Bachelorarbeit

Zur Erlangung des Grades
Bachelor of Science (B. Sc.) Wirtschaftsinformatik

an der Wirtschafts- und Sozialwissenschaftlichen Fakultät
und
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

Architekturerstellung und Implementierung eines Systems zur Visualisierung und Analyse von Auslastungszeitreihen

Vorgelegt von:

Nicolas Walk
Matrikelnummer: 773353

Studiengang:
Wirtschaftsinformatik

Betreuung:
Dr. Eldar Sultanow
Dr. André Ullrich

Inhalt

Vorwort

Während meines Studiums der Wirtschaftsinformatik an der Universität Potsdam weckte die Problematik der Softwarearchitektur mein besonderes Interesse. Als dann zur Bearbeitung der vorliegenden Untersuchung die endgültige Thematik dazu vorlag und ein Begutachter einen konkreten Untersuchungsgegenstand aus der Praxis offerierte, war ich mit großer Begeisterung in das Thema eingestiegen.

Die Auslastungsdaten von Großrechenzentren mit verschiedensten Kenngrößen und deren jeweiligem Auslastungsgrad standen zur Untersuchung an.

Konkret ging es darum, die Auslastung des Rechenzentrums der Bundesagentur für Arbeit zu überwachen. Hierbei geht es insbesondere um die Auswertung und graphische Darstellung der vorhandenen Daten. Die Herausforderung bestand hierbei, die entsprechenden Zeitreihen aus diesen Daten graphisch so aufzubereiten, dass Anomalien bzw. Störungsursachen schnell identifiziert und behoben bzw. beseitigt werden können.

Um einen userfreundlichen Lösungsansatz zu entwickeln, wurden für die vorliegende Untersuchung zahlreiche und umfangreiche Implementierungen abgeleitet und entwickelt.

Mein Dank für die motivierende Unterstützung gilt der Betreuung. Ganz besonders bedanken möchte ich mich bei Dr. Eldar Sultanow für die hilfreichen Anmerkungen und seine Geduld bei den zahlreichen Diskussionen zur Entwicklung der Lösungsansätze.

Berlin, 06.09.2019

Nicolas Walk

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AWS	Amazon Web Serivces
CPU	Central processing unit
CSS	Cascade Style Sheet
CSV	Comma-seperated values
DOM	Document Object Model
GPU	Graphic processing unit
HTML	Hypertext Markup Languge
JS	JavaScript
JSON	JavaScript Object Notation
MVC	Model View Controller
NFA	Nicht funktionale Anforderungen
POC	Proof of Concept
REST	Representational State Transfer
SVG	Scalable Vektor Graphics
UI	Userinterface
URL	Uniform Resource Locator
XML	Extensible Markup Language

1 Einleitung

Nicht nur im unternehmerischen Umfeld ist die Bedeutung von Rechenzentren in einer digitalisierten Welt stark gestiegen. Diese sind teilweise sogar direkt für den unternehmerischen Erfolg verantwortlich und dabei ist es höchst sensibel, diese entsprechend zu verwalten und deren Performance zu überwachen [Gilbert et al. 2013, S. 3].

Beim Betreiben des Rechenzentrums kann es beispielsweise lastbedingt zu außerplanmäßigen Vorfällen kommen. Diese müssen, besonders wenn es sich um Business Services mit unternehmenskritischen Funktionen handelt, frühzeitig erkannt werden, um ernsthaften Störungsvorfälle vorzubeugen oder diese zeitnah zu beseitigen.

Dies erfordert eine Analyse der zugrundeliegenden Metriken des Rechenzentrums. Bei diesen Metriken handelt es sich beispielsweise auch um den Ressourcenverbrauch [Gilbert et al. 2013, S. 6]. Ein Indikator wie der Ressourcenverbrauch wird in Zahlen ausgedrückt. Somit besteht die Notwendigkeit einer entsprechenden Aufbereitung dieser Daten. Diese Aufbereitung muss den Zweck erfüllen, Unstimmigkeiten aufzudecken, um die Funktionsfähigkeit der Services insgesamt garantieren zu können. Diese Aufbereitung kann auch mitunter in einer graphischen Darstellungsform erfolgen und sollte aber den Zusammenhang zwischen zeitlicher Abfolge und den generierten Kennzahlen herstellen.

1.1 Problemstellung und Zielsetzung

Die Bundesagentur für Arbeit hostet drei hochverfügbare Datacenter, welche aus mehr als 10.000 einzelnen Servern bestehen [Chircu et al. 2019, S. 15].

Bei den Auslastungsdaten eines Rechenzentrums in der entsprechenden Interpretation handelt es sich um eine Sammlung verschiedenster Kenngrößen zur Beschreibung des jeweiligen Auslastungsgrades des ausgewählten Rechenzentrums.

