktop-Anwendungen mit Hilfe des .NET-Frameworks vorwiegend in der objektorientierten Programmiersprache *C#*.

Sobald eine Datenanfrage die WebAPI erreicht, prüft diese zuerst, ob die Anfrage von einem autorisierten Benutzer kommt und wenn dies der Fall ist, dann holt sie sich die entsprechenden Daten via SQL-Schnittstelle vom Datenbank-Server ab. Eine weitere Einschränkung bezüglich der Berechtigung, die angefragten Daten auch erhalten zu dürfen, stellt das Rollenkonzept des HR Portals dar, welches in Abschnitt 3.4 näher vorgestellt wird. Hat der Anfragende aufgrund seiner definierten HR Portal-Rolle nicht das Recht, Daten, z.B. von anderen Mitarbeitern, sehen zu dürfen, dann verwehrt die WebAPI auch den Datenzugriff.

Zwei Hauptgründe gibt es, warum die WebAPI als Schnittstelle zu den PW-Daten einem direkten Zugriff durch den Solr-Server vorgezogen werden sollte. Würde der Solr-Server direkt Daten abfragen wollen, müssten dazu sämtliche SQL-Anfragen, die bereits von der Geschäftslogik der PW durchgeführt werden, in redundanter Form neu definiert werden. Änderungen müssten demzufolge an beiden Stellen gleichzeitig vorgenommen werden, was fehleranfälliger ist, sobald einmal nicht daran gedacht wurde. Der zweite Hauptgrund besteht in der Fähigkeit der WebAPI, Benutzer für den Datenzugriff zu autorisieren. Dadurch ist es nicht notwendig, eine Authentifizierungsfunktion für den Solr-Server direkt einzusetzen, solange er nur über die WebAPI und nicht aus dem weltweiten Netzwerk heraus erreichbar ist.

Die Notwendigkeit, sich mit einem gültigen Benutzerkonto an der WebAPI zu authentifizieren, ist auch der Grund, warum das Indexieren der von dort abgefragten Daten durch den DIH nicht mit einer von Solr bereitgestellten DataSource-Klasse durchgeführt werden kann. Denn nachdem die Anmeldung authorisiert wurde, sendet der WebAPI-Server im Kopf der HTTP-Antwort einen zufällig erzeugten 32 Zeichen langen Sitzungsschlüssel mit, der für die Dauer der Anmeldung im Nachrichtenkopf von allen HTTP-Anfragen der Anwendung mitgeliefert werden muss, sollen die Anfragen nicht abgewiesen werden. Die *URLDataSource*-Klasse vom DIH zum Auslesen von Netzwerkdatenquellen via HTTP bietet standardmäßig keine Möglichkeit, den Kopf einer Anfragenachricht zu definieren. Wie die benutzerdefinierte DataSource-Klasse für den Prototyp implementiert werden kann, beschreibt Abschnitt 4.2.2.

## Navigation innerhalb des HR Portals

Auf **Abbildung 7** ist das Menü des HR Portals für die Navigation zu den einzelnen Modulen dargestellt. Einige der Bereiche, wie z.B. *Bewerber*, haben ein Untermenü, welches zu weiteren Views des Moduls führt. Das Menü gehört zum Anwendungskopf und wird nur einmal vom Server abgerufen und vom Browser dargestellt, während nach der Betätigung einer Navigationsschaltfläche die zugehörige View separat in den Bereich unterhalb des Menüs neu nachgeladen wird.

Abbildung 7: Menü zu den Hauptmodulen des HR Portals

Ein wenig komplexer gestaltet sich das Navigieren innerhalb einer View, die sich wie eine SPA verhält. Mit dem Navigieren bzw. der Navigation ist in diesem Kontext gemeint, wie Views ein- und ausgeblendet werden, sodass nur die Ansicht zu sehen ist, die zuvor angefordert wurde. Beispielsweise bietet die *Übersicht*-View innerhalb des Kalender-Moduls zwei verschiedene Ansichtsarten des Kalenders – entweder als Tabelle oder als Liste. Über zwei Schaltflächen kann immer nur eine der beiden Ansichten sichtbar werden.

Im HR Portal gibt es keinen einheitlichen Prozess für das Umschalten zwischen Views. Verschiedene Einblendeffekte sind über den Bindungsmechanismus des Kendo UI-Framework an einer zentralen Stelle definiert. Um den Mechanismus besser nachvollziehen zu können, wird er mit Hilfe eines vereinfachten Programmcodebeispiels auf **Abbildung 8** dargestellt. Ziel des Beispiels ist es, zu zeigen, wie zwischen zwei Views, A und B, mit Hilfe von Kendo UI umgeschaltet werden kann.

Abbildung 8: Code-Beispiel zum Umschalten zwischen zwei Views mit Hilfe des Kendo UI Frameworks

Im Quelltext ist auszugsweise der Inhalt von vier Dateien aufgeführt, die zum Anzeigen einer Haupt-View mit der Bezeichnung *example-view* im Browser geladen sind. In Zeile 2 bis 7 wird zunächst in einer eigenen JavaScript-Datei ein ViewModel namens *ExampleViewModel* angelegt. Dazu definiert Zeile 10 bis 15 in einer eigenen Datei ein JavaScript-Objekt zu der Haupt-View. In der wird zum einen das ViewModel mit Hilfe der *observable*-Funktion des *globalen* *kendo*-Objekts[[23]](#footnote-23) instanziiert. Zum anderen stellt das View-Objekt eine initialize-Funktion öffentlich[[24]](#footnote-24) bereit, die nach ihrem Aufruf das HTML-Element zur View mit dem ViewModel verknüpft. Im Jargon von Kendo UI nennt sich diese Verknüpfung *Bindung* oder auf Englisch *binding*. In einer eigenen .*cshtml*-Datei[[25]](#footnote-25) ist die HTML-Struktur der umzuschaltenden Views *exampleA-view* und *exampleB-view* innerhalb der Haupt-View zu sehen. Zunächst sind dort zwei Schaltflächen zum Anzeigen von A oder B definiert. Beide lösen beim Betätigen ein Klick-Ereignis aus, welches von der zugewiesenen ViewModel-Funktion *onChangeViewModeClick* behandelt wird und u.a. das angeklickte HTML-Element mit übergibt. Dadurch kann in der Funktion der Wert des Datenattributs namens *key* ausgelesen und die ViewModel-Eigenschaft *viewMode* überschrieben werden.

Außerdem sind in der .cshtml-Datei zwei Views A und B festgelegt, die mit dem Kendo UI eigenen Datenattribut *bind* versehen wurden, welches das in Zeile 30 bis 39 definierte JavaScript-Objekt *fadeByKey* mit der ViewModel-Eigenschaft *viewMode* verknüpft. Dabei stellt *fadeByKey* eine selbstgeschriebene Erweiterung des globalen kendo.binders-Objekts dar, das Zugriff auf das ViewModel hat, welches mit dem View-HTML-Element verbunden ist. Ändert sich die ViewModel-Eigenschaft *viewMode*, dann wird automatisch über einen internen Prozess durch Kendo UI die *refresh*-Funktion des *fadeByKey*-Objekts aufgerufen.

Diese fragt zunächst den Wert des *fadeByKey*-Objekts ab, der den Wert der *viewMode*-Eigenschaft referenziert. Anschließend wird das Datenattribut *key* aus dem HTML-Element abgerufen, welches auch die Verbindung zwischen *fadeByKey* und *viewMode* gesetzt hat. Die *refresh*-Funktion wird für alle HTML-Elemente ausgeführt, die diese Verknüpfung definiert haben, weswegen ein Vergleich in Zeile 35 zwischen dem *key*-Wert des HTML-Elements und dem aktuellen *viewMode*-Wert darüber entscheidet, ob die HTML-View in Zeile 37 ein- oder ausgeblendet werden soll.

Der Vorteil dieser über Kendo UI-Bindungen gesteuerte Ein- bzw-. Ausblendung von Views ist deren zentrale Ausführung von Animationsfunktionen in JavaScript. Sie müssen nur einmal definiert werden und alle HTML-Elemente, die damit verknüpft werden, führen sie aus. Die Navigation innerhalb des HR Portals ist demnach vollständig davon abhängig, welche Schaltfläche betätigt wurde, da nur in dem zugehörigen HTML-Element die für das Umschalten benötigte Information gespeichert ist.

Was ist aber, wenn zum Navigieren zu einer bestimmten View keine Schaltfläche betätigt werden kann, weil diese noch nicht sichtbar ist? Wenn die Suchergebnisse einer Volltextsuche zu passenden Views führen, dann kann genau dieser Fall eintreten, da die Suche von überall aus modulübergreifend möglich sein soll. Um dieses Problem zu lösen, ist eine veränderte Navigation notwendig, die unabhängig von zu betätigenden HTML-Elementen innerhalb der Anwendung ist. Ein Vorschlag für eine zugehörige Implementierung ist in Abschnitt 4.5 zu finden.

## Benutzerberechtigungen

Da über Solr sensible Benutzerdaten abgerufen werden können, sind diesbezüglich Vorkehrungen zu treffen, sodass jeder HR Portal-Anwender nur Zugriff auf die Daten besitzt, die seiner Anstellung im Unternehmen gerecht werden. Apache Solr bietet seit der Version 5.3 eine integrierte Lösung zur Authentifizierung und Autorisierung von Benutzern [14]. Diese Funktion ist dann sinnvoll, wenn eine Web-Anwendung direkt auf den Solr-Server zugreifen will, sodass dieser aus dem weltweiten Netzwerk heraus für jedermann zugänglich ist. Weil die WebAPI aber bereits eine Benutzerautorisierung bietet, ist eine zusätzliche von Seiten Solrs nicht notwendig, sodass nur sicher gestellt werden muss, dass die WebAPI Zugriff auf den Solr-Server hat, dieser aber nicht im offenen Internet zur Verfügung steht. Sie dient als Sicherheitsschranke zwischen dem HR Portal und dem Volltextsuche-Server.

Ein weiterer Vorteil, die WebAPI für die Steuerung des Benutzerzugriffs zu nutzen, ist die ebenfalls über sie laufende Prüfung, welche HR Portal-spezifische Benutzerrolle der Nutzer innehat und welche Daten er aufgrund dessen von anderen Arbeitnehmern sehen oder auch verändern darf. Dazu ist es im HR Portal möglich, beliebige Benutzerrollen anzulegen und diesen individuell verschiedene Zugriffsrechte zu geben, sodass entweder einzelne Module nicht sichtbar, nicht bearbeitbar aber sichtbar oder voll bearbeitbar sind, wenn sich ein Benutzer mit dieser ihm zugeordneten Rolle anmeldet. Damit ist es beispielsweise möglich, einer Führungskraft Lesezugriff auf benötigte Personaldaten der ihr unterstellten Arbeitnehmer zu gewähren, während die Mitarbeiter nur ihre eigenen Daten einsehen können.

Dies stellt jedoch ein Problem für die modulübergreifende Volltextsuche dar. Schließlich findet die bisherige Prüfung der Berechtigungen nur dann statt, wenn eine entsprechende Navigationsschaltfläche vom Hauptmenü (s. **Abbildung 7**) betätigt wird. Die Informationen für diese Prüfung sind in HTML-Attributen der jeweiligen HTML-Elemente zu den Schaltflächen gespeichert. Im Rahmen dieser Arbeit ist für die Berücksichtigung von Benutzerrollen innerhalb des HR Portals ein Konzept zu erarbeiten. Um genau zu verstehen, an welchen Stellen Anpassungen vorgenommen werden müssen, ist es notwendig, den bisherigen Prozess der Zugriffsprüfung zu untersuchen.

Wenn sich ein Anwender mit einem gültigen Benutzerkonto am HR Portal anmeldet, findet auf dem Server der WebAPI neben der Authentifizierung der Anmeldedaten eine Zuordnung des Benutzers zu einer entsprechenden Benutzerrolle statt. Die Zugriffsrechte dieser Rolle werden automatisch von der WebAPI berechnet und als Objekt in der PW-Datenbank serialisiert gespeichert. Sobald der Anwender eine Navigationsschaltfläche auswählt, werden die Zugriffsrechte zu einer bestimmten Benutzerrolle von der WebAPI abgefragt. Welche Daten das HR Portal von der WebAPI-Schnittstelle erhalten darf, entscheidet wiederum nur sie selbst bzw. die dahinter befindliche Geschäftslogik anhand der für den Benutzer bei der Anmeldung berechneten Rechteobjekte. Diese umfassen zwei Anwendungsgebiete. Zum einen legen sie fest, ob bestimmte Daten angelegt, gelesen, überschrieben oder gelöscht werden können. Solche werden im Folgenden als Objektrechte bezeichnet. Zum anderen definieren sie, welche Unternehmensbereiche einsehbar sein dürfen. Beispielsweise darf ein Vorgesetzter nur die Daten seiner unterstellten Mitarbeiter sehen, jedoch nicht eines anderen Vorgesetzten mit anderen Angestellten. Diese Art sei als Divisionsrecht benannt.

Um die Objektrechte bei einer Suchanfrage zu berücksichtigen und nur die Suchergebnisse anzuzeigen, die zu den Zugriffsrechten des angemeldeten Benutzers passen, bieten sich mindestens zwei Möglichkeiten an. Das HR Portal selbst kümmert sich darum oder die WebAPI liefert nur für den Nutzer gültige Ergebnisse. Für den ersten Fall bringt das HR Portal eine Funktion mit, die es erlaubt, das Anzeigen von Navigationsschaltflächen abhängig von den Objektrechten zu steuern. Dazu besitzen alle HTML-Elemente, die je nach Berechtigung anzuzeigen sind oder nicht, die CSS-Klasse *sagehr-security*. Deren Schlüssel-Wert-Paare, gespeichert in den Datenattributen *data-security-keys* und *data-security-values*, werden mit den von der WebAPI abgefragten Objektrechten verglichen, sodass daraufhin entschieden werden kann, ob die jeweilige Schaltfläche ein- oder ausgeblendet wird. Wenn sie angezeigt werden soll, wird die CSS-Klasse *sagehr-security* entfernt, da die verknüpfte CSS-Eigenschaft *display: none* nur für HTML-Elemente mit dieser Klasse in Kraft tritt.

Genau diese Funktionalität kann sich auch die Volltextsuche zunutze machen, sofern die gesamte HTML-Struktur des HR Portals bereits geladen ist. Denn dann kann beispielsweise die JavaScript-Komponente, die sich um die Suchabfrage und die Darstellung der Ergebnisse kümmert, zusätzlich jedes Suchergebnis herausfiltern, welches zu einer View mit der besagten CSS-Klasse gehört. Die Zuordnung von Suchergebnis zu entsprechender View ist schließlich gegeben. Ein gravierender Nachteil dieses Lösungsvorschlags ist jedoch das ungefilterte Senden von Suchergebnisdaten von der WebAPI an den Web-Browser. Selbst wenn diese dann ausgeblendet sind, besteht die Sicherheitslücke, indem der Netzwerkverkehr überwacht und ausgelesen werden kann und dabei sensible Daten noch vor der Behandlung durch das HR Portal einsehbar sind, ohne dass der Betrachter das Recht hat, diese auch sehen zu dürfen.

Daher ist der zweite Lösungsansatz zu bevorzugen, die WebAPI noch vor dem Senden der Suchergebnisse an den Client, eine Filterung dieser nach Benutzerrechten vornehmen zu lassen. Je unabhängiger die WebAPI dabei von der Anwendung bleibt, desto flexibler ist sie einsetzbar. Nur wie kann sie entscheiden, ob ein Suchergebnis zu den Benutzerrechten passt, ohne die HTML-Struktur des HR Portals zu kennen? Dazu könnte das HR Portal zu jeder Suchanfrage an die WebAPI eine zusätzliche Liste von JavaScript-Objekten mit dazugeben, die zu allen berechtigungsrelevanten HTML-Elementen deren Schlüssel-Wert-Paare bezüglich ihrer Objektrechte sowie den Dokumenttyp enthalten. Nur Suchergebnisse, die einen berechtigten Dokumenttyp aufweisen, werden daraufhin an den Client zurückgesendet.

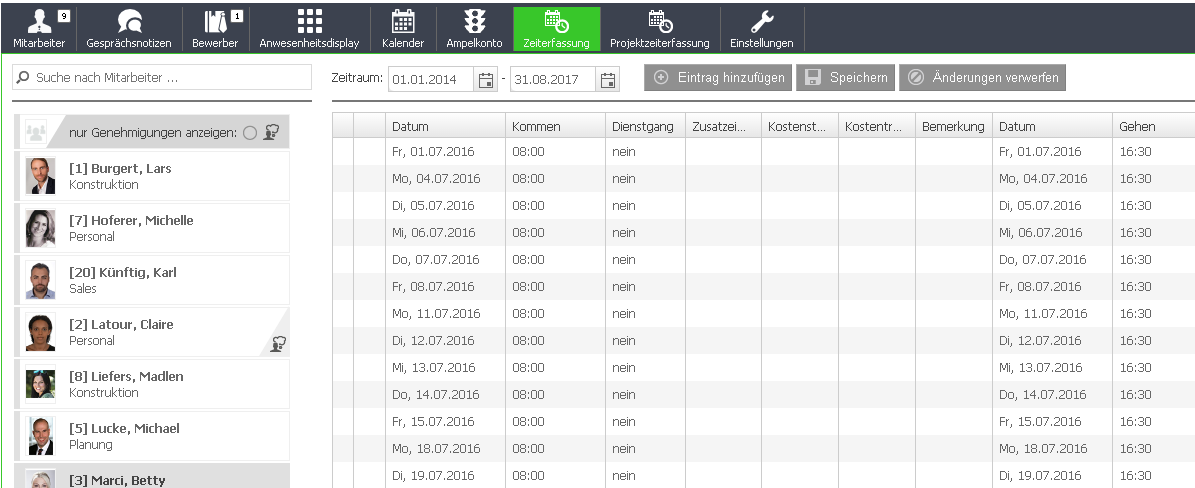
Die Divisionsrechte werden aktuell innerhalb des HR Portals nur an einer Stelle abgerufen und verwendet. Das Modul *Zeiterfassung* umfasst eine Detailansicht von Arbeitszeitdaten, siehe **Abbildung 9**. Beim Initialisieren des zugehörigen ViewModels werden die Divisionsrechte abgefragt, um festlegen zu können, ob der Benutzer Arbeitszeitdaten eines ausgewählten Arbeitnehmers hinzufügen, löschen oder bewilligen darf. Beim Laden von Arbeitnehmerlisten, wie z.B. links auf der genannten Abbildung zu sehen ist, ist im Gegensatz dazu die WebAPI nach einer Anfrage des HR Portals für das Herausfiltern von Arbeitnehmern verantwortlich. Dies geschieht u.a. anhand der Divisionsrechte des angemeldeten Nutzers während des Datenbankzugriffs innerhalb der PW-Geschäftslogik. Da das Laden der Zeiterfassungsansicht ohne weitere Abhängigkeiten zu Benutzereingaben automatisch passiert, muss für die Berücksichtigung der Divisionsrechte keine weitere Anpassung im HR Portal bei der Implementierung der Volltextsuche vorgenommen werden.