Ein Rechenzentrum ist die gesammelte Rechentechnik beziehungsweise Infrastruktur beispielsweise einer Unternehmung, welches jeweils zu einem bestimmten Zeitpunkt

einen bestimmten Grad der Auslastung aufweist. Die Auslastung lässt sich an verschiedenen Merkmalen messen. Ein Beispiel wäre dabei die prozentuale Auslastung der CPU oder auch die gemessene Temperatur im Inneren des Systems. Von Auslastungszeitreihen wird hier gesprochen, wenn in solch einem System für jeden Zeitpunkt T ein Auslastungswert beziehungsweise eine Beobachtung existiert [Rainer Schlittgen 2001, S. 1]. Bei der Visualisierung von Zeitreihen ist also im Besonderen die strikte zeitliche Reihenfolge der Daten zu beachten. Daneben muss zwischen den Größen Zeit und Beobachtung unterschieden werden. Dafür müssen insbesondere bei der Wahl der Zeitskala adäquate Werte gewählt werden [Dr. Chun-houh Chen 2006, S.74]. Ändert sich eine Beobachtung nur alle zehn Minuten, würde die Darstellung der Änderungen alle zwei Minuten keinen Mehrwert erzeugen.

Zur Darstellung und Auswertung solcher Zeitreihen existieren bereits verschiedene Tools und Anwendungen. Zu den Kernaufgaben gehört, die Daten graphisch so aufzubereiten, dass der User auf möglichst einfache Weise, die von ihm benötigten Informationen schnell erfassen und auswerten kann.

Die Bundesagentur für Arbeit verwendet zur Überwachung der Auslastung ihres Rechenzentrums die Anwendungen Kibana sowie Grafana. Ein Problem, das unter der Verwendung dieser Tools entsteht, ist die Schwierigkeit, in der zur Verfügung gestellten Aufbereitung der Daten die gewünschten Informationen effizient extrahieren zu können. Des Weiteren geht mit der Verwendung verschiedener Tools das Problem einher, dass zum Auffinden der Ursache einer Anomalie, zusätzliche Tools aufgerufen werden müssen, was einer userfreundlichen Verwendung entgegensteht.

Aus diesen Gründen soll ein neues Tool entwickelt werden, mit welchem es möglich ist, die Auslastung des Data Centers der Bundesagentur für Arbeit schnell, effizient und einfach überwachen zu können. Damit soll ermöglicht werden, Störfälle und Anomalien sofort erkennen zu können und doch eine hohe Integrierbarkeit in bereits bestehende Anwendungen zu erreichen. Ziel dieser Untersuchung ist es hier, eine Softwarearchitektur so zu entwickeln, dass eine Implementierung solch einer Anwendung wie oben gefordert ermöglicht wird.

1.2 Methodisches Vorgehen

In der vorliegenden Untersuchung soll eine Softwarearchitektur entwickelt werden. Zum Vergleich der verschiedenen hier entwickelten Ansätze, die in Kapitel 6 vorgestellt werden, werden die Architekturen hinsichtlich bestimmter Kriterien gegenübergestellt und diskutiert. Diese zugrundeliegenden Kriterien leiten sich von der ISO-Norm 25010 ab, welche die Qualität von Software als Produkt beschreibt. Die Merkmale, welche bei der vorliegenden Untersuchung besondere Beachtung finden, sind: Performance, Reliability, Usability und Maintainability. Diese werden in Kapitel 2 ausführlich diskutiert und deren Auswahl begründet.

Im ersten Schritt wird der Untersuchungsgegenstand diskutiert. Es wird die Problemstellung erörtert und die Zielsetzung festgelegt.

Dann werden spezielle Anforderungen an das System diskutiert und analysiert. Anschließend werden bereits bestehende Lösungen zur Visualisierung und Analyse von Auslastungszeitreihen diskutiert und bewertet, das heißt mit den gesetzten Anforderungen verglichen.

Nach diesen vorbereitenden Untersuchungen und darauf aufbauend wird ein erster Grobentwurf für eine Softwarearchitektur entwickelt und diskutiert. Anschließend wird eine Begründung zum Aufbau und der Designentscheidungen dieses Grobentwurfs dargelegt. Auf dieser Basis werden dann diverse Architekturen entwickelt und deren verwendete Technologien beschrieben und hinsichtlich ihrer Vor- und Nachteile mit den gesetzten Qualitätskriterien diskutiert.

Auf dieser Basis erfolgt anschließend eine Validierung hinsichtlich der Performance und Skalierbarkeit der entwickelten Architekturen.