Abbildung 9: Ausschnitt aus der Detailansicht innerhalb des HR Portal-Moduls Zeiterfassung

## Erzeugung eines möglichst eindeutigen Primärschlüssels für Dokumente

Damit Dokumente nicht jedes Mal beim Indexieren neu hinzugefügt werden, müssen sie den eindeutigen Schlüssel *uniqueKey* besitzen. Wenn dieser bereits existiert, wird nur das betreffende Dokument ausgetauscht. Es gibt mehrere Möglichkeiten, einen solchen Schlüssel zu erzeugen. Es könnte fortlaufend ein Integer-Wert inkrementiert werden mit jedem zu indexierenden Dokument. Nur besteht dann die Gefahr, dass die Löschung von Quelldatensätzen die Reihenfolge dieser mit verändert, sodass völlig verschiedene Datensätze den selben uniqueKey erhalten würden und sich somit überschreiben. Um möglichst eineindeutige Schlüssel zu erzeugen, können auch die Daten selbst dafür benutzt werden. Dazu seien möglichst Daten verwendet, die zusammengesetzt das Dokument klar identifizieren. Als zusammengesetzten String-Wert sind sie dann in einen Primärschlüssel[[26]](#footnote-26) konvertierbar.

Mit Hilfe einer Hashfunktion ist es möglich, eine große Eingabemenge auf eine kleine Zielmenge abzubilden. Dadurch können beliebige Zeichenketten, die in all ihren Varianten einen unendlich großen Wertebereich darstellen, dennoch auf einen endlich kleineren Wertebereich reduziert werden, der sogar bei jeder Umwandlung die gleiche Länge aufweist. Ziel der Hashfunktion ist es zudem, dass sie unterschiedliche Eingabedaten auch zu unterschiedlichen Ausgabedaten führt, sodass es so selten wie möglich zu Kollisionen zwischen den Eingabedaten kommt, indem zu verschiedenen Eingaben der gleiche Hashwert herauskommt [15].

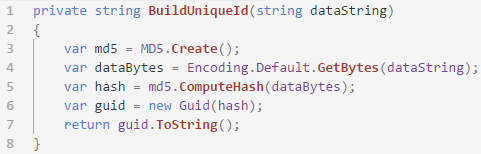
Da kein Zugriff auf den Solr-Server von außerhalb aus dem Netzwerk möglich ist, spielt die Sicherheit der Hashfunktion keine Rolle. Der Ausgabewert sollte lediglich lang genug sein, um Kollisionen zu vermeiden. Daher ist eine Kombination aus dem Hash-Algorithmus MD5[[27]](#footnote-27) und der Umwandlung des Hashwerts in eine GUID[[28]](#footnote-28) denkbar, die eine 128 Bit große Zeichenkette darstellt. **Abbildung 10** zeigt die verwendete Hilfsmethode *BuildUniqueId* im C#-Quelltext.

Abbildung 10: C#-Quelltext zur Generierung eines möglichst einzigartigen Dokumentenschlüssels anhand einer gegebenen Zeichenkette

Die .NET-Programmbibliothek *System.Security.Cryptography* bietet den MD5-Algorithmus, der in Zeile 3 instanziiert wird. Dieser benötigt in Zeile 5 als Argument einen Wert des Typs *Byte-Array*, welcher zuvor in Zeile 4 aus der Eingabezeichenkette gewonnen wurde, um ihn dann in einen Hashwert des Typs Byte-Array der festen Länge 16 umzuwandeln. Da 16 Byte 128 Bit ergeben, ist eine Konvertierung des Byte-Array-Werts aus der Hashfunktion in eine GUID-Zeichenkette möglich, wie die Zeilen 6 und 7 demonstrieren.

## Überlegungen zur partiellen Indexierung

Bei jedem Indexierungsvorgang den Index komplett neu aufzubauen, auch wenn sich nur wenige Daten seit der letzten Aktualisierung geändert haben, erscheint nur wenig sinnvoll, vor allem hinsichtlich der Dauer des gesamten Indexierungsprozesses. Größere Datenbanken mit einem Umfang von mehreren Gigabyte können dazu führen, dass selbst kleine Datenänderungen erst nach mehreren Minuten – wenn nicht sogar Stunden – für die nächste Suchabfrage zur Verfügung stehen.

Die Lösung für das Problem stellt die teilweise oder partielle Aktualisierung der Indexdatenbank dar. Um dies zu realisieren, bietet der DIH, welcher im Abschnitt 2.2.3 eingeführt wurde, die Möglichkeit, per *delta-import*–Kommando alle geänderten Daten zu aktualisieren. Jedoch ist diese Funktion nur für relationale SQL-Datenbanken verfügbar, weil ausschließlich der *SqlEntityProcessor*, der für die Indexierung der abgefragten Datenbanktabellen sorgt, den partiellen Import unterstützt.

Außerdem muss jede indexierte Tabelle eine zusätzliche Spalte mit dem letzten Änderungszeitpunkt jedes Datensatzes enthalten, damit dieser mit dem Zeitstempel des zugehörigen Dokuments verglichen werden kann. Liegt der Tabellenzeitstempel nach dem Dokumentzeitstempel, wird das gesamte Dokument gegen das neue Dokument ausgetauscht. Soll nicht das gesamte Dokument aktualisiert werden, so ist es seit der Solr-Version 4.0 möglich, über die Angabe von zusätzlichen Modifizierern, wie z.B. *set*, *add*, *remove*, nur einzelne Felder in bereits vorhandenen Dokumenten zu bearbeiten. Dazu ist es aber notwendig, dass sämtliche Felder nicht nur indexiert sondern auch gespeichert werden, was u.U. den Index unnötig anwachsen lässt.

Grundsätzlich gibt es zwei Möglichkeiten, eine Zeitstempelspalte innerhalb der Datenbank zu pflegen. Entweder kümmert sich der Datenbank-Server selbst bei jedem Schreibzugriff darum oder die jeweilige Anwendung, sobald sie Änderungen auf der Datenbank vornimmt. Der erste Fall kann über eine *Triggerfunktion* innerhalb des MS SQL Server gesteuert werden, die nach einem entsprechend ausgelösten Ereignis den Zeitstempel speichert. *Trigger* bringen jedoch einen Umgebungswechsel von *SQL* nach *PL/SQL*[[29]](#footnote-29) mit sich, welcher bei sehr häufigem Auslösen zu signifikanten Leistungseinbußen führen kann, abhängig von der Art des Triggers und von der Menge an betroffenen Tabellenzeilen [16].

Besteht dieses Problem, dann ist die zweite Möglichkeit geeigneter, indem die Zeitstempelaktualisierung in die jeweilige SQL-Anfrage der Anwendung integriert wird. Für die PW bedeutet das jedoch eine massive Nachpflege aller SQL-Befehle, die eine Datensatzänderung vornehmen, was deutlich aufwändiger und vor allem fehleranfälliger umsetzbar ist, als wenn der SQL-Server mit der Aktualisierung beauftragt wird. Demzufolge ist die Triggerfunktion vorzuziehen, sofern keine auffälligen Leistungseinbußen im SQL-Server feststellbar sind.

# Implementierung der Volltextsuche

Im Folgenden ist die Implementierung einer Volltextsuche in einem ausgewählten Bereich des HR Portals innerhalb der PW beschrieben. Vor allem die entscheidenden Schritte werden näher erläutert, die notwendig sind, um Solr erfolgreich für die Suche einsetzen zu können. Dabei kann nicht auf sämtliche Funktionsmöglichkeiten des Solr-Servers eingegangen werden, da diese Arbeit nicht den Anspruch einer Dokumentation erhebt. Es wird vielmehr der Bezug zur Anwendungsspezifik der PW hergestellt, sodass auch eine Hilfestellung für die vollständige Implementierung gegeben ist.

Der Ablauf der Implementierung wird in folgenden Unterkapiteln chronologisch dargestellt. Nachdem die technischen Voraussetzungen geklärt sind, stellt der Aufbau der Indexdatenbank einen großen Schwerpunkt beim Umgang mit Solr dar. Daraufhin wird auf die Suchabfrage eingegangen, um anschließend die Darstellung der Suchergebnisse zu beschreiben. Abschließend sei die Navigation vom Suchergebnis zur entsprechenden View beispielhaft erläutert und eine Möglichkeit analysiert, wie die modulübergreifende Navigation implementiert werden kann.

## Technische Infrastruktur

Da die Web-Anwendung HR Portal mit Hilfe des ASP.NET MVC 4.0 – Framework und der Programmiersprache C# in .NET 4.0 entwickelt wurde, bietet sich als Entwicklungsumgebung das Visual Studio von Microsoft an, welches während der Arbeit in der 2012er Version zur Verfügung steht. Auch die WebAPI ist in C# programmiert. Somit begrenzt sich die Auswahl der möglichen Betriebssysteme auf *Windows* von Microsoft begrenzt, welches in der Version 7 vorliegt.

Weil Apache Solr, welches aktuell die Version 6.6 aufweist, auf die Java-Programmbibliothek Lucene aufsetzt, kann für anwendungsspezifische Anpassungen die Entwicklungsumgebung Eclipse – Version Oxygen – genutzt werden, die problemlos das Einbinden des zugehörigen Open Source Projekts ermöglicht [17]. Wie der Apache Solr-Server eingerichtet werden muss, ist der offiziellen Dokumentation unter dem Abschnitt *Getting Started – Installing Solr* zu entnehmen [9].

Auf **Abbildung 11** ist zu sehen, wie die Startseite der Weboberfläche zur Konfiguration und Verwendung des Solr-Servers aussieht. Darauf wird außerdem ersichtlich, dass zu Testzwecken der Server lokal unter dem Standard-Netzwerkport 8983 erreichbar ist.

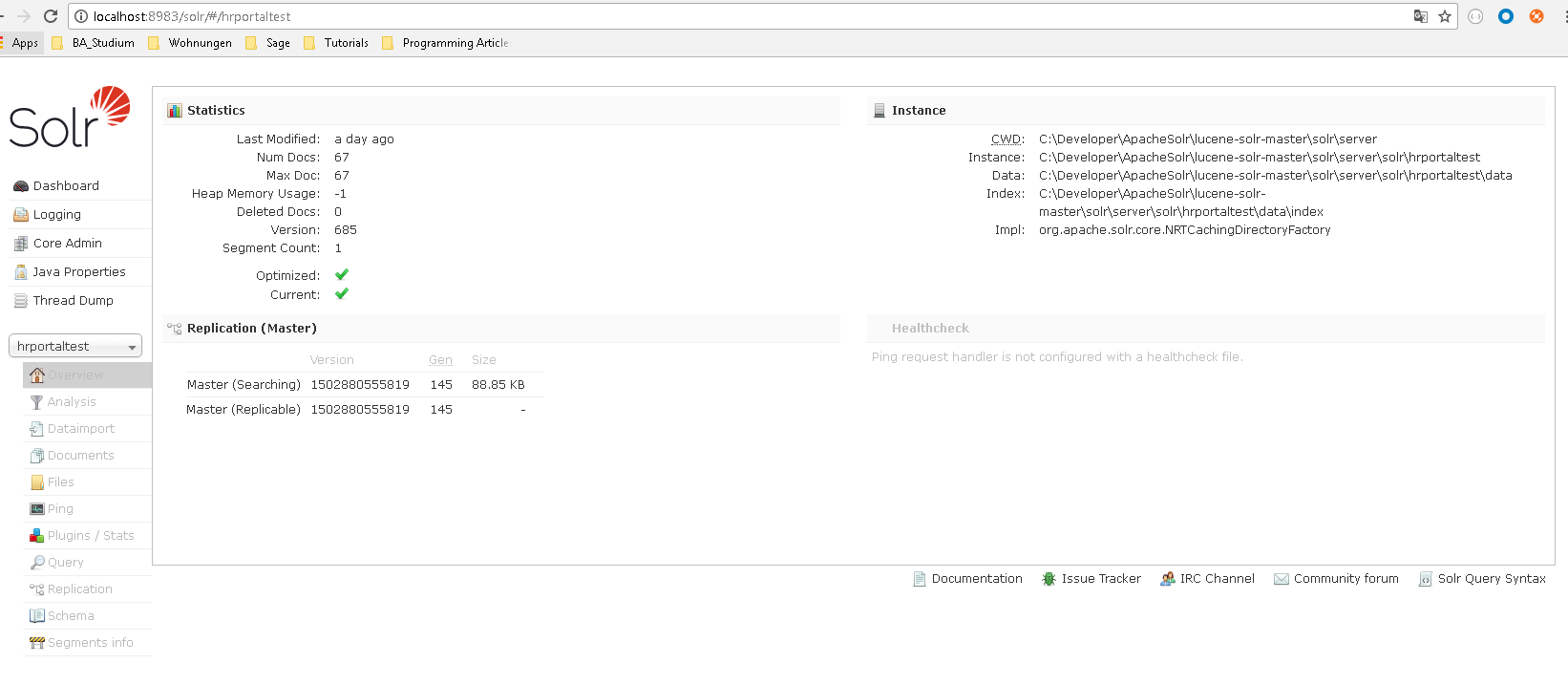
Die WebAPI ist eine reine Server-Anwendung, die ebenfalls lokal ausgeführt wird und in der Testumgebung den Netzwerkport 40001 besetzt, an den sämtliche Anfragen gesendet werden. Das HR Portal, wofür die PW-Version 2017.4.0 installiert ist, bringt ebenfalls eine eigene Server-Anwendung mit, mit der dann die Web-Benutzeroberfläche auf dem Client-System via HTTP kommunizieren kann.

Abbildung 11: Startseite der Weboberfläche zum Solr-Server

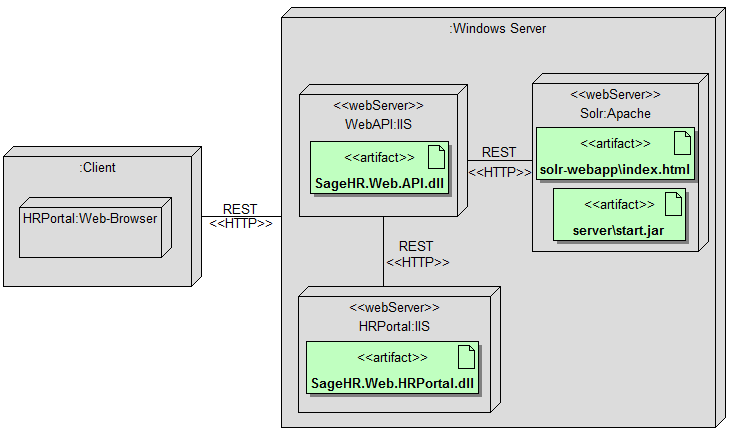
Während dieser Arbeit wird der HR Portal-Server auf dem gleichen Computer wie die WebAPI und Solr ausgeführt – unter dem Port 40003. Wie die Verteilung der einzelnen Komponenten aussehen kann, wenn sie in einem Produktivsystem verwendet werden sollen, veranschaulicht das UML-Verteilungsdiagramm[[30]](#footnote-30) auf **Abbildung 12**.

Abbildung 12:UML-Verteilungsdiagramm zur Client-Server-Architektur des Produktivsystems HR Portal mit Volltextsuche

Daraus wird ersichtlich, dass die einzelnen Server-Komponenten mit der WebAPI-, dem HRPortal- und dem Apache Solr-Server auf einem Windows-Server-System installiert werden können. Jedoch sind nur die Web-Server der WebAPI und des HR-Portals für Client-Systeme aus dem Internet via REST-Schnittstelle über das HTTP-Protokoll ansprechbar. Wie bereits in Kapitel 3.2 erläutert, darf aus Sicherheitsgründen nur die WebAPI auf den Solr-Server über REST-konforme HTTP-Anfragen zugreifen. Der Client kann sowohl direkt mit Hilfe von AJAX per HTTP von der WebAPI Daten anfordern oder indirekt, indem er Views ebenfalls über REST-konforme HTTP-Anfragen an den HR Portal-Web-Server stellt, der wiederum per HTTP Daten von der WebAPI abfragt.

Die im Verteilungsdiagramm zu sehenden Artefakte sind nur die Hauptausgabedateien der einzelnen Server-Systeme. Deren Abhängigkeiten sind aufgrund der Übersichtlichkeit nicht mit aufgeführt. Solrs Web-Server von Apache bietet zum einen für die Web-Benutzeroberfläche eine *index.html*-Datei samt zugehörigen CSS- und Javascript-Dateien. Zum Starten des Web-Server wird eine *start.jar*-Datei zusammen mit der Lucene-Java-Bibliothek und den Solr-Java-Paketen ausgeliefert. Die WebAPI sowie das HR Portal stellen u.a. jeweils eine eigene *DLL*-Datei[[31]](#footnote-31) zum Ausführen der Anwendungen durch den *IIS*-Web-Server[[32]](#footnote-32) zur Verfügung.

## Indexierung der Daten

Nachdem Solr erfolgreich installiert wurde, ist die zugehörige Indexdatenbank zu installieren, um diese anhand von Suchanfragen durchsuchen zu können. Zuvor müssen jedoch die zu indexierenden Daten geklärt werden. Da für die Umsetzung des Prototyps die zu durchsuchenden Bereiche stark eingeschränkt sind, kann sich auf die wenigen zugehörigen Views im HR Portal konzentriert werden.

### Auswahl der zu indexierenden Daten

Die in Abschnitt 1.4 festgelegten Anforderungen an den Prototyp sehen u.a. vor, dass ein Benutzer mit Hilfe der Volltextsuche zu bestimmten Views gelangen kann. Dazu ist es notwendig, dass mit den Ansichten entsprechende Schlagwörter verknüpft sind, die möglichst viele potentielle Suchwörter darstellen, um die Volltextsuche intuitiver werden zu lassen. Es gibt zwei Möglichkeiten, wie solche Wörter zum Indexieren gefunden werden können. Entweder legt der Entwickler der Ansicht sie fest oder ein Programm durchsucht die View nach dort befindlichen Texten und übergibt sie dem Indexierungsprozess.

Der erste Fall führt zu mehr Aufwand, da Entwickler sich neben der Gestaltung der Views auch Gedanken zu treffenden Stichwörtern machen und diese dann manuell in das Programm einpflegen müssen. Im zweiten Ansatz wäre das Heraussuchen und Einpflegen automatisiert, jedoch besteht das Problem, dass in der Ansicht Wörter existieren können, die semantisch nichts mit dem Thema zu tun haben müssen und dennoch mit indexiert werden. Wer oder was entscheidet dann, was davon für die Volltextsuche relevant ist? Ein weiterer Vorschlag für eine teilweise automatisierte Schlagwortauswahl wird in Kapitel 5.1 diskutiert. Es sei an dieser Stelle nur so viel gesagt, dass sich für die erste Variante der manuellen Schlagwortdefinition entschieden wurde, um nicht den Rahmen dieser Arbeit zu sprengen. Außerdem sind für den Prototyp nur wenige Views zu berücksichtigen, sodass sich die manuelle Pflege in Grenzen hält.

Zu den wenigen Views zählt die arbeitnehmerabhängige Kalenderübersicht für Vorgesetzte. Je nachdem welcher ihm zugeordnete Arbeitnehmer aus der Liste ausgewählt wurde, können dessen Resturlaubsübersicht, die für diesen Mitarbeiter zu erledigenden Aufgaben sowie die Erfassung seiner Fehlzeiten der Führungskraft angezeigt werden. Um die zu den Views zugehörigen definierten Schlagwörter mit dem jeweiligen Arbeitnehmer zu verknüpfen, ist es notwendig, dass von der WebAPI alle vom Benutzer einsehbaren Arbeitnehmer abgefragt werden. Daraufhin kann für jeden Arbeitnehmer pro View ein eigenes Dokument mit dessen vollständigem Namen sowie Arbeitnehmerschlüssel, bestehend aus Mandanten- und Arbeitnehmernummer, angelegt und indexiert werden. Der Benutzer sucht mit großer Wahrscheinlichkeit anhand dieser Daten in Kombination mit den Schlagwörtern, um zu entsprechenden Views zu gelangen.

Für die zweite Anforderung, Bewerberdokumente indexieren zu können, sind mehrere Schritte notwendig. Zunächst müssen die Daten zu sämtlichen gespeicherten Bewerbungen von der WebAPI abgefragt werden. Die zurückgegebenen XML-Daten enthalten dann zu einer Ausschreibung den Bewerbernamen, die eindeutigen Nummern des Bewerbers und der Bewerbung. Für jeden Bewerber gilt es daraufhin, sämtliche XML-Daten seiner Bewerbungsdokumente von der WebAPI anzufragen und sich die Bewerbungsdokumentnummern zu merken. Erst mit Hilfe dieser Nummern sind alle seine Bewerbungsdokumente als Binärdaten von der WebAPI anforderbar. Zusätzlich sind auch der Ausschreibungsschlüssel sowie der Titel der Ausschreibung mit zu indexieren, die bereits im ersten Schritt mitkommen.

Zur eindeutigen Zuordnung eines Dokuments zu einer View ist zu jedem Dokument ein zusätzliches Feld anzulegen, das den Dokumenttyp beschreibt. Der Typ muss eine eindeutige Zuordnung von Dokument zu bestimmten Views gewährleisten, um die Navigation später innerhalb der Anwendung zu vereinfachen. Mehr dazu ist in Abschnitt 4.5 erläutert. Eine Übersicht über alle benötigten Dokumentfelder ist in Abschnitt 4.2.3 aufgelistet.

### Definition einer eigenen Datenquelle für die WebAPI

Um, wie in Abschnitt 3.2 erläutert, Daten von der WebAPI automatisch abzufragen, gilt es für die Prototypentwicklung zwei eigene Java-Klassen vom Typ *DataSource* im Java-Projekt des DIH anzulegen. Zur Verwendung von Datenquellen aus dem Netzwerk über HTTP-Anfragen existiert bereits standardmäßig die DataSource-Klasse *URLDataSource*. Durch Kopieren und stellenweises Anpassen dieser Klasse entsteht daraus eine neue Klasse namens *SageHRWebAPIDataSource* für die Abfrage von Daten, die strukturiert als Datensätze in der PW-Datenbank vorliegen. Unstrukturierte Daten, wie z.B. Bewerberdokumente, die im Binärformat serialisiert in der Datenbank gespeichert sind, benötigen eine DataSource-Klasse für Tika, die nur die Binärdaten ausliest. Für Datenabfragen über HTTP-Anfragen gibt es dafür standardmäßig die Klasse *BinURLDataSource*, die für den Zugriff auf die WebAPI entsprechend geändert und in einer neuen Klasse namens *SageHRWebAPITikaDataSource* angelegt wird.

Bei beiden Klassen ändert sich exakt das Gleiche, indem in der Initialisierungsmethode die Authorisierung der WebAPI abgefragt wird. Jedoch können ohne weitere Anpassungen in den Kernkomponenten von Solr keine zusätzlichen Parameter definiert werden, die über die Konfigurationsdatei übermittelbar sind. Daher werden sämtliche für die Authentifizierung notwendigen Parameter über die Basis-URL-Zeichenkette übermittelt, wie es Abschnitt 4.2.2 veranschaulicht. Mit Hilfe spezieller Platzhalter werden die Parameter extrahiert und für die HTTP-Anfrage verwendet.

Nach erfolgreicher Anmeldung gibt die WebAPI innerhalb eines *Cookies* der HTTP-Antwort eine Zeichenkette als Sitzungsschlüssel mit, die in der *getData*-Methode im Kopf der Datenanfragenachricht verpackt an die WebAPI zur Validierung der Anfrage mitgeschickt werden muss. Die Antwort wird entweder als fortlaufender Text bei ersterer Klasse oder als Binärdatenstrom bei der Tika-spezifischen Klasse zurückgegeben, sodass der jeweils zugehörige EntityProcessor die Indexierung anhand der Daten vornehmen kann, worauf im nächsten Abschnitt näher eingegangen wird. Der vollständige Code der beiden DataSource-Klassen ist in Anhang 11.2 abgebildet.

### Konfiguration des Data Import Handler

Damit der in Abschnitt 2.2.3 eingeführte Data Import Handler (DIH) zum automatisierten Indexieren der PW-Daten genutzt werden kann, gilt es ein paar Vorkehrungen zu treffen. Zunächst muss das zugehörige Java-Paket als Ressource in der Hauptkonfigurationsdatei von Solr namens *solrconfig.xml* angegeben werden, um ausgeführt werden zu können. Folgende Zeile ist dort zu den anderen Bibliothekspfaden hinzuzufügen.



Mit Hilfe einer Parameterangabe, die mit *${…}* eingehüllt wird, kann über den Variablennamen *solr.install.dir* der Pfad zum Installationsverzeichnis von Solr angegeben werden. Dank des regulären Ausdrucks hinter dem Attribut *regex* ist der Pfad der JAR-Datei auch versionsunabhängig anzeigbar.

Damit der DIH über die REST-Schnittstelle des Solr-Server gestartet werden kann, muss er in die solrconfig.xml–Datei als *RequestHandler* eingetragen werden. **Abbildung 13** zeigt den Eintrag für den Prototyp. Darin ist zum einen unter der Eigenschaft *config* der Dateipfad zur Konfigurationsdatei des DIH anzugeben - siehe Zeile 4. Da der Prototyp zum Indexieren von Dokumenten, wie PDF- oder Word-Dateien, den *TikaEntityProcessor* von Solr verwendet, ist noch zusätzlich über die Eigenschaft *map.content* in Zeile 5 der Name des Dokumentfelds zu übergeben, in dem der extrahierte Text gespeichert wird. Die Verwendung von *Tika* wird weiter unten in diesem Abschnitt erläutert.

Abbildung 13: Eintrag des DIH als RequestHandler in die Hauptkonfigurationsdatei von Solr

Um mit Hilfe des DIH Daten auslesen und indexieren zu können, müssen in dessen Konfigurationsdatei eine oder mehrere Datenquellen sowie der Inhalt der Dokumente definiert werden. Über das XML-Tag *dataSource* wird eine Datenquelle festgelegt. Je nach Art der DataSource-Klasse sind verschiedene Eigenschaften anzugeben. Gemeinsam haben alle DataSource-Klassen die Eigenschaften *name* zur eindeutigen Identifizierung und *type* zur Angabe des zugehörigen Java-Klassennamens. Da beide verwendeten Datenquellen über URLs erreichbar sind, ist als weiterer Parameter die Basis-URL anzugeben, damit sämtliche Anfragen durch relative URLs ausgedrückt werden können.

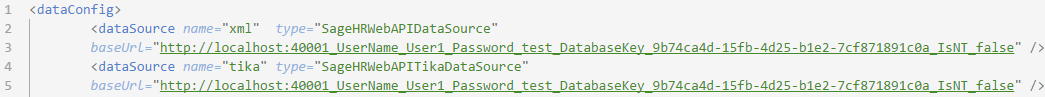
**Abbildung 14** zeigt den zugehörigen Auszug aus der DIH-Konfigurationsdatei. Wie im Abschnitt 4.2.2 beschrieben, werden sämtliche Parameter zur Authentifizierung an der WebAPI über die baseUrl-Zeichenkette übermittelt. Mit den Platzhaltern *„\_UserName\_*“ und *„\_Password\_*“ werden die die Anmeldedaten angegeben. *„\_DatabaseKey\_“* steht für den Schlüssel zur Identifizierung der verwendeten PW-Datenbank und *„\_IsNT\_“* sagt aus, ob es sich um eine Windows-Anmeldung handelt, wodurch die Anmeldedaten des PC-Benutzers geprüft würden.

Abbildung 14: Definition der Datenquellen innerhalb der DIH-Konfigurationsdatei

Ein Dokument wird über sogenannte Entitäten definiert, ähnlich wie eine Tabelle innerhalb einer relationalen Datenbank. Jede Entität kann völlig eigenständig und unabhängig Daten aus einer bestimmten Quelle speichern. Zudem ist es durch die Baumstruktur einer XML-Datei möglich, Entitäten beliebig weit ineinander zu verschachteln, sodass die Kind-Entitäten auf Daten ihrer Eltern-Entitäten zugreifen und für deren Anfragen verwenden können. Egal wie tief der dadurch aufgespannte Baum wird, das resultierende Dokument besteht aus nur einer Ebene, die sich aus den in allen Entitäten definierten Feldern zusammensetzt. Die flache Daten-Hierarchie ist bei Solr-Dokumenten eine Grundbedingung [7 S. 6]. Bevor jedoch die Felder im Dokument festgelegt werden können, müssen diese in der Schema-Datei, die in Abschnitt 2.1.1 beschrieben ist, definiert werden.

Für die Definition von Dokumenten bezüglich der Ansichten innerhalb der Kalenderübersicht ist exemplarisch die Entität für die Resturlaubsansicht dargestellt und erläutert. Die weiteren Kalenderentitäten unterscheiden sich nur im Dokumenttyp und in der URL zum Abrufen der zugehörigen Schlagwörter.

Zeile 5 bis 10 auf **Abbildung 15** definieren die Parameter der Entität. Über den Schlüssel *name* erhält jede Entität einen eindeutigen Namen, um auf sie aus anderen Entitäten heraus zugreifen zu können. Die Eigenschaft *dataSource* legt fest, welche von denen anfangs definierten Datenquellen für die Befüllung der Felder mit Daten abfragbar ist. Der Parameter *url* definiert die zur Basis-URL relative HTTP-Adresse, mit dessen Hilfe die Datenquelle gezielt ausgelesen wird. Der Einfachheit halber ist darauf verzichtet worden, außerhalb der selbst angelegten DataSource-Klassen Anpassungen an weiteren Kernklassen von Solr zu unternehmen, die eine saubere Parametererweiterung der getData Methode erlauben. Um dennoch die zusätzliche Information, welche HTTP-Methode zum Abfragen der WebAPI verwendet werden soll, von der Konfigurationsdatei aus an die Methode zu übergeben, kann der url-String getrennt durch den unverkennbaren Platzhalter *\_HTTPMETHOD\_* um den Namen der HTTP-Methode erweitert werden.

Abbildung 15: Definition eines Dokuments in DIH-Konfigurationsdatei für Suche der Resturlaubsansicht

Der Parameter *entityProcessor* gibt den Namen der EntityProcessor-Klasse an, die bereits in Abschnitt 2.2.3 eingeführt wurde und ermöglicht mit Hilfe des speziellen *XPathEntityProcessor* den Zugriff auf bestimmte Knoten über XML-Pfade entlang der Baumstruktur. Ein solcher Pfad wird bereits vom Parameter *forEach* danach verwendet und zeigt auf den Namen von Listenelementen innerhalb des von der WebAPI zurückgegebenen XML-Dokuments. Diese XML-Liste wird in einer Schleife einzeln durchgegangen und für jedes Element ein eigenes Dokument angelegt. Dadurch kann pro Arbeitnehmer ein eigenes Resturlaubsdokument erzeugt werden.

Zu guter Letzt ist es über die Entitätsparameter möglich, verschiedene *Transformer*-Klassen zu nutzen, um weitere Operationen auf die definierten Felder anzuwenden. Im Rahmen dieser Arbeit ist die *TemplateTransformer*-Klasse verwendet worden, um dem Feld *documentType* über das Attribut *template* einen fixen Wert mit dem für die Navigation im HR Portal wichtigen Dokumenttyp zuzuweisen (siehe Zeile 11). Außerdem nutzt auch das Feld *dataString* aus Zeile 14 den TemplateTransformer, um bereits in anderen Feldern gespeicherte Daten zu einem String zusammenzufassen. Dieser wird zur Berechnung des Primärschlüssels benötigt, welche in Abschnitt 3.5 erklärt wird.

Am Beispiel des Felds dataString wird außerdem noch eine weitere nützliche Funktion der DIH-Konfigurationsdatei deutlich. Wie in diesem Abschnitt zuvor schon einmal gesehen, kann über das Symbol *$* in geschweiften Klammern eine Variable verwendet werden. In diesem Fall wird auf die beiden Felder *employeeFullname* und *employeeKey* über die Angabe des zugehörigen Entitätnamens und, durch einen Punkt getrennt, des jeweiligen Feldnamens zugegriffen.

Nachdem von der WebAPI alle in der PW-Datenbank enthaltenen Arbeitnehmer in Form eines XML-Dokuments an den Solr-Server zurückgegeben wurden, speichert laut Definition in Zeile 12 und 13 der DIH die über den *xpath*-Parameter angegebenen Arbeitnehmerdaten in den Feldern *employeeFullName* und *employeeKey*. Erst dann wird zur Erzeugung der Kind-Entität namens *SearchKeywords* eine weitere WebAPI-Anfrage abgesetzt, um den Primärschlüssel des Dokuments im Feld *documentId* und die zum Thema zugehörigen Schlagwörter im Feld *searchKeywords* innerhalb des Gesamtdokuments zu speichern. Anhand des Kind-Entität-Parameters *pk*, der das Feld für den Primärschlüssel angibt, wird deutlich, dass es sich dabei um eine flaches Dokument handelt. Denn es ist völlig unerheblich, wie tief verschachtelt die Entitäten sind. Wenn der Primärschlüssel aus einer tieferen Ebene erst bestimmt wird, gilt er trotzdem für das gesamte Dokument. Die vollständige Konfigurationsdatei des DIH ist im Anhang 11.3 abgebildet.

### WebAPI-Schnittstelle für Datenanfrage zur Indexierung

Zwei Gründe gibt es, warum zur Umsetzung des Prototyps eine neue WebAPI-Schnittstelle sinnvoll ist. Zum einen werden Schlagwörter zur Abdeckung von möglichen Sucheingaben benötigt, die der DIH mit indexiert. Da der Benutzer das HR Portal in der deutschen oder englischen Sprache sehen kann, müssen auch die Suchwörter mehrsprachig berücksichtigt werden. Dazu existiert bereits innerhalb der Geschäftslogik der PW eine eigene Komponente namens *SageHr.Localization* für die Lokalisierung[[33]](#footnote-33) von Texten. Die WebAPI hat Zugriff auf diese und ermöglicht deren Abfrage über HTTP-Anfragen.

Zum anderen sind die indexierten Dokumente mit einem Primärschlüssel auszustatten, der für die dort neu zusammengesetzten Daten noch nicht existiert. Die Generierung des Schlüssels, wie es Abschnitt 3.5 vorschlägt, könnte direkt von der abgeleiteten DataSource-Klasse in Solr übernommen werden. Jedoch ist dazu das Durchsuchen der XML-Daten von der WebAPI notwendig, um nur bestimmte Daten für die Schlüsselerzeugung zu berücksichtigen. Die Möglichkeit XML-Daten Knoten für Knoten auszulesen, ist aber erst innerhalb der XPathEntityProcessor-Klasse möglich, da diese alles dazu bereits mitbringt. Und den Code dieser Klasse anzupassen, ist deutlich aufwändiger, als eine separate Methode innerhalb der PW-Geschäftslogik neu anzulegen, die sich nur um die Schlüsselerzeugung kümmert.

Ohne zu sehr ins Detail einzugehen, besteht die WebAPI aus abgeleiteten Klassen der ApiController-Klasse aus der .NET-Assembly[[34]](#footnote-34) *System.Web.Http*. Diese Klassen heißen auch *Controller* und können direkt über HTTP-Anfragen aufgerufen werden. Für jede spezifische Anfrage ist im Controller eine eigene Methode definiert, die den Namen der HTTP-Methode trägt, mit der sie aufgerufen wird. Die Argumente der Methode können über die URL als Parameter übergeben werden. **Abbildung 17** zeigt die *GET*-Methode, um zusätzliche Daten abzufragen, die für die Volltextsuche vom DIH benötigt werden.

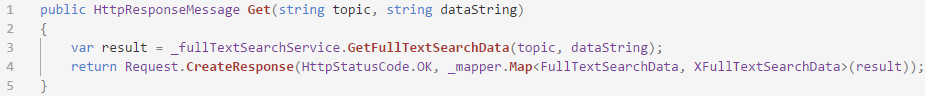
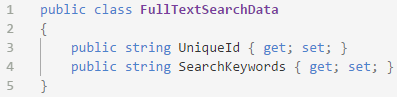
Als Antwort liefert die Methode in Zeile 4 den HTTP-Statuscode 200 für *OK* und ein Objekt namens *FullTextSearchData*, das mit Hilfe des Werkzeugs *Automapper* repräsentiert durch das Klassenattribut *\_mapper* in das DTO[[35]](#footnote-35) *XFullTextSearchData* konvertiert wird.

Abbildung 16: DTO-Klasse für Datenobjekt zur Indexierung

Abbildung 17: WebAPI-Controller-Methode zum Abfragen von Daten zur Indexierung

Das DTO und das zugehörige Geschäftslogikobjekt, dessen Definition auf **Abbildung 16** dargestellt ist, sind in diesem Fall identisch. Jedoch wird der Konvention gefolgt, dass Objekte, die innerhalb der Geschäftslogik verwendet werden, nicht direkt von der WebAPI sondern über reine Datenträgerobjekte, den DTOs, weitergeleitet werden [18].

Innerhalb der Controller-Methode wird gemäß Konvention keine Logik ausgeführt. Die empfangenen Argumente leitet die Methode direkt weiter an die Klasse *FullTextSearchService* innerhalb der Geschäftslogik, welche die Methode *GetFullTextSearchData* zur eigentlichen Bearbeitung anbietet, die die angeforderten Daten liefert. Wie auf **Abbildung 18** in Zeile 5 nachvollzogen werden kann, ruft sie anhand des vom Client angegebenen Themaschlüssels *topic* die Schlagwörter von der zuvor erläuterten Lokalisierungs–Komponente ab.

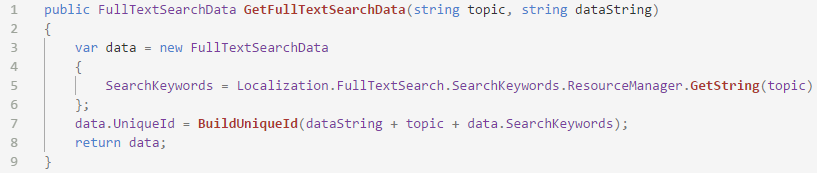
In Zeile 7 wird anhand eines zusammengesetzten Strings aus dem in der jeweiligen Entität definierten Daten-Strings, dem Thema und den zuvor erhaltenen Suchbegriffen eine annähernd einzigartige Nummer erzeugt, so wie es bereits in Abschnitt 3.5 ausführlich beschrieben ist. Das neu instanziierte Datenobjekt wird an den Aufrufer der Methode zurückgegeben (siehe Zeile 8) und landet letztendlich über den WebAPI-Controller – dank des im HTTP-Anfrage-Kopf festgelegten *Content-Type* *application/xml* – im XML-Format in der getData-Methode des anfragenden DataSource-Objekt, von wo aus es über den entsprechenden EntityProcessor mit indexiert wird.

Abbildung 18: Methode aus der *FullTextSearchService*-Klasse zur Beschaffung der angeforderten Daten für die Indexierung

## Benutzereingabe zur Volltextsuche

Die Volltextsuche ist perspektivisch modulübergreifend ausgerichtet. Daher ist auch die Benutzeroberfläche zum Suchen und Anzeigen von Suchergebnissen unabhängig von einzelnen Modulen zu halten. Erreicht werden kann dies entweder durch eine komplett neue Ansicht, die über eine neue Navigationsschaltfläche ansteuerbar ist oder die Suchelemente werden in die Hauptseite des HR Portals integriert. Da der Fokus dieser Arbeit auf der Machbarkeitsanalyse einer Volltextsuche liegt, wird darauf verzichtet, eine optisch ansprechende und aufwändige Lösung dafür zu entwickeln. Entscheidend ist, dass über ein Textfeld Suchbegriffe eingegeben werden können und die vom Solr-Server erhaltenen Suchergebnisse adäquat und transparent dargestellt sind. Über eine Verlinkung erfolgt dann die Navigation zur entsprechenden View innerhalb des HR Portals.

Weil die Suchfunktion sofort nach der Anmeldung des Benutzers zur Verfügung steht, wird kein weiterer Controller innerhalb des HR Portals benötigt, der die neue View ausliefert. Die Start-View *Layout.cshtml* stellt das Grundgerüst der Web-Anwendung dar und kann durch Platzhalter markiert andere Views aus jeweils eigenen *cshtml*-Dateien einbinden. Beispielsweise wird so auch das Navigationsmenü integriert und sofort geladen. Da oberhalb dieses Menüs noch freier Platz ist, bietet sich dieser für das Eingabefeld der Volltextsuche an. **Abbildung 19** zeigt das fertige Resultat der Eingabe.

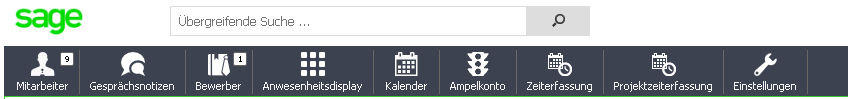
Auf eine ausführliche Darstellung und Erläuterung des Quelltexts zu diesem Element der Suche wird in dieser Arbeit verzichtet, da das bloße Anzeigen der Volltextsuchfunktion beliebig änderbar ist und keine Aussage darüber zulässt, inwiefern eine Volltextsuche innerhalb des HR Portals realisierbar ist. Anhang 11.4 zeigt dazu die einzelnen Dateiinhalte.

Abbildung 19: Benutzeroberfläche der prototypsartigen Sucheingabe im HR Portal

Entscheidender ist die Frage, was passiert, nachdem der Benutzer die Suchschaltfläche betätigt hat? Zunächst wird die vom Mausklick ausgelöste Ereignisbehandlungsfunktion in JavaScript ausgeführt, die auf **Abbildung 20** zu sehen ist. Diese fragt in Zeile 2 den aktuell eingegebenen Text aus dem Sucheingabefeld ab und sofern etwas eingegeben wurde, leitet die Funktion den Text über eine HTTP-Anfrage an die GET-Methode des *FullTextSearchController* zum Abfragen von relevanten Suchergebnissen weiter. Zusätzlich zum Suchtext werden durch das *&*-Zeichen getrennt weitere Parameter angegeben, die die Auswahl der Ergebnisse einschränken können. Mit Hilfe der Festlegung von *startIndex* und *maxCountResults* fragt die WebAPI die ersten zehn der vom Solr-Server ermittelten Suchergebnisse ab. Natürlich ist es sinnvoll die beiden Parameter variabel zu halten, um u.a. auch weitere Ergebnisse anzeigen zu können, jedoch ist dies nicht explizit gefordert und bedarf keiner weiteren Berücksichtigung für die Umsetzung des Prototyps.

Die daraufhin aufgerufene GET-Methode des FullTextSearchController ist in Anhang 11.6 abgebildet - genauso wie die vollständige Geschäftslogik zum Senden einer Suchanfrage an den Solr-Server und das anschließende Auslesen der Server-Antwort. Wichtiger ist es zu klären, wie die konkrete Suchanfrage an den Solr-Server aussieht. Über die Solr-REST-Schnittstelle namens *select* kann eine neue Suchanfrage abgesetzt werden, die verschiedene Parameter enthalten darf, wie auf **Abbildung 21** nachzuvollziehen ist. Die Parameter stellen nur eine für die Umsetzung des Prototyps sinnvolle Auswahl aller möglichen dar. Hier sei auf den Abschnitt *Searching* der offiziellen Dokumentation von Solr verwiesen [9].

Abbildung 20: JavaScript-Funktion zum Behandeln von Mausklickereignissen nach Betätigung der Suchschaltfläche

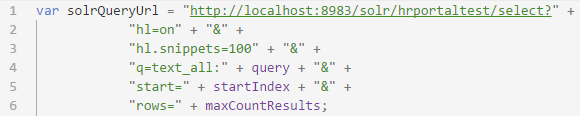


Abbildung 21: URL für Suchanfrage an den Solr-Server

Mit *hl=on* in Zeile 2 wird festgelegt, dass neben den gefundenen Dokumenten auch die Textstellen zurückgegeben werden sollen, in denen die Suchbegriffe existieren. Über *hl.snippet=100* in der nächsten Zeile werden nur die ersten 100 gefunden Textstellen von jedem Suchergebnis abgefragt, sofern so viele vorhanden sind. Die eigentliche Suchanfrage steckt im Parameter *q* in Zeile 4, welcher in diesem Fall von allen indexierten Dokumenten nur das *text\_all*-Feld nach den Suchbegriffen aus der Benutzereingabe *query* durchsuchen lässt. Mit *start* und *rows* in Zeile 5 und 6 werden wie schon zuvor beim WebAPI-Aufruf der Index des ersten Suchergebnis und die Anzahl der zurückzugebenden Ergebnisse festgelegt. Dabei ist zu beachten, dass die Suchergebnisse vom Solr-Server bereits nach ihrer Relevanz geordnet werden, mit dem am meisten relevanten als erstes Listenelement.

## Auswahl und Darstellung der Suchergebnisse

Nachdem der Nutzer einen oder mehrere Suchbegriffe nur durch Leerzeichen getrennt eingegeben und die Suchschaltfläche mit dem Lupensymbol betätigt hat, erscheint die Ansicht mit den Suchergebnissen. Dazu erscheint eine Oberfläche, die das gesamte HR Portal überdeckt und über Schaltflächen in der rechten, oberen Ecke entweder ein- bzw. ausklappbar oder gänzlich schließbar ist. In dem nun sichtbaren weißen Rechteck sind die ersten zehn relevantesten Suchergebnisse als geordnete Liste angezeigt, deren Elemente aus einem Titel und näheren Details zum Suchergebnis sowie den gefundenen Textstellen bestehen. Der Ergebnistitel ist zusätzlich eine Verlinkung, um per Weiterleitung die zum Titel zugehörige View einzublenden. Auf **Abbildung 22** ist ein Ausschnitt dieser Ansicht für die Suchanfrage „*Urlaub Bewerbung*“ zu sehen. Die komplette HR Portal-Ansicht zu den Suchergebnissen ist im Anhang 11.5 abgebildet.

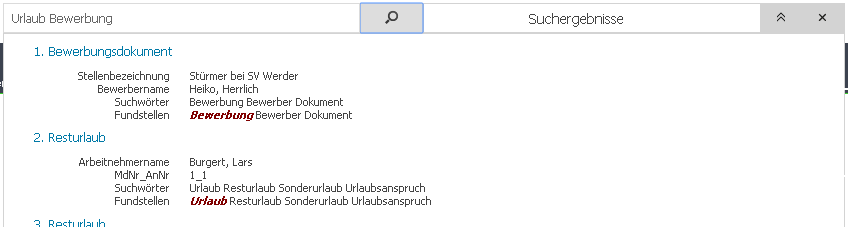
Ein Beispielsuchergebnis für das Durchsuchen von Bewerberdokumenten ist aus Platzgründen im Anhang 11.5.2 abgebildet. Es demonstriert, wie Auszüge aus dem Volltext zusammen mit den markierten Fundstellen aussehen können und beweist damit die Machbarkeit der gestellten Anforderung, Volltextdokumente im HR Portal indexieren und durchsuchen zu können.

Abbildung 22: Ausschnitt aus Ansicht der Ergebnisse der Volltextsuche

Ein Hauptaugenmerk bei der Implementierung der Suchergebnisansicht liegt in der Darstellung der Details. Die gesamte Ergebnisliste samt Details wird dynamisch mit Hilfe von JavaScript erzeugt, sodass es kein starres HTML-Konstrukt für diese Ansicht gibt. Wenn für jeden Dokumenttyp explizit definiert wäre, welche Details anzuzeigen sind, würde dies einen entsprechend großen Pflegeaufwand bedeuten – bei einer Anzahl von mindestens 100 verschiedenen Dokumenttypen. Daher wird das im Dokument gespeicherte, mehrwertige Feld *text\_all* dazu verwendet, die darin befindliche Liste mit den anderen gespeicherten Dokumentfeldern zu vergleichen. Alle Felder, deren Inhalt im text\_all-Feld enthalten ist, werden in der Detailansicht angezeigt.

Lediglich die Lokalisierung der Detailbeschriftungen muss noch manuell in der SageHR.Localization-Komponente sowie in der HR Portal-spezifischen Klasse *ClientLocalization* angelegt werden. Letztgenannte Klasse dient der Überführung von Texten aus SageHR.Localization in JavaScript-Objekte, auf die dann aus JavaScript heraus zugegriffen werden kann. Am Ende jeder Detailauflistung erscheinen die von Solr gefundenen Textstellen fett gedruckt und in dunkelroter Kursivschrift.

## Navigation vom Suchergebnis zur entsprechenden Ansicht

Doch wie funktioniert die Weiterleitung zu einer bestimmten View, nachdem ein Suchergebnis ausgewählt wurde? Bisher sind sämtliche View-Aufrufe in verschiedenen Ereignisbehandlungsfunktionen von Navigationsschaltflächen verankert oder mit anderen Ereignissen innerhalb von ViewModels verknüpft, bei denen oftmals eine Eigenschaftsänderung zu einem Ein- oder Ausblenden von Views führt, wie in Abschnitt 3.3 näher erläutert. Da es keine zentrale Stelle innerhalb des HR Portals gibt, die sich um die View-Anzeige kümmert und nicht die komplette Web-Anwendung dahingehend umgeändert werden kann in der Zeit, die für den Prototyp zur Verfügung steht, muss ein Kompromiss aus dem bisherigen und dem gewünschten Navigationsverhalten gefunden werden, sodass das HR Portal weiterhin uneingeschränkt genutzt werden kann.

Auf **Abbildung 5** in Abschnitt 3.1 ist mit der Detailansicht eines Arbeitnehmers zum Thema Resturlaub eine View dargestellt, auf die ein Suchergebnis verweisen muss – laut Anforderung. Die bisherige Anzeige der View wird durch eine Änderung der Eigenschaft *visible* des ViewModel *EmployeeDetailsViewModel* erreicht, da daran das Kendo-UI-Objekt *kendo.data.binder.slide* gebunden ist, welches bei jeder Änderung der Eigenschaft das Ein- bzw. Ausblenden ausführt. Weil diese Funktion von vielen verschiedenen HTML-Elementen im HR Portal genutzt wird, ist eine Änderung risikobehaftet.

Die View mit den Suchergebnissen hat jedoch keine Kenntnis von Navigationsschaltflächen oder ViewModel-Eigenschaften anderer Views, sodass es möglich sein muss, jede Ansicht innerhalb des HR Portals auch ohne Nutzerhilfe zu erreichen. Daher sieht das Navigationskonzept für den Prototyp zwei entscheidende Änderungen vor. Zum einen gibt es eine zentrale JavaScript-Komponente namens *SAGEHR.Navigator*, die innerhalb der gesamten HR Portal-Anwendung erreichbar ist. Diese sorgt dafür, dass zwischen der aktuell angezeigten und einer neu angeforderten View gewechselt werden kann. Da sie aber unabhängig von der Anwendung sein sollte, um als eigenständige Komponente für beliebige Web-Anwendungen verwendbar zu sein, muss sich die Anwendung selbst um das Ein- und Ausblenden von Ansichten kümmern.

Dies sei am Beispiel des Kalendermoduls innerhalb der Übersichtsansicht erläutert. Es gibt dafür eine JavaScript-Datei namens *poweruser.js*, die beim Start des Moduls sofort ausgeführt wird und u.a. das ViewModel für die gesamte Kalenderansicht initialisiert. Damit jede einzelne ein- und ausblendbare View unabhängig von den anderen angezeigt werden kann, benötigt deren HTML-Element *E*, welches durch ein div-Tag definiert ist, einen eineindeutigen Namen. Dieser Name wird im *id*-Attribut von E gespeichert, sodass E darüber von überall aus abrufbar ist. Im Folgenden sei dieser Name als *ViewId* bezeichnet. Auch jede Konstellation von zusammen angezeigten Views erhält einen eigenen Namen, welcher *ViewsComboId* genannt wird. In der *poweruser.js*-Datei erfolgt für jede *ViewsComboId* eine Zuordnung aller einzublendenden Views. Fehler! Verweisquelle konnte nicht gefunden werden. zeigt einen Auszug aus dem dazu definierten JavaScript-Objekt. Das Beispiel stellt eine Zuweisungsvorschrift für drei verschiedene Ansichtkonstellationen dar. Die erste trägt den Namen *calendar* und bezeichnet damit die Hauptansicht des Kalenderübersichtmoduls, welches vollständig in Anhang 11.7.1 zu sehen ist. Diese View ist in zwei Bereiche unterteilt, da sie in der HTML-Struktur an verschiedenen Stellen definiert sind. Der erste Bereich umfasst den Kopf der Kalenderansicht mit Schaltflächen zur zeitlichen Filterung der angezeigten Daten sowie zum Umschalten zwischen der Tabellen- und der Listenansicht.

Abbildung 23: Beispiel für Zuordnung von allen anzuzeigenden Views pro View-Schlüssel

Die zweite ViewsComboId *calendarEventList* zeigt auf die Ereignislistenansicht, die an der rechten Seite erscheint, sobald ein Tag im Kalender ausgewählt wurde. In Anhang 11.7.2 kann diese nachvollzogen werden. Wie in dem zugeordneten Array zu sehen ist, sind vier ViewIds dazu gespeichert, wobei die erste keine ViewId sondern der erste Eintrag des Zuordnungsobjekts ist. Das bedeutet, dass auch andere ViewsComboIds als Werte angegeben werden können, deren enthaltene Views nacheinander eingeblendet werden. Die calendarEventList besteht dazu noch aus der Hauptansicht für Kalenderereignisse, siehe Zeile 5, und einem Kopf- und einem Rumpfabschnitt, siehe Zeile 6 und 7.

Analog gilt dies auch für die Ereignisdetailansicht, die in Anhang 11.7.3 betrachtet werden kann und zusätzlich noch einen Fußabschnitt beinhaltet. Mit Hilfe dieser teilweise redundanten Definition von View-Kombinationen ist es nun möglich, eine spezifische Gesamtansicht direkt anzusprechen, wie es für die Weiterleitung ausgehend von der Suchergebnisverlinkung notwendig ist.

Doch wie kommt es genau zur Anzeige der Views? Jedes Betätigen der Verlinkung eines Suchergebnistitels bewirkt das Auslösen eines Klick-Ereignisses, welches die ereignisbehandelnde JavaScript-Funktion ausführt. Darin wird eine neue Hilfsinstanz erzeugt, die den gleichen Namen trägt, wie der Dokumenttyp des ausgewählten Suchergebnisses lautet. Dieses Objekt besitzt standardmäßig zwei Funktionen als Eigenschaften. Zum einen eine Initialisierungsfunktion *init()*, in der das von Solr zurückgegebene Dokument zum Ergebnis überreicht wird. Zum anderen weist es eine Navigationsfunktion namens *navigate()* auf, die sämtliche Schritte durchführt, um zu der anzuzeigenden View entweder das zugehörige ViewModel zu instanziieren oder andere notwendige Operationen auszuführen. Diese Navigationshilfsobjekte stellen eine Kompromisslösung dar, die dem unterschiedlichen Ladeverhalten der vielen Views geschuldet ist. Als Beispiel ist auf **Abbildung** **24** die Funktion zum Navigieren zur Resturlaubsübersicht dargestellt.

Abbildung 24: Funktion eines Hilfsobjekts zum Navigieren zur Resturlaubsübersichts-View

Die Zeilen 2 bis 15 erzeugen ein Datenobjekt aus den Daten des von Solr erhaltenen Dokuments. Dieses Datenobjekt wird daraufhin in Zeile 17 zum Instanziieren des zur View-Anzeige notwendigen ViewModel *EmployeeDetails.MainViewModel* verwendet. Daraufhin muss das neue ViewModel in Zeile 18 mit der View verbunden und in Zeile 19 mit dem aktuellen Jahr initialisiert werden.

Nachdem alle Bedingungen zum Anzeigen der View gegeben sind, ist das Einblenden dieser möglich. Jedoch kann das nicht vom Navigationshelfer oder vom ViewModel übernommen werden, da eine entscheidende Information dazu fehlt. Wenn eine View neu zu sehen ist, was passiert mit der eventuell zuvor gezeigten? Dazu wird eine zentrale Komponente verwendet, die alle aktiven Views kennt und vor dem Einblenden der neuen Views die alten einzeln ausblendet. Und wenn sich alle vorhandenen Views beim Start des HR Portal bei dieser Komponente anmelden, dann kann diese das Umschalten zwischen alten und neuen Views steuern. Die besagte JavaScript-Komponente sei als *SAGEHR.Navigator* oder kurz *Navigator* bezeichnet und ist von überall aus erreichbar, da sie beim Laden der JavaScript-Dateien zum Programmstart direkt instanziiert und in einer globalen Variable gespeichert wird.

Ein Software-Entwurfsmuster, das sich der Navigator dabei zunutze macht, ist das sogenannte *Publish/Subscribe*-Muster [19]. Während der Initialisierung registriert sich jede View am Navigator und übergibt dabei eine Funktion, die auch als Callback-Funktion bezeichnet und ausgeführt wird, sobald sie ein- oder ausgeschaltet werden soll. Zum Ausführen dieser Funktion wird vom Navigator ein Austauschobjekt mitgeliefert, welches zum einen die ViewsComboId, zum anderen den boolschen Wert, ob die zugehörigen Views eingeblendet werden sollen oder nicht sowie ein zusätzliches Datenträgerobjekt enthält, das optional beim Aufruf der Umschaltfunktion des Navigators mitgegeben wurde. Mit Hilfe dieser Angaben kümmert sich die betroffene View selbst um ihre Anzeige und ermöglicht dabei auch eine höhere Flexibilität, sollte der Ein- bzw. Ausblendeeffekt individuell gestaltbar sein.

Das ViewModel kann letztendlich beim Initialisieren die Umschaltfunktion *switchView* des Navigators zusammen mit der ViewComboId der zuerst anzuzeigenden View aufrufen. Im zuvor beschriebenen Beispiel ist das die ViewsComboId *calendarEventList*. Der gesamte Quelltext des Navigators sowie der Registrierung und Ausführung der dabei übergebenen Callback-Funktion zur Anzeige der Views in der Kalenderübersicht ist im Anhang 11.8 nachzulesen.

Um von einem Suchergebnis zur Bewerberdetailsansicht direkt navigieren zu können, ist ausnahmsweise keine ViewModel-Instanziierung notwendig. Das gesamte Bewerber-Modul bietet eine integrierte Volltextsuche auf Basis der relationalen PW-Datenbank. Dadurch reicht es, wenn diese Suchfunktion vom Bewerber-Navigationshilfsobjekt aufgegriffen wird und über die Angabe des Bewerbernamens aus den Dokumentdaten der View-Wechsel erfolgt. Auch dabei wird in einem ViewModel der Navigator aufgerufen, sodass das Umschalten problemlos funktioniert.

# Diskussion der Thesen

Mit Hilfe der zuvor erläuterten Implementierung des Prototyps ist es möglich, innerhalb der Kalenderübersicht verschiedene Ansichten über die Volltextsuche zu erreichen ohne eine Menüschaltfläche betätigen zu müssen. Außerdem konnte damit erreicht werden, dass Bewerbungsdokumente gezielt nach Stichwörtern durchsuchbar sind, sofern das eingegebene Wort richtig geschrieben wurde und im Volltext existiert.

Fünf verschiedene Thesen gilt es auf ihre Richtigkeit zu überprüfen bzw. deren Grenzen aufzuzeigen. In Abschnitt 1.5 ist deren Formulierung nachzulesen. Während der Implementierung des Prototyps sind Schwierigkeiten und Probleme aufgekommen, die bei der Diskussion der Thesen im Folgenden mit berücksichtigt werden.

## Legitimierung der Volltextsuche in volltextarmen Anwendungen

Zunächst stellte sich die Frage, ob die Integration der Volltextsuche in eine Anwendung, wie das HR Portal, sinnvoll ist, obwohl diese vorwiegend rein strukturierte Daten zeigt. Wenn die wenigen Volltextdaten, wie z.B. Bewerbungsdokumente oder Mitarbeitergesprächsnotizen, außer Acht gelassen werden, so bleiben hauptsächlich Daten übrig, die wenig Text enthalten. Als Beispiel für Textdaten gelten u.a. Namen, Adressen, Titel, Geburtsort und -land oder auch Projektbezeichnungen. Da diese Daten bereits strukturiert vorliegen ist eine weitere Durchsuchung mit Hilfe einer Volltextsuche nicht notwendig.

Andere Quellen, die nach Textdaten durchsucht werden können, stellen die zahlreichen Bezeichnungen für anzuzeigende Daten dar, die im Fachjargon auch *Labels* genannt werden. Hinzu kommen pro Ansicht beschreibende Überschriften, die das dort angezeigte Thema verdeutlichen, damit ein Benutzer sofort weiß, was er sich gerade anschaut. All diese nicht in der PW-Datenbank vorkommenden Textdaten können von einer Volltextsuche dennoch durchsucht werden und dem Benutzer die Bedienung der Anwendung erleichtern. Dann dient die Volltextsuche weniger dem Finden von Informationen innerhalb eines unstrukturierten Textes, sondern vielmehr der gezielten Hinführung zu gewünschten Daten, die auf Benutzeroberflächen zu sehen sind.

Wenn aber kaum unstrukturierte Daten für die Suche verwendet werden, stellt sich die Frage, ob es sinnvoll ist, dennoch eine Volltextsuche mit Solr zu implementieren. Gibt es keine einfacher umzusetzenden Alternativen? Wenn nur Daten indexiert werden sollen, deren Struktur bekannt ist, könnte stattdessen diese in einer eigenen relationalen Datenbanktabelle zum Durchsuchen abgebildet und durchsucht werden. Stellenweise existiert bereits im HR Portal eine solche Möglichkeit der Datenbank gestützten Volltextsuche, wie z.B. im Bewerber-Modul.

Jedoch hat ein solcher Datenbankfilter gegenüber Solr einen entscheidenden Nachteil. Er ist stark kontextabhängig. Das bedeutet, dass pro Themengebiet eine eigene Tabelle definiert werden muss, weil die Definition der Spaltendatentypen für alle Datensätze identisch ist. Jede Tabelle wird über einen invertierten Index, ähnlich wie dem Solr-Index, effizienter durchsuchbar. Dadurch entstehen genauso viele verschiedene Indizes wie es Suchthemen gibt, sodass pro Thema eine eigene Abfrage an den Server gestellt werden muss.

Der Index von Solr ist da deutlich flexibler, da mit der NoSQL-typischen Datenstruktur jedes Dokument individuell gestaltet werden kann und dennoch alle innerhalb eines einzigen Indexes existieren. Somit ist nur eine Abfrage notwendig, egal wie viele Themen es gibt. Höchstwahrscheinlich ergibt dieser Umstand einen Leistungsvorteil des Solr-Servers gegenüber der Datenbankvolltextlösung. Wie groß der Leistungsunterschied tatsächlich ist, müsste erst noch geprüft werden. Wenn jedoch die Suchanfragen über ein Netzwerk zum Such-Server gelangen, ergibt sich logischerweise aus mehreren themenabhängigen Anfragen an einen Datenbank-Server eine zeitliche Verzögerung gegenüber der einen Anfrage an den Server von Solr.

Nachdem die Legitimation der Solr-Volltextsuche gegenüber der Verwendung eines relationalen DBMS[[36]](#footnote-36) bestätigt ist, wurde aber noch nicht geklärt, ob Solr auch als Hilfsmittel zur Navigation zu bestimmten Anwendungsansichten geeignet ist.

## Eignung der Volltextsuche für Navigation im HR Portal

Abschnitt 3.3 hat gezeigt, wie das Aufrufen von Views im HR Portal grundsätzlich funktioniert. Es stellte sich dabei heraus, dass die Navigation vollständig abhängig von Benutzerschaltflächen ist. In diesen sind zum einen URLs gespeichert, die für den Controller-Aufruf des HR Portal-Server notwendig ist, der daraufhin die angefragte View an den Web-Browser ausliefert. Zum anderen verknüpft das Kendo UI–Framework ViewModel-Eigenschaften mit dem Ein- und Ausblenden von Views, sodass eine Änderung dieser Eigenschaften mit Hilfe eines Schalters vorgenommen und damit letztendlich navigiert werden kann.

Das reine Finden von Dokumenten anhand der Suchbegriffe des Benutzers ist davon völlig unabhängig. Und so erhält das HR Portal, egal in welchem Zustand es sich befindet, Suchergebnisse mit aussagekräftigen Informationen, die zu einer bestimmten Ansicht passen. Was die Anwendung damit macht, um die entsprechende Anzeige der View zu gewährleisten, ist allein ihr Problem und wird in den Abschnitten 5.3 und 5.4 diskutiert. Dennoch kann besonders eine Solr-Volltextsuche sich als Ausgangspunkt zum Navigieren eignen. Alleine die Funktionen zur Analyse von Nutzereingaben sind von Vorteil. Falls die Eingabe beispielsweise falsch geschrieben wurde, können dennoch Ergebnisse vorgeschlagen werden, die ähnliche Begriffe beinhalten. Auch eine mögliche Autovervollständigung während des Suchbegriffeintrags kann den Benutzer dabei unterstützen, die gewünschte Ansicht zu finden. Im Rahmen des Prototyps konnten jedoch nicht alle umgesetzt werden, da dies einen deutlich größeren Konfigurationsaufwand bedeutete und noch mehr Zeit benötigt. An dieser Stelle sei daher nur auf den umfangreichen Abschnitt *Searching* der offiziellen Solr-Dokumentation [9] sowie auf Kapitel 10 der auf Solr spezialisierten Literatur verwiesen [7].

Obwohl diese Funktionen nicht im Prototyp angewendet werden können, ist es dennoch dank der Volltextsuche möglich, im HR Portal über einen einzigen Navigationsschritt des Benutzers Ansichten aufzurufen. Dies erfolgt über eine Verlinkung, die dessen Ziel deutlich informationsreicher beschreibt, als es eine Schaltfläche mit einem Symbol oder Wort vermag. Abschnitt 4.4 veranschaulicht dies sehr gut anhand der Abbildung der Suchergebnisse.

Auch wenn sich die Volltextsuche prinzipiell gut für die Navigation im HR Portal eignet, musste dennoch deren bisherige Abhängigkeit vom Betätigen der Schaltflächen durch den Benutzer beseitigt werden, wie Abschnitt 4.5 es ausführlich beschreibt. Warum dies zentral gesteuert erfolgen sollte, erörtert der nächste Unterpunkt.

## Zentrale Steuerung der Navigation

Um einordnen zu können, welche Vor- und Nachteile eine zentrale Komponente für die Weiterleitung zu Views innerhalb des HR Portals mit sich bringt, ist zunächst zu klären, wie die Alternative dazu aussieht. Prinzipiell kann in der bisher entwickelten Web-Anwendung der PW von einer dezentralen Navigation gesprochen werden. Es gibt keinen einheitlichen Mechanismus, der zum Umschalten zwischen verschiedenen Views verwendet wird. Dies ist auch der Software-Architektur geschuldet, die in Kapitel 3.1 beschrieben wurde.

Die Gesamtanwendung HR Portal basiert auf dem ASP .NET MVC-Framework, das ein permanentes Anfragestellen des Client an den Server vorgibt, um andere Views erhalten und anzeigen zu können. Hinzu kommt, dass viele der HR Portal–Module, wie z.B. die Kalenderübersicht, für sich betrachtet eine SPA auf Basis des MVVM-Musters darstellen – vorgegeben vom Kendo UI–Framework. Dort sind sämtliche Views bereits vom Browser abgerufen worden, sodass zwischen ihnen durch reines Ein- und Ausblenden sowie Nachladen von Daten mit Hilfe von AJAX gewechselt werden kann.

Wenn nun Ergebnisse der Volltextsuche über die beschriebene dezentrale Navigation zu bestimmten Ansicht führen sollen, dann ergibt sich das Problem, dass die View, die sich um die Darstellung der Suchergebnisse kümmert, keine Informationen zu den anderen Views beinhaltet. Demzufolge kann dort keines der zugehörigen ViewModels instanziiert und initialisiert werden. Die in Abschnitt 4.5 beschriebenen Hilfsobjekte sind mit Hilfe des eindeutig zuordenbaren Dokumenttyps dazu in der Lage. Und weil jedes ViewModel den Namen der miteinander verknüpften View speichern darf – auch im Sinne von MVVM – kann es beim Initialisieren die View anzeigen.

Was aber ViewModels sowie Hilfsobjekte nicht leisten, ist die Notwendigkeit, die bisher eingeblendeten Views auszublenden. Schließlich kann sich der Benutzer innerhalb eines bestimmten Moduls befinden und dort über die Volltextsuche eine andere View anfragen. Genau für das Umschalten zwischen Views ist eine zentrale Navigation unablässig. Dazu speichert sie den Namen der aktuell angezeigten Ansicht.

Ein weiterer Vorteil des zentralen Navigationskonzepts stellt die Möglichkeit dar, Daten der Views in der sogenannten Browser-Historie abzulegen, wodurch die im Web-Browser zur Verfügung stehende Navigation über die vor- bzw.- rückwärts gerichteten Pfeilschaltflächen genutzt werden kann. Auch darum würde sich die zentrale Komponente kümmern und bietet für die Umsetzung des gesamten HR Portals als SPA die Chance, ohne Aufruf von URLs durch den Browser, jeder View eine eigene Adresse zuzuordnen und in der Adresszeile des Web-Browser anzuzeigen, indem die sogenannte *History API* des aktuellen HTML-Standards *HTML5* genutzt wird [20]. Denn wie der nächste Abschnitt erläutert, ist ein Umbau des HR Portals zu einer reinen SPA für die vollständige Implementierung einer modulübergreifenden Volltextsuche notwendig.

## HR Portal als Single-Page-Anwendung

Die bisherige Software-Architektur des HR Portals führt dazu, dass mit dem Betätigen von Schaltflächen innerhalb des Hauptmenüs die komplette Web-Anwendung mit allen für das anzuzeigende Modul benötigten Dateien neu vom Server abgerufen und daraufhin vom Web-Browser neu geladen wird. Die Implementierung der Volltextsuche mit Hilfe von Solr beschränkte sich einmal auf das Modul Kalenderübersicht für Benutzer mit entsprechenden Rechten zur Anzeige der Mitarbeiterdaten und zum anderen auf das Bewerbermodul. Solange der Nutzer nicht das Modul aktiviert hat, welches durchsucht werden soll, sind sämtliche Informationen zu dessen HTML-Struktur, u.a. mit den für die Navigation notwendigen View-Bezeichnungen, nicht verfügbar.

Eine SPA hat dieses Problem nicht. Sämtliche HTML-Dateien müssen bereits an den Web-Browser ausgeliefert worden sein, damit ein Nachladen dieser nicht mehr notwendig ist. Außerdem spart es Zeit, wenn beim Navigieren zu einer View nicht das gesamte zugehörige Modul neu abgerufen werden muss. Insofern bietet die Umwandlung des HR Portals in eine reine SPA die Chance, über die Volltextsuche verlinkte Views anhand von Suchergebnissen schneller aufzurufen, als es bisher der Fall ist [21]. Zumal nur die Daten vom Server per AJAX-Anfrage im Hintergrund nachgeladen werden müssen, die für die Darstellung der gesuchten View notwendig sind.

Der besagte Umbau bedeutet aber eine gravierende Änderung der bisher bestehenden Software-Architektur bezogen auf das verwendete ASP.NET MVC–Framework. Nur wenn die Views nicht mehr einzeln über zugehörige Controller, sondern als Gesamtpaket abrufbar sind, kann das HR Portal als SPA eingesetzt werden. Dazu muss nicht zwingend auf ASP .NET MVC verzichtet werden, da schließlich auch ein einziger Controller zum Abrufen der Views zur Verfügung gestellt werden kann. Jedoch ist dann die Sinnhaftigkeit eines Einsatzes dieses Framework zu hinterfragen, da dessen Vorteile bei der Trennung von Model, View und Controller nicht voll ausgeschöpft werden können [22]. Schließlich ergibt sich dann keine Unterteilung der Web-Anwendung in Views, sondern bei diesem Ansatz besteht die gesamte Anwendung aus einer View.

## Verwendung des DIH zum automatisierten Indexieren

In Abschnitt 2.2.3 sind zwei Möglichkeiten angegeben, in Solr Daten zu indexieren. Entweder müssen selbst definierte Dokumente an den Solr-Server über dessen REST-Schnittstelle an einen definierten IndexHandler gesendet werden, der die Indexierung dann vornimmt. Oder die Solr-Komponente namens Data Import Handler, bzw. DIH, kümmert sich selbstständig um die Erstellung und Indexierung der Dokumente, nachdem die Daten automatisch von einer definierten Datenquelle abgerufen wurden. Der DIH verspricht somit weniger Arbeit für den Solr-Administrator bei der Erstellung des Indexes, weswegen die Entscheidung auf ihn bei der Umsetzung des Prototyps fiel. Während der Verwendung des DIH traten jedoch einige Probleme auf, deren Lösung teilweise mit größerem Aufwand verbunden war.

Ein Hauptproblem bestand darin, dass keine der von Solr unterstützten Datenquellen mit der WebAPI der PW vereinbar waren, was Abschnitt 3.2 mit deren fehlender Möglichkeit begründet, sich an dieser zu authentifizieren. Da der Quelltext von Solr aufgrund der Apache-Lizenz offen verfügbar ist, können benutzerdefinierte Datenquellen angelegt werden, wie es Kapitel 4.2.2 ausführlich beschrieben hat. Obwohl es aufgrund der modularen Software-Architektur möglich ist, eine eigene DataSource-Klasse zu definieren und in der Konfigurationsdatei des DIH zu verwenden, erwies sich das Hinzufügen von eigenen Parametern, die in dieser Datei zusätzlich übergeben werden sollen, als äußerst umständlich. Es gibt dafür keine Schnittstelle und das Auslesen erfolgt sehr intransparent innerhalb von Kernklassen des DIH. Weil der Auslesemechanismus nicht ohne tiefergehende Untersuchung des Solr-Quelltextes anpassbar war, wurde sich für die Notlösung entschieden, bereits vorhandene Parameter des Datentyps *String* mit zusätzlichen Daten, z.B. zur Autorisierung der WebAPI, zu versehen und diese durch eindeutige Platzhalter extrahierbar werden zu lassen.

Aufgrund der besseren Anpassbarkeit und höheren Kontrolle sei die Alternative empfohlen, eine eigene Indexierungskomponente in einer Programmiersprache der Wahl zu entwickeln, die im Falle der PW das Auslesen der WebAPI und Erstellen von Dokumenten selbstständig vornimmt. Sobald die Dokumente an einen entsprechenden IndexHandler gesendet wurden, steht der anschließende Indexierungsprozess dem des DIH in nichts nach. Bei beiden erfolgt ein Abbruch der Indexierung, sobald ein Fehler auftritt. Erst nach einer abschließenden Bestätigung, im Fachjargon von Solr als *Commit* bezeichnet, werden alle validierten Dokumente dem Index hinzugefügt. Zu beachten ist, dass ein Commit bei der Verwendung des IndexHandler vom Administrator selbst initiiert werden muss, sonst findet keine Indexierung statt.

# Zusammenfassung, Fazit und Ausblick

## Zusammenfassung

Eine Volltextsuche gilt es für die Sage HR Suite bzw. genauer für deren Web-Benutzeroberfläche HR Portal zu konzipieren und in einem Prototyp exemplarisch deren Machbarkeit nachzuweisen. Mit Apache Solr wird eine Suchtechnologie verwendet, die bereits in vielen namhaften Anwendungen, wie z.B. *eBay*, *Netflix* oder *Instagram*, eingesetzt wird [23]. Sie ist dazu gedacht, reine Volltextdaten besser durchsuchen zu können. Ähnlich wie ein Buch am Ende einen Index bereitstellt, mit dessen Hilfe Schlagwörter bestimmten Seiten zugeordnet werden können, liefert die Volltextsuche eine Indexdatenbank, mit deren Hilfe zu Begriffen aus der Textquelle die zugehörigen Textstellen in kürzester Zeit ermittelbar sind.

Da der Suchindex in einer NoSQL-Datenbank angelegt ist, basiert die Volltextsuche auf Dokumenten, die völlig verschieden entsprechend der Suchthemen aufgebaut sein dürfen. Diese Flexibilität macht sich die auf Apache Lucene basierende Suchtechnologie im Gegensatz zur Volltextsuche in relationalen Datenbankmanagementsystemen, wie dem Microsoft SQL Server, insofern zunutze, als dass beliebig viele verschiedene Themenbereiche durch nur eine Abfrage innerhalb eines Indexes durchsucht werden können. Wohingegen eine relationale Datenbank pro Tabelle einen eigenen Index anlegt, wodurch zu *n* Suchbereichen *n* Indizes entstehen und entsprechend *n* Suchanfragen gestellt werden müssten.

Das HR Portal bietet jedoch kaum Volltextdaten, dafür umso mehr strukturierte Daten, sodass sich die Frage stellt, ob eine Volltextsuche dann noch sinnvoll ist. Eine Chance, die sich anbietet, ist die verbesserte Bedienbarkeit der Anwendung, indem durch die Benutzereingabe von themenorientierten Schlagwörtern zugehörige Ansichten des HR Portals sofort auffindbar sind. Das wäre eine Alternative zur klassischen Navigation über das Hauptmenü. Zudem könnten die wenigen existierenden Volltextdaten, z.B. in Form von digital gespeicherten Bewerbungsunterlagen, leichter gefunden werden, sofern nach spezifischen Wörtern darin gesucht wird.

Nach der recht schnellen Installation von Solr besteht eine Hauptaufgabe darin, die Anwendungsdaten zu indexieren, die aus der relationalen PW-Datenbank stammen. Dazu muss eine Entscheidung getroffen werden, ob direkt auf die Datenbank zugegriffen werden soll oder indirekt über die bereits für die PW vorhandene WebAPI, die eine gut dokumentierte Schnittstelle zu den Daten darstellt und durch REST-artige HTTP-Anfragen den gezielten Zugriff auf diese ermöglicht. Die WebAPI nicht zu nutzen, würde bedeuten, dass sämtliche dahinter in der Geschäftslogik befindlichen SQL-Abfragen zum Auslesen der PW-Daten an anderer Stelle neu definiert und gepflegt werden müssten. Außerdem liefert sie die Daten in wohl strukturierter Form im JSON- oder XML-Format, sodass diese programmatisch sofort weiter verarbeitet werden können. Zusätzlich kümmert sich die WebAPI um die Autorisierung von Benutzern und um die Bestimmung von Benutzerrechten für verschiedene Anwendungen der PW. Dadurch ist keine weitere Authentifizierung am Solr-Server notwendig, sofern der Zugriff auf diesen nur lokal über die WebAPi erfolgt und nicht aus dem Netzwerk möglich ist. Sonst wäre kein Zugriffsschutz für indexierte, sensible Daten gewährleistet.

Der Weg über die WebAPI-Schnittstelle erschwert jedoch die Konfiguration des verwendeten Data Import Handler zur automatisierten Indexierung durch Solr. Denn eine dafür benötigte DataSource-Java-Klasse, die die Authentifizierung an der WebAPI ermöglicht, muss erst neu angelegt werden. Da auch Daten aus Dateiformaten, wie u.a. PDF, Word-Dokumenten, indexiert werden müssen, ist eine weitere neu angelegte DataSource-Klasse zur Verwendung des Open-Source-Projekts Apache Tika notwendig. Innerhalb der DIH-Konfigurationsdatei sind Dokumente zu definieren, die alle benötigten Daten in Feldern des Solr-Schemas speichern und als Suchergebnisse vom Solr-Server an das HR Portal zurückgeben.

Die erhaltenen Ergebnisdokumente werden so auf der Ergebnisansicht dargestellt, dass sofort ersichtlich wird, um welches Thema es sich dabei handelt und inwiefern sich gleiche Themen durch spezifische Datenangaben unterscheiden, sodass für den Benutzer auch ohne Erfahrungen mit der Anwendung die Entscheidung erleichtert wird, welches Suchergebnis am besten passen könnte. Pro View des HR Portals ist ein Dokumenttyp definiert, der jedem Dokument zugeordnet ist, damit innerhalb des HR Portals entschieden werden kann, zu welcher View ein Suchergebnis per Weiterleitung führt.

Um das Umschalten zwischen bisher angezeigten und anzuzeigenden Views kümmert sich eine eigens entwickelte Navigator-Komponente, die von überall innerhalb der Anwendung aufrufbar ist. Für den eingeschränkten HR Portal-Bereich mit der Kalenderübersicht und der Bewerberdetailansicht ist zudem die bisherige Navigation der Anwendung an das neue Navigationskonzept angepasst worden. Um dafür jedoch die Volltextsuche nutzen zu können, ist es notwendig, das jeweilige Modul über die Betätigung der entsprechenden Navigationsschaltfläche aufzurufen. Grund dafür ist die vom ASP.NET MVC-Framework vorgegebene Software-Architektur, die die Navigation insofern einschränkt, als dass die gesamte HTML-Struktur aller Views beim Start der Web-Anwendung nicht zur Verfügung steht. Innerhalb des Kalenderübersichtmoduls erfolgt jedoch die sofortige Auslieferung aller Views, deren Daten erst dann per AJAX-Anfrage an die WebAPI nachgeladen werden, sobald eine View eingeblendet werden soll, was dem Grundprinzip einer Single-Page-Application entspricht.

Ein wichtiger Aspekt, der bei der Umsetzung der Volltextsuche im HR Portal zu beachten ist, stellt die Berücksichtigung von Zugriffsrechten dar, da ansonsten jeder Benutzer Suchergebnisse und damit sensible Daten zu Arbeitnehmern erhalten kann, zu denen er aufgrund seiner Position im Unternehmen keinen Zugang haben darf. Auch wenn der Prototyp dies aus Zeitgründen nicht berücksichtigt, existiert innerhalb dieser Arbeit ein Konzept, das beschreibt, wie die WebAPI die Suchergebnisse entsprechend der Nutzerrechte vorfiltern kann.

Ebenfalls aus Zeitgründen nur erläutert und nicht umgesetzt ist die Möglichkeit, den Indexierungsprozess zu optimieren, indem nur seit dem letzten Indexierungszeitpunkt geänderte Daten neu indexiert werden. Dies funktioniert aber erst dann, wenn die PW-Datenbank zu jedem Datensatz eine Spalte hinzufügt, die den Zeitpunkt der letzten Änderung speichert. Dadurch kann womöglich die Dauer der Aktualisierung der Indexdatenbank stark verkürzt werden. Weil eine Optimierung der Indexierung nicht Teil der Arbeit ist, jedoch gemessene Leistungsdaten des Solr-Server für den Prototyp von Interesse sein können, sind im Anhang 11.9 ein paar Daten dazu angegeben.

## Fazit

Als Fazit lässt sich festhalten, dass prinzipiell eine Volltextsuche für die Sage HR Suite möglich ist. Die PW bietet zwar keine typische Datenquelle für die Durchsuchung von Volltexten, jedoch lassen sich zu jeder Ansicht Themen zuordnen, deren zugehörige Schlagworte indexiert und durchsucht werden können. Um dies zu untermauern, konnte anhand eines Prototyps gezeigt werden, dass eine Volltextsuche innerhalb des HR Portals umsetzbar ist. Wenn aber diese modulübergreifend funktionieren soll, gilt es noch zusätzlichen Aufwand zu betreiben, die gesamte Web-Anwendung zu einer reinen Single-Page-Anwendung umzugestalten.

Des Weiteren erwies sich die Arbeit mit Apache Solr als recht mühselig, obwohl eine umfangreiche Dokumentation vorhanden ist. Vor allem die vielen Konfigurationsmöglichkeiten erschweren den Einstieg, was jedoch nicht als negativ zu werten ist, da sie auch enorm viel Potential zur Optimierung von Suchanwendungen bieten. Der Fokus der Arbeit lag aber nicht auf der Optimierung der Suche, sondern auf der Analyse der Verwendungsmöglichkeit für die PW bzw. genauer für das HR Portal der PW. Soll die Volltextsuche auch für deren Desktop-Anwendungen genutzt werden, ist eine zusätzliche Analyse der Navigationsmechanismen der jeweiligen Applikationen notwendig. Die Erkenntnisse zur reinen Verwendung des Solr-Server gelten anwendungsübergreifend, sodass lediglich innerhalb jedes Moduls die Weiterleitung von Suchergebnissen zu den entsprechenden Views individuell umgesetzt werden muss.

## Ausblick

Während der Umsetzung des Prototyps sind Ideen zur Optimierung aufgekommen, die sich für eine sofortige Implementierung als zu aufwändig herausstellten, aber für eine spätere vollständige Implementierung durchaus wertvoll sein können. Die folgenden Abschnitte stellen nur eine Auswahl an möglichen Erweiterungen bzw. Optimierungen des Prototyps dar und geben teilweise einen Vorgeschmack auf das, was Solr anbietet, um die Volltextsuche so benutzerfreundlich wie möglich werden zu lassen.

### Automatisierte Schlagwortgewinnung für Indexierung

In dem bisherigen Prototypkonzept ist vorgesehen, dass jede View anhand von Schlagwörtern u.a. auffindbar ist, die vom Software-Entwickler manuell zu definieren und anzulegen sind. Um die Volltextsuchfunktion weniger pflegen zu müssen und dennoch für möglichst alle Views verfügbar werden zu lassen, bietet es sich an, die Schlagwörter automatisch zu generieren und zu speichern. Da die Bezeichnungen auf jeder HR-Portal-View je Sprache in einer eigenen Ressourcendatei lokalisiert auf Deutsch und Englisch vorliegen, wäre es ein Ansatz, eine Möglichkeit zu schaffen, die dortigen Schlüssel-Wert-Paare mit dem passenden Dokumenttyp zu verknüpfen. Dazu könnte beispielsweise eine zusätzliche Ressourcendatei angelegt werden, die zu den vorhandenen Schlüsseln den eindeutigen Dokumenttyp als Wert speichert.

Sollen weitere PW-Anwendungen mit anderen Benutzeroberflächen als der des HR Portals mit der Volltextsuche ausgestattet werden, ist es notwendig, die dafür neu angelegten Lokalisierungsdateien um eine entsprechende Zuordnung der dort festgelegten Begriffe zu Dokumenttypen zu erweitern. Nur durch die Kategorisierung der WebAPI-Daten über Dokumenttypen ist es möglich, Solr weitestgehend anwendungsunspezifisch für die Suche einzurichten und nicht für jede Anwendung die Volltextsuche neu zu erfinden.

### Fehlerhafte Sucheingaben und Autovervollständigung

Wenn ein Benutzer in der aktuellen Prototyp-Volltextsuche Suchwörter eingibt, die nicht in der gleichen Schreibweise indexiert wurden, dann erhält er keine Ergebnisse. Solr bietet die Möglichkeit, nach entsprechender Erweiterung der Konfiguration, auch bei Falscheingaben Vorschläge zu liefern, die sich nur gering von der Sucheingabe unterscheiden. Wie stark sie sich unterscheiden dürfen, ist über Parameter einstellbar und benötigt eine Justierung, die im Rahmen dieser Arbeit nicht im Zeitrahmen lag. Weitere Informationen finden sich dazu im Abschnitt *Spell Checking* aus der Online-Dokumentation von Solr [9]. Das gleiche gilt für die von Solr unterstützte Funktion, während der Nutzereingabe Vorschläge für bekannte Suchbegriffe zu liefern, die den bisher eingegebenen Wortanteil enthalten. Dazu gibt es in der besagten Dokumentation einen Abschnitt namens *Suggester*.

### Übersichtlichere Darstellung der Suchergebnisse

Nach einer Suchanfrage durch den Prototyp erscheinen bisher nur die ersten zehn gefundenen Suchergebnisse auf der entsprechenden View. Sollten jedoch deutlich mehr Ergebnisse von Interesse sein, gibt es die Möglichkeit, in fest definierten Blöcken immer mehr Suchergebnisse abzufragen und auf je einer eigenen Seite anzuzeigen. Diese seitenweise Darstellung muss nicht auf zehn Einträge beschränkt sein, jedoch sorgt die geringe Zahl für mehr Übersicht und es muss nicht vertikal *gescrollt* werden, um weiter unten liegende Ergebnisse sehen zu können. Dazu muss nichts weiter getan werden, als zusätzliche Schaltflächen oder Verlinkungen in die Ergebnis-View einzubauen, die das Nachladen und Anzeigen der nächsten Ergebnisse bewirken. Über solche Seiten-Verlinkungen kann dann zwischen verschiedenen hin- und her navigiert werden. Mehr dazu ist in der Solr-Dokumentation im Abschnitt *Pagination of Results* zu finden [9].

Bisher werden sämtliche Details zu Ergebnissen als Liste aufgeführt, deren Länge davon abhängig ist, wie viele Felder des Dokuments zur Durchsuchung nach Suchbegriffen vom Solr-Server verwendet werden. Um eine einheitliche Darstellung der Suchergebnisse zu ermöglichen, die genau auf die Ansicht passt, bietet es sich an, die Details zu je einem Ergebnis in einem Block gleicher Höhe anzuordnen und über eine Erweiterungsschaltfläche, den Block zu verlängern, sofern alle Informationen zu sehen sein sollen. Dadurch sind ausführlichere Informationen zu den Suchergebnissen bereitstellbar ohne den Überblick über die Ergebnisliste zu verlieren.

Sollten sehr viele Ergebnisse vom Solr-Server gefunden worden sein, bietet Solr eine Funktion an, verschiedene Kategorien zur weiteren Filterung der Suchtreffer anzeigen zu lassen. Dazu könnte an einer Seite der Ergebnis-View ein Menü angezeigt werden, das zum einen alle gefundenen Suchkategorien auflistet und zum anderen die Anzahl der zugehörigen Suchergebnisse mit angibt, sodass der Benutzer selbst entscheiden kann, wie er die Suche weiter einschränken möchte. Nähere Informationen sind dazu in der besagten Solr-Dokumentation unter dem Punkt *Faceting* zu finden.

# Abkürzungsverzeichnis

**API** *Application Programming Interface*

**CSS** *Cascading Style Sheets*

**DIH** *Data Import Handler*

**DLL** *Dynamic Link Library*

**DTO** *Data Transfer Object*

**GUID** *Globally Unique Identifier*

**HTTP** *Hypertext Transfer Protocol*

**HTML** *Hypertext Markup Language*

**IIS** *Internet Information Services*

**IR** *Information Retrieval*

**JDBC** *Java Database Connectivity*

**JSON** *JavaScript Object Notation*

**MVC** *Model View Controller*

**MVVM** *Model View ViewModel*

**NoSQL** *Not only SQL*

**PDF** *Portable Document Format*

**PW** *Sage HR Suite*

**REST** *Representational State Transfer*

**SQL** *Standard Query Language*

**URL** *Uniform Resource Locator*

**XML** *Extensible Markup Language*

# Abbildungsverzeichnis

[Abbildung 1: Indexierungsprozess von Lucene [4 S. 34] 12](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755558)

[Abbildung 2: Indexierungsvorgang mit Zusammenhang zwischen Dokument, Feldern, Analysierer und Indexdatenbank 14](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755559)

[Abbildung 3: Beziehung zwischen den Hauptklassen von Lucene zum Durchsuchen des Indexes 15](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755560)

[Abbildung 4: Schematische Darstellung des Architekturmusters MVC (durchgezogene Linie: direkte Beziehung; gestrichelte Linie: Benachrichtigung über Ereignis [roter Blitz]) 23](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755561)

[Abbildung 5: Bildschirmausschnitt zum Aufruf der Resturlaubansicht in Kalenderübersicht des HR Portals. Die gesamte Ansicht zum Menüpunkt Übersicht ist eine SPA. 24](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755562)

[Abbildung 6: Schematische Darstellung des Architekturmusters MVVM (durchgezogene Linie: direkte Beziehung; gestrichelte Linie: Benachrichtigung über Ereignis [roter Blitz]) 25](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755563)

[Abbildung 7: Menü zu den Hauptmodulen des HR Portals 27](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755564)

[Abbildung 8: Code-Beispiel zum Umschalten zwischen zwei Views mit Hilfe des Kendo UI Frameworks 28](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755565)

[Abbildung 9: Ausschnitt aus der Detailansicht innerhalb des HR Portal-Moduls Zeiterfassung 32](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755566)

[Abbildung 10: C#-Quelltext zur Generierung eines möglichst einzigartigen Dokumentenschlüssels anhand einer gegebenen Zeichenkette 33](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755567)

[Abbildung 11: Startseite der Weboberfläche zum Solr-Server 37](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755568)

[Abbildung 12:UML-Verteilungsdiagramm zur Client-Server-Architektur des Produktivsystems HR Portal mit Volltextsuche 37](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755569)

[Abbildung 13: Eintrag des DIH als RequestHandler in die Hauptkonfigurationsdatei von Solr 41](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755570)

[Abbildung 14: Definition der Datenquellen innerhalb der DIH-Konfigurationsdatei 42](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755571)

[Abbildung 15: Definition eines Dokuments in DIH-Konfigurationsdatei für Suche der Resturlaubsansicht 43](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755572)

[Abbildung 16: DTO-Klasse für Datenobjekt zur Indexierung 45](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755573)

[Abbildung 17: WebAPI-Controller-Methode zum Abfragen von Daten zur Indexierung 45](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755574)

[Abbildung 18: Methode aus der *FullTextSearchService*-Klasse zur Beschaffung der angeforderten Daten für die Indexierung 46](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755575)

[Abbildung 19: Benutzeroberfläche der prototypsartigen Sucheingabe im HR Portal 47](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755576)

[Abbildung 20: JavaScript-Funktion zum Behandeln von Mausklickereignissen nach Betätigung der Suchschaltfläche 47](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755577)

[Abbildung 21: URL für Suchanfrage an den Solr-Server 48](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755578)

[Abbildung 22: Ausschnitt aus Ansicht der Ergebnisse der Volltextsuche 49](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755579)

[Abbildung 23: Beispiel für Zuordnung von allen anzuzeigenden Views pro View-Schlüssel 51](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755580)

[Abbildung 24: Funktion eines Hilfsobjekts zum Navigieren zur Resturlaubsübersichts-View 52](file:///H:\BA%20Leipzig\Bachelorarbeit\Bachelorarbeit.docx#_Toc491755581)

# Tabellenverzeichnis

[Tabelle 1: Zu definierende Eigenschaften einer Solr-Schema-Datei 13](#_Toc491755582)

# Literaturverzeichnis

[1]: **Chris, A.** Top 10 Search Engines In The World. [Online] 2017. https://www.reliablesoft.net/top-10-search-engines-in-the-world/.

[2]: **Kuc, R.** Top 15 Solr vs. Elasticsearch Differences. [Online] Juni 19, 2017. [Cited: August 5, 2017.] https://sematext.com/blog/2017/06/19/solr-vs-elasticsearch-differences/.

[3]: **Yigal, A.** Solr vs. Elasticsearch: Who’s The Leading Open Source Search Engine? [Online] Sept. 2016. https://logz.io/blog/solr-vs-elasticsearch/.

[4]: **McCandless, M., Hatcher, E. und Gospodnetic, O.** *Lucene in Action.* Greenwich : Manning Publications Co, 2010.

[5]: **Lucidworks.** Full Text Search Engines vs. DBMS. [Online] Sept. 2009. https://lucidworks.com/2009/09/02/full-text-search-engines-vs-dbms/.

[6]: **GmbH, Sage.** Das Unternehmen. [Online] 2017. http://www.sage.de/ueber-uns/ueber-sage/unternehmensprofil.

[7]: **Grainger, T. und Potter, T.** *Solr in Action.* New York : Manning Publications Co., 2014.

[8]: **Edlich, S. et al.** *NoSQL - Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken.* Münschen : Carl Hanser Verlag, 2011.

[9]: **Solr, Apache.** Apache Solr Reference Guide. [Online] Juni 2017. https://lucene.apache.org/solr/guide/6\_6/.

[10]: **Goll, J. and Dausmann, M.** *Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java.* Wiesbaden : Springer Vieweg, 2013.

[11]: **Fettke, P.** Client-Server-Architektur. [Online] Sept. 2016. http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/is-management/Systementwicklung/Softwarearchitektur/Architekturparadigmen/Client-Server-Architektur.

[12]: **Microsoft.** The MVVM Pattern. [Online] Feb. 2012. https://msdn.microsoft.com/en-us/library/hh848246.aspx.

[13]: **Fowler, M.** GUI Architectures. [Online] Juli 2006. https://martinfowler.com/eaaDev/uiArchs.html.

[14]: **Wikipedia.** Apache Solr - Integrating Solr. [Online] Aug. 2017. https://en.wikipedia.org/wiki/Apache\_Solr#Integrating\_Solr.

[15]: **Wikipedia**. Hasfunktion. [Online] Juni 2017. https://de.wikipedia.org/wiki/Hashfunktion.

[16]: **Sieben, J.** Informatik Aktuell. [Online] Oktober 20, 2015. [Cited: Juli 4, 2017.] https://www.informatik-aktuell.de/entwicklung/programmiersprachen/to-trigger-or-not-to-trigger.html.

[17]: **Miller, B.** How to Download and Compile Solr 6 in Eclipse. [Online] Mai 2016. http://coding-art.blogspot.de/2016/05/how-to-download-and-compile-solr-6-in.html.

[18]: **Fowler, M.** Data Transfer Object. [Online] https://martinfowler.com/eaaCatalog/dataTransferObject.html.

[19]: **Osmani, A.** Learning Javascript Design Patterns. [Online] 2017. https://addyosmani.com/resources/essentialjsdesignpatterns/book/.

[20]: **Zala, A.** An Introduction To The HTML5 History API. [Online] Okt. 2014. https://code.tutsplus.com/tutorials/an-introduction-to-the-html5-history-api--cms-22160.

[21]: **Schiering, M.** SEO und SEA im Kreuzfeuer von AJAX und Single-Page-Applications. [Online] Okt. 2015. https://www.mso-digital.de/seo-und-sea-im-kreuzfeuer-von-ajax-und-single-page-applications/.

[22]: **Microsoft.** Übersicht über ASP.NET MVC. [Online] 2017. https://msdn.microsoft.com/de-de/library/dd381412(v=vs.108).aspx.

[23]: **Solr, Apache.** Solr. [Online] 2017. http://lucene.apache.org/solr/.

[24]: **Wikipedia.** Unstrukturierte Daten. [Online] https://de.wikipedia.org/wiki/Unstrukturierte\_Daten.

[25]: **W3C.** Lokalisierung vs. Internationalisierung. [Online] Sept. 2010. https://www.w3.org/International/questions/qa-i18n.de.php.

[26]: **Wikipedia.** Webanwendung. [Online] Juli 2017. https://de.wikipedia.org/wiki/Webanwendung.

[27]: **Wikipedia**. Subskription - Software. [Online] Juni 2017. https://de.wikipedia.org/wiki/Subskription#Software.

[28]: **Wikipedia.** Softwarearchitektur. [Online] Mai 2017. https://de.wikipedia.org/wiki/Softwarearchitektur.

[29]: **Wikipedia**. Programmierschnittstelle. [Online] Mai 2017. https://de.wikipedia.org/wiki/Programmierschnittstelle.

[30]: **Wikipedia**. Anwendungsserver. [Online] Aug 2017. https://de.wikipedia.org/wiki/Anwendungsserver.

[31]: **Wikipedia**. Hypertext Transfer Protocol. [Online] Juli 2017. https://de.wikipedia.org/wiki/Hypertext\_Transfer\_Protocol.

[32]: **Wirtschaftslexikon, Gabler.** URL. [Online] 2017. http://wirtschaftslexikon.gabler.de/Archiv/55200/url-v8.html.

[33]: **Wikipedia.** Framework. [Online] März 2017. https://de.wikipedia.org/wiki/Framework.

[34]: **selfhtml.** JavaScript/ Anwendung und Praxis/ Webanwendung. [Online] Feb. 2017. https://wiki.selfhtml.org/wiki/JavaScript/Anwendung\_und\_Praxis/Webanwendung.

[35]: **selfhtml**. JavaScript/ Ajax. [Online] Juli 2017. https://wiki.selfhtml.org/wiki/JavaScript/Ajax.

[36]: **Wikipedia.** Message-Digest Algorithm 5. [Online] Aug. 2017. https://de.wikipedia.org/wiki/Message-Digest\_Algorithm\_5.

[37]: **Wirtschaftslexikon, Gabler.** Primärschlüssel. [Online] 2017. http://m.wirtschaftslexikon.gabler.de/Definition/schluessel.html?referenceKeywordName=Prim%C3%A4rschl%C3%BCssel.

[38]: **Microsoft.** Exemplarische Vorgehensweise: Erstellen einer Website mit Razor-Syntax in Visual Studio. [Online] 2017. https://msdn.microsoft.com/de-de/library/gg606533(v=vs.100).aspx.

[39]: **Wirtschaftslexikon, Gabler.** Datenbankmanagementsystem (DBMS). [Online] http://wirtschaftslexikon.gabler.de/Archiv/74907/datenbankmanagementsystem-dbms-v9.html.

[40]: **Microsoft.** Was ist eine DLL? [Online] Mai 2017. https://support.microsoft.com/de-de/help/815065/what-is-a-dll.

[41]: **Kecher, C. and Salvanos, A.** *UML 2.5. Das umfassende Handbuch.* Bonn : Rheinwerk Verlag GmbH, 2015.

[42]: **Oettinger, R.** Was ist eine Softwarelizenz. [Online] Juni 2014. https://www.computerwoche.de/a/was-ist-eine-softwarelizenz,1913465.

[43]: **Wirtschaftslexikon, Gabler.** Open Source. [Online] http://wirtschaftslexikon.gabler.de/Archiv/77360/open-source-v8.html.

[44]: **Wikipedia.** Microsoft Access. [Online] März 2017. https://de.wikipedia.org/wiki/Microsoft\_Access.

[45]: **Wikipedia**. Globally Unique Identifier. [Online] Juni 2017. https://de.wikipedia.org/wiki/Globally\_Unique\_Identifier.

[46]: **Schwichtenberg, Dr. H.** Erklärung des Begriffs: Internet Information Server (IIS). [Online] Dez. 2015. http://www.it-visions.de/glossar/alle/109/Der\_Internet\_Information\_Server\_IIS\_und\_ASPNET.aspx.

[47]: **Tilkov, S. et al.** *REST und HTTP - Entwicklung und Integration nach dem Architekturstil des Web.* Heidelberg : dpunkt.verlag, 2015.

[48]: **Ullenboom, C.** *Java ist auch eine Insel - Das umfassende Handbuch.* Bonn : Rheinwerk Verlag, 2010.

[49]: **Wikipedia.** Anwendungssoftware. [Online] März 2017. https://de.wikipedia.org/wiki/Anwendungssoftware.

[50]: **Wikipedia**. PL/SQL. [Online] Mai 2017. https://de.wikipedia.org/wiki/PL/SQL.

# Anhang

## Auszug aus Solr-Schema-Datei für Prototyp



## Programmcode der beiden DataSource-Klassen

Siehe Ordner *Anhang\_11\_2* auf dem Datenträger

## Konfigurationsdatei des Data Import Handler für Prototyp

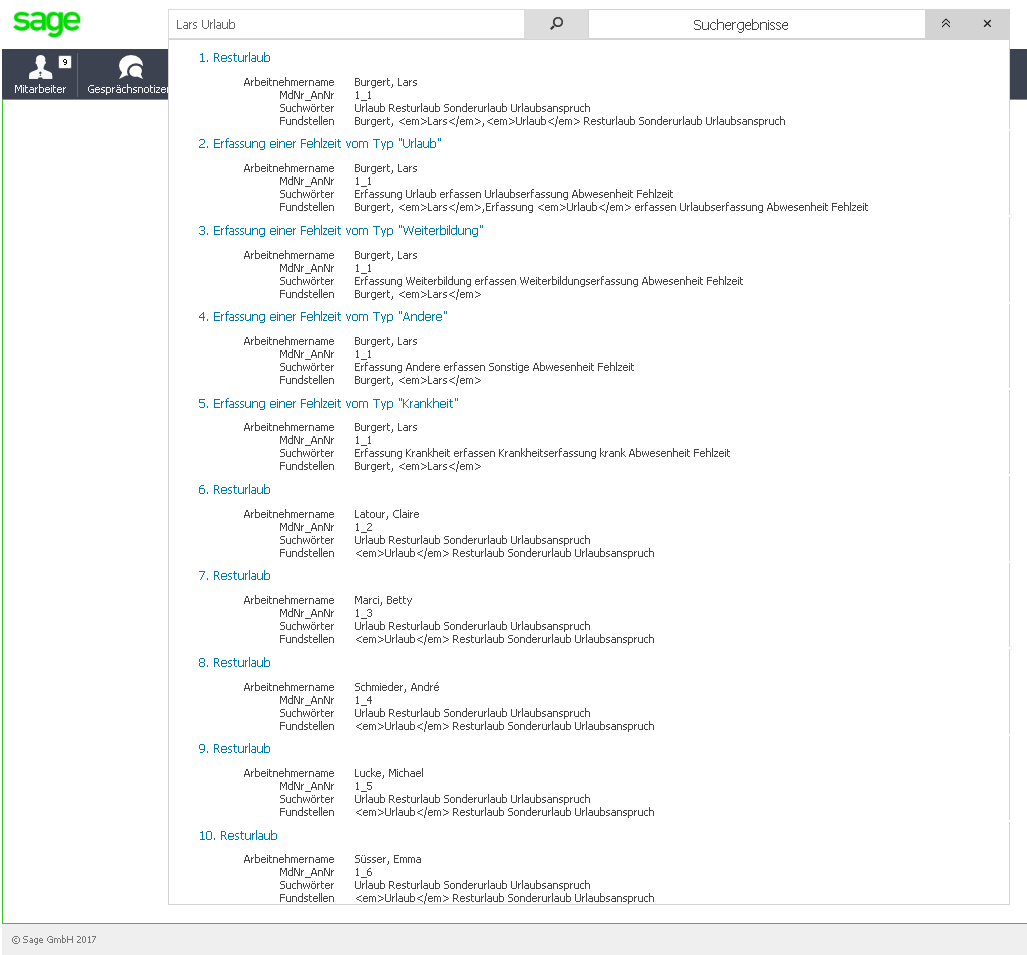
Siehe Ordner *Anhang\_11\_3* auf dem Datenträger

## Quelltext zur Benutzeroberfläche der Volltextsuche

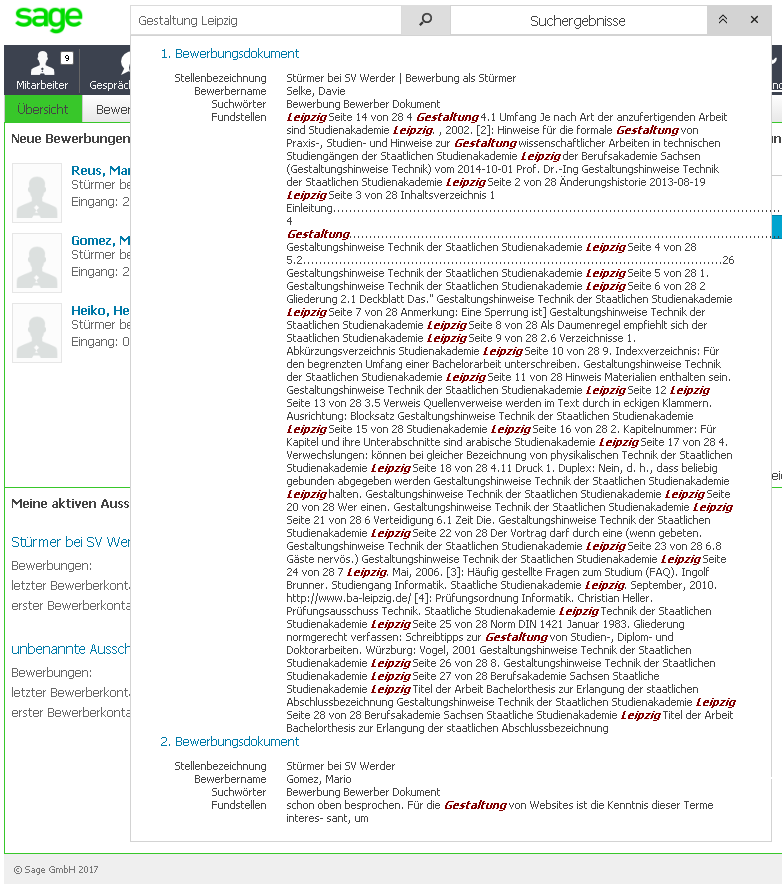
Siehe Ordner *Anhang\_11\_4* auf dem Datenträger

## Vollständige Ansicht der Suchergebnisse

### Beispiel für Kalenderübersichtmodul



### Beispiel für Bewerberdokumente

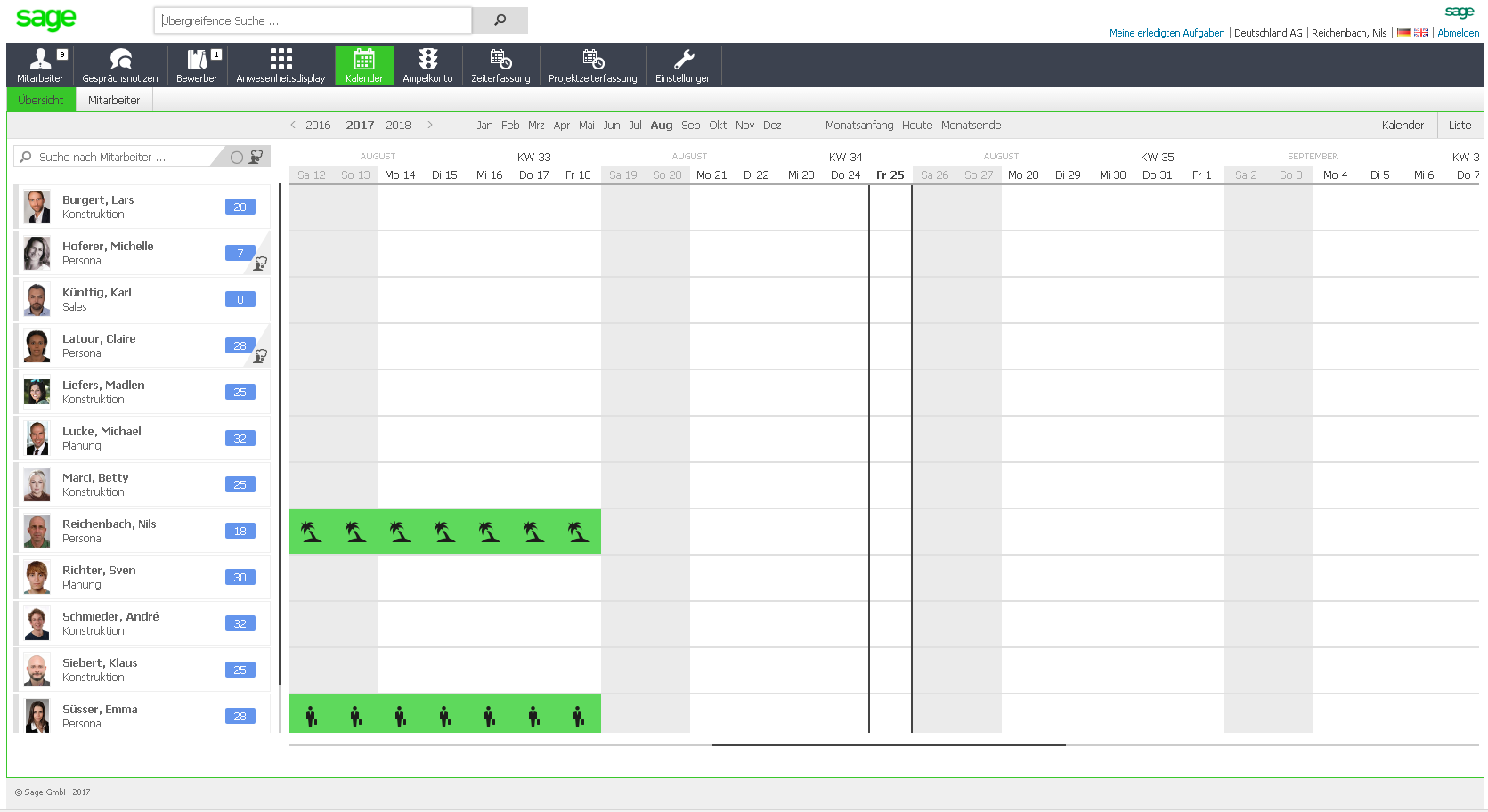


## Vollständiger Quelltext innerhalb der WebAPI zum Absenden einer Solr-Suchanfrage

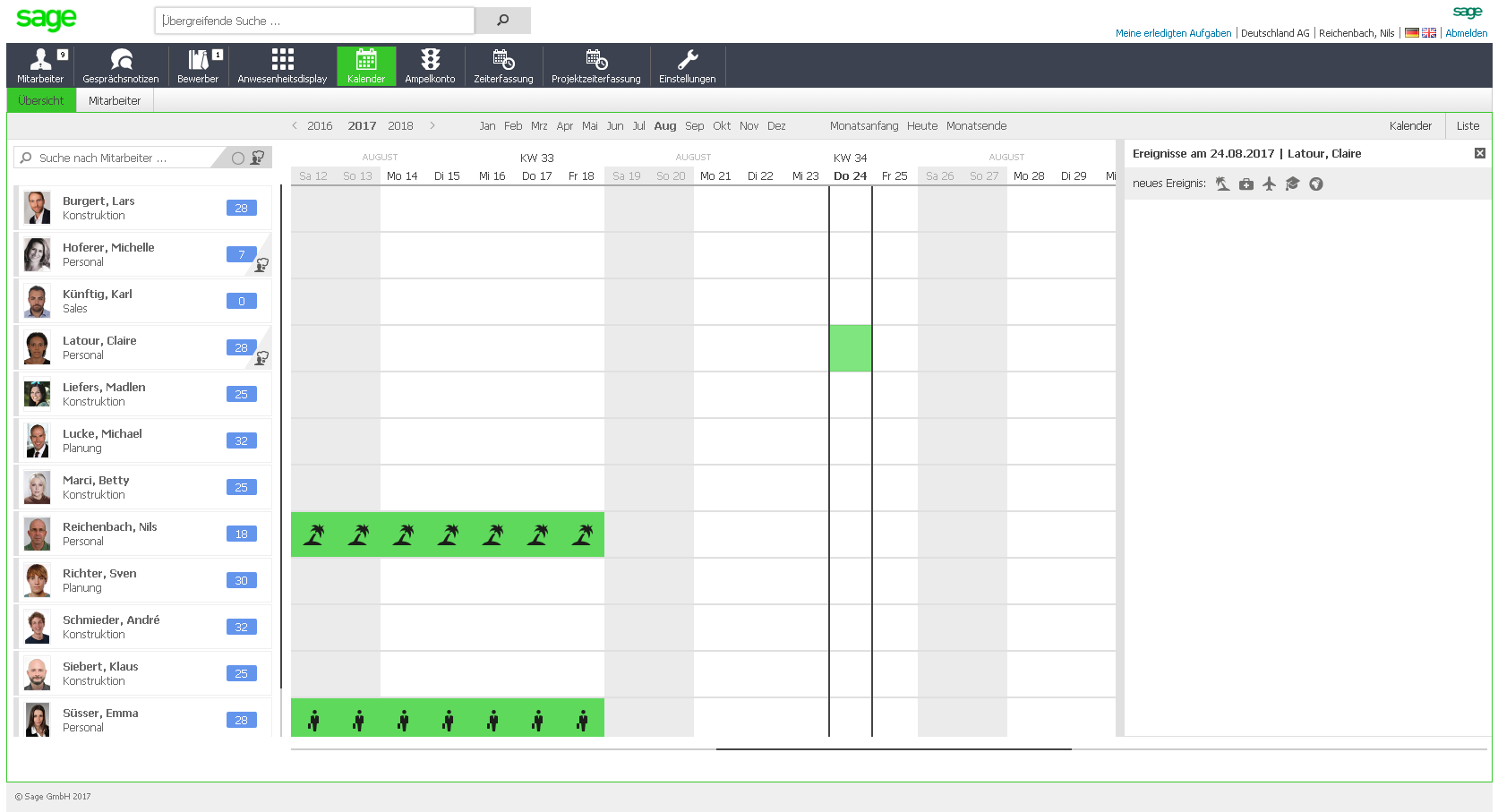
Siehe Ordner *Anhang\_11\_6* auf dem Datenträger

## Kalendermodulansichten

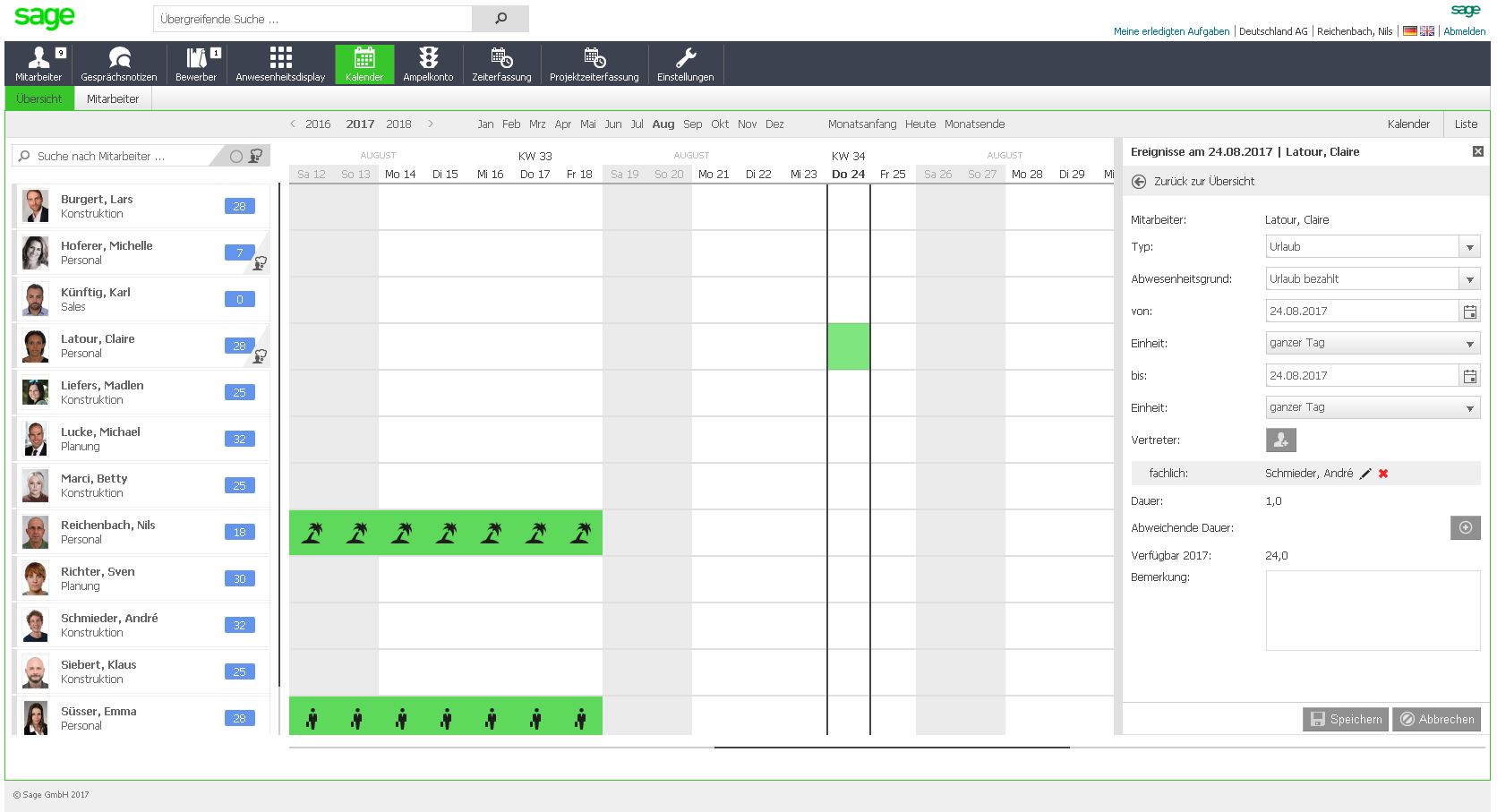
### Hauptansicht



### Ereignisliste



### Ereignisdetails



## Quelltext zur Navigation innerhalb der Kalenderübersicht

Siehe Ordner *Anhang\_11\_8* auf dem Datenträger

## Gemessene Leistungsdaten während der Indexierung und der Suche

Die Indexierung von Daten erfolgte mit Hilfe einer Testdatenbank, die 13 Mitarbeiter enthält und vier Bewerbungsdokumente. Um möglichst alle Mitarbeiter und Bewerbungen der Datenbank erfassen zu können, wird für den Zugriff auf die WebAPI ein Benutzer mit HRPortal-Universalrechten verwendet. Eine Analyse der Geschwindigkeit der Indexierung ist nicht erforderlich, es sei aber angemerkt, dass die Indexierung von 65 Dokumente für 5 verschiedene Ansichten in der Kalenderübersicht pro Mitarbeiter sowie vier Dokumenten für jedes Bewerbungsdokument insgesamt 30 Sekunden laut dem Solr-Server dauerte. Durchgeführt wurde dies auf einem Computer mit folgenden Leistungseigenschaften.

* Prozessor: Intel®Core™ i7-3770 CPU @ 3,4 GHz
* Arbeitsspeicher: 16 GigaByte
* Betriebssystem: Windows 7 Enterprise 64 Bit

Für die Abfrage der Daten zum Aufbau der 69 Dokumente waren vom DIH 79 WebAPI-Anfragen notwendig, von denen drei in einer Sekunde durchgeführt wurden.

Um die Dauer der Indexierung von Bewerbungsdokumente unabhängig von den indexierten Dokumenten zur Navigation innerhalb der Kalenderübersicht betrachten zu können, wurde zusätzlich eine PDF der Größe 6,2 MB mehrfach in Folge zur Datenbank hinzugefügt und die Indexierung vor jedem Mal vorgenommen. Die dadurch erhaltene Nettodauer nur für diese eine Datei betrug durchschnittlich ca. 6 Sekunden. Jedoch lässt sich aus diesen Daten keine genauen Rückschlüsse, wie lange der Indexierungsprozess für deutlich größere Datenmengen braucht, da dies abhängig von der Anzahl der WebAPI-Anfragen ist, die aufgrund der unterschiedlichen Aufbaukomplexität von Dokumenten sich beträchtlich unterscheiden können. Der Idealfall wäre pro Dokument nur ein WebAPI-Aufruf zur Datenabfrage.

Neben der Geschwindigkeit der Indexierung ist auch die Ausführungsdauer von Suchanfragen auf dem Solr-Server interessant. Sämtliche Suchanfragen, die eine mehrere verschiedene Ergebniskonstellationen von Dokumenten ergaben, bewegten sich im einstelligen Millisekundenbereich. Je größer die Datenmenge ist, desto länger ist die Suchdauer. Eine Einschätzung der Leistungsdaten von Solr im Einsatz für das HR Portal kann jedoch erst nach einer vollständigen Implementierung und mit einer hinreichend großen Datenmenge vorgenommen werden. Aus diesem Grund ist im Rahmen dieser Arbeit keine Aussage zur Leistungsoptimierung möglich.

# Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder veröffentlicht, noch einer anderen Prüfungsbehörde vorgelegt.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ort, Datum Unterschrift

1. Unter einer Web-Anwendung wird ein Programm verstanden, das von einem Web-Browser ausgeführt wird und demzufolge nicht auf einem Computer installiert werden muss [26]. [↑](#footnote-ref-1)
2. Das Wort *natürlichsprachig* beschreibt die Eigenschaft einer Sprache, von Menschen verwendet und verstanden zu werden. Im Gegensatz dazu gibt es u.a. maschinelle Sprachen, wie z.B. Byte-Code, die nicht vom Menschen ohne weiteres interpretierbar sind. [↑](#footnote-ref-2)
3. Open-Source stellt ein Konzept dar, nach dem Programme mit ihrem Quellcode ausgeliefert werden, der somit einseh- und veränderbar ist [43]. [↑](#footnote-ref-3)
4. Eine Desktop-Anwendung ist ein Computerprogramm, das auf einem Computer lokal installiert werden muss [49]. [↑](#footnote-ref-4)
5. Mit dem Subskriptionsverfahren im Softwarevertrieb ist die Vermietung von Anwendungen für einen bestimmten Zeitraum, zumeist ein Jahr oder weniger, gemeint [27]. [↑](#footnote-ref-5)
6. Eine Jahreslizenz enthält die für ein Jahr geltenden Bedingungen, die die Installation und Nutzung der Software regeln [42]. [↑](#footnote-ref-6)
7. API ist ein Acronym für Application Programming Interface und bezeichnet damit eine dokumentierte Programmierschnittstelle, über die externe Anwendungen auf dahinter befindliche Daten oder Geschäftslogik zugreifen können [29]. [↑](#footnote-ref-7)
8. „Eine Software-Architektur beschreibt die grundlegenden Komponenten und deren Zusammenspiel innerhalb eines Software-Systems“ [28]. [↑](#footnote-ref-8)
9. REST ist die Abkürzung für *Representational State Transfer* und bezeichnet ein Programmierparadigma für verteilte Systeme, welches eine standardisierte Grundlage für die Kommunikation zwischen Computern schafft und u.a. bei der Definition von Web-APIs als Programmierschnittstellen - für Anwendungen innerhalb eines Netzwerks - eingesetzt werden kann [47 S. 11ff.]. [↑](#footnote-ref-9)
10. HTTP steht auf englisch für *Hypertext Übertragungsprotokoll* und bezeichnet damit „ein zustandsloses Protokoll zur Übertragung von Daten auf der Anwendungsschicht über ein Rechnernetz“ [31]. [↑](#footnote-ref-10)
11. Das Framework (englisch für Rahmenstruktur) bezeichnet einen Rahmen, „innerhalb dessen der Programmierer eine Anwendung erstellt“ und gibt u.a. die Software-Architektur vor [33]. [↑](#footnote-ref-11)
12. JDBC steht „für Java Database Connectivity und bezeichnet einen Satz von Schnittstellen, um relationale Datenbanksysteme von Java zu nutzen“. Der Treiber wird benötigt, um die JDBC-API zu implementieren und um zwischen dem Java-Programm und der Datenbank zu vermitteln [48 S. 24.3]. [↑](#footnote-ref-12)
13. Eine URL bzw. *Uniform Resource Locator*, ist eine “eindeutige Identifikation bzw. Adresse eines HTML-Dokuments im Internet“ [32]. [↑](#footnote-ref-13)
14. ASP .NET MVC ist ein Framework von Microsoft basierend auf dem MVC-Architekturmuster [22]. [↑](#footnote-ref-14)
15. Der Begriff Client stammt aus der Systemarchitektur Client-Server-Architektur, welche verteilte Anwendungssysteme in den Dienstnehmer (Client) und den Dienstgeber (Server) unterteilen. Dabei können nur Clients einen Server anfragen, eine bestimmte Aufgabe für sie zu erledigen, woraufhin der Server entsprechend antwortet [11]. [↑](#footnote-ref-15)
16. Die spezifische Controller-Klasse zur View ist abgeleitet von der *System.Web.Mvc.Controller*-Klasse. [↑](#footnote-ref-16)
17. HTML (*Hypertext Markup Language*) bestimmt als Auszeichnungssprache den strukturellen Aufbau einer Internetseite. [↑](#footnote-ref-17)
18. CSS (*Cascading Style Sheet*) ist eine Formatierungssprache für HTML- und XML-Dokumente. [↑](#footnote-ref-18)
19. JavaScript, als objektorientierte Programmiersprache, ist für die dynamische Manipulation von Internetseiten zur Laufzeit notwendig. [↑](#footnote-ref-19)
20. Eine SPA ist eine Web-Anwendung, „die nur aus einem einzigen HTML-Dokument besteht und dessen gesamter Code (HTML, CSS und JavaScript) beim Seitenaufruf geladen wird“. Sämtliche Daten werden dynamisch mit Hilfe von *AJAX* nachgeladen, ohne dass der Web-Browser die HTML-Seite neu vom Server abrufen muss [34]. [↑](#footnote-ref-20)
21. „AJAX (*Asynchronous JavaScript and Xml*) ist ein Konzept, das es Web-Anwendungen ermöglicht, neue Daten vom Server zu erhalten und/oder dem Server für die weitere Verarbeitung zu senden, ohne dass die Seite als Ganzes neu geladen wird“ [35]. [↑](#footnote-ref-21)
22. Microsoft Access ist eine Anwendung, die ein relationales DBMS mit den Werkzeugen einer integrierten Entwicklungsumgebung verbindet und es ermöglicht, auch ohne Programmierkenntnisse Datenbankanwendungen zu erstellen [44]. [↑](#footnote-ref-22)
23. Kendo UI liefert u.a. JavaScript-Dateien mit aus, die u.a. ein globales kendo-JavaScript-Objekt definieren, welches sämtliche Eigenschaften, Objekte und Funktionen, die das Kendo UI Framework vorgibt, beinhaltet. Mit *global* ist die Möglichkeit gemeint, aus allen JavaScript-Objekten heraus auf das globale Objekt zugreifen zu können. [↑](#footnote-ref-23)
24. *Öffentlich* bezeichnet im objektorientierten Konzept der Datenkapselung die Sichtbarkeit von Objekteigenschaften für andere Objekte. [↑](#footnote-ref-24)
25. Webseiten, die mit der Razor-Technologie von Microsofts ASP.NET MVC-Framework arbeiten und C#-Quelltext mit integrieren weisen die spezielle Dateierweiterung .*cshtml* auf [38]. [↑](#footnote-ref-25)
26. Der Primärschlüssel ist ein Schlüssel, der einen Datensatz eindeutig kennzeichnet [37]. Im Kontext von dokumentenorientierten NoSQL-Datenbanken stellt er ein Datenfeld innerhalb des Dokuments dar, dessen Wert in der gesamten Datenbank nur einmal vorkommen darf. [↑](#footnote-ref-26)
27. Der MD5 (*Message-Digest Algorithm 5*)ist eine kryptographische Hashfunktion, „die aus einer beliebigen Nachricht einen 128-Bit-Hashwert erzeugt. Mittlerweile gilt sie nicht mehr als sicher, da es mit überschaubarem Aufwand möglich ist, unterschiedliche Nachrichten zu erzeugen, die den gleichen MD5-Hashwert aufweisen“ [36]. [↑](#footnote-ref-27)
28. GUID (*Globally Unique Identifier*) ist eine Implementierung der UUID (*Universally Unique Identifier*), die zur eindeutigen Kennzeichnung von Informationen in verteilten Systemen ohne zentrale Koordination verwendet wird [45]. [↑](#footnote-ref-28)
29. *PL/SQL* ist eine proprietäre Programmiersprache der Firma *Oracle* und bedeutet ausgeschrieben *Procedural Language/ Structured Query Language* und verbindet eine prozedurale Programmiersprache mit der Abfragesprache *SQL* [50]. [↑](#footnote-ref-29)
30. „Die UML (*Unified Modeling Language*) definiert eine allgemein verwendbare Modellierungssprache” [41 S. 20]. Das Verteilungsdiagramm veranschaulicht die physische Hardware- und Softwareumgebung sowie die Verteilung der Komponenten in dieser Umgebung [41 S. 173ff.]. [↑](#footnote-ref-30)
31. DLL steht für *Dynamic Link Library* und ist eine .NET-Bibliothek, die Code und Daten enthält und von mehreren Programmen gleichzeitig ausgeführt werden kann. Dies ermöglicht eine Aufteilung der Anwendung in separate Komponenten, die erst zur Laufzeit in das Hauptprogramm geladen werden, wenn sie benötigt werden [40]. [↑](#footnote-ref-31)
32. „Der *Internet Information Services* (IIS) ist der Webserver, den Microsoft als optionale Komponente mit allen Betriebssystemen seit Windows 2000 ausliefert [46]. [↑](#footnote-ref-32)
33. Lokalisierung bedeutet in der Softwareentwicklung die Anpassung eines Produkts, einer Anwendung oder eines Dokuments an verschiedene Sprachen oder kulturelle Anforderungen eines bestimmten Zielmarkts [25]. [↑](#footnote-ref-33)
34. Eine .NET Assembly ist das kompilierte Produkt einer .NET-Anwendung und stellt ein Paket dar, in dem alle Dateien zum Ausführen der Anwendung enthalten sind. [↑](#footnote-ref-34)
35. DTO bedeutet ausgeschrieben *Data Transfer Object* und bezeichnet Objekte, die Daten zwischen Prozessen transportieren, um die Anzahl an Methodenaufrufen zu reduzieren, die sich ergeben würden, wenn die Daten einzeln übergeben werden würden. DTOs beinhalten keine Logik sondern nur Eigenschaften, die Daten speichern [18]. [↑](#footnote-ref-35)
36. Ein Datenbankmanagementsystem (DBMS) stellt den Teil eines Datenbanksystems dar, der zur Verwaltung der Daten der Datenbank dient [39]. [↑](#footnote-ref-36)