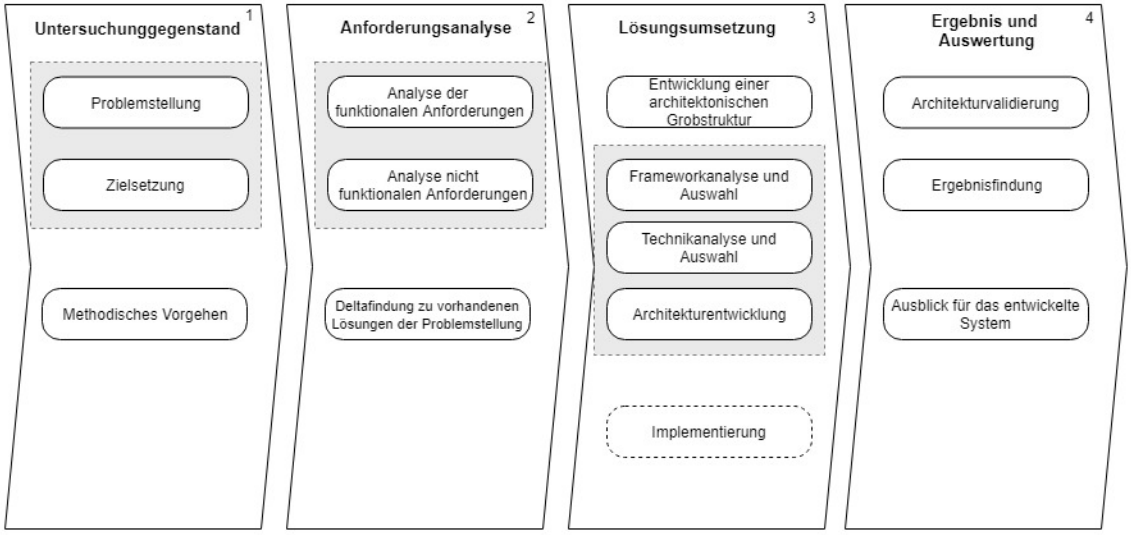

Abbildung 1: methodisches Vorgehen [eigene Darstellung]

2 Anforderung an das System zur Visualisierung

In diesem Kapitel werden die Anforderungen an ein System zur Visualisierung und Analyse von Zeitreihendaten diskutiert, analysiert und vorgestellt. Diese gliedern sich in funktionale sowie nicht funktionale Anforderungen. Funktionale Anforderungen beschreiben Funktionsweisen und Attribute, die auf technischer Ebene Elemente beschreiben, welche Teil der Eingabe, der Verarbeitung und der Ausgabe sind. [Robertson & Robertson 2011, p. 9]

Nicht funktionale Anforderungen beschreiben mit welcher Qualität beziehungsweise in welchem Ausmaß das zu entwickelnde System, die funktionalen Anforderung dann umsetzen muss. [Böhm & Fuchs 2015, S. 139]

Zur Analyse der funktionalen Anforderungen werden im folgenden Kapitel Use Cases diskutiert, welche das Verhalten des Systems in bestimmten Situationen unter bestimmten Eingaben beschreiben und veranschaulichen.

2.1 Funktionale Anforderungen

Zur Erhebung und zum besseren Verständnis der Anforderungen wurde mit Mitarbeitern der Bundesagentur für Arbeit, welche mit der Aufgabe des Monitorings der Datencenter vertraut sind, qualitative Interviews durchgeführt. Die Auswertungen dieser Interviews führten zu dem Ergebnis, dass die zu entwickelnde Anwendung verwendet werden soll, um zum Beispiel Anomalien festzustellen und den Grund für diese schnell identifizieren zu können. Es soll ein genauer und schneller Überblick über den Betriebszustand des Datencenters gewonnen werden können. Außerdem soll die zu implementierende Anwendung es ermöglichen, durch die Verwendung einer fortgeschritteneren Visualisierungstechnologie, Anomalien für einen menschlichen Benutzer schneller erkennbar zu machen. Das System zur Visualisierung und Analyse von Auslastungszeitreihen muss im Bezug zur Analyse, zum Beispiel die Auslastung eines Servers einer beliebigen Metrik wie die Anzahl der Threads oder die Anzahl der Requests jedes Servers des Systems zu einem bestimmten Zeitpunkt visualisieren können [Chircu et al. 2019, 19 ff].

Dies erfordert, dass dem User ein Userinterface zur Verfügung gestellt wird, über welches es ermöglicht wird, die jeweilige Metrik auszuwählen. Denn für jeden Visualisierungsoutput ist es notwendig, dass die Metriken der zu visualisierenden Information eingestellt werden können. Da die Anzahl von Parametern schon bei einfachen Daten sehr hoch ausfällt, stellt dann die Wahl der Darstellung, aber auch die der darzustellenden Daten eine zentrale Anforderung dar [Chen et al. 2008, p. 84].

Die Quelle der Auslastungszeitreihen soll frei konfigurierbar sein. Dies erfordert, dass der User diese in der Anwendung auswählen und konfigurieren kann.

Ein weiterer Aspekt der Anforderungen liegt in der Aufbereitung der Daten. Dies verlangt eine Visualisierung. Diese muss in einem Maße konfigurierbar sein, dass für die zu extrahierende Information eine möglichst hinreichende Darstellungsform gewählt werden kann. Dies gewährleistet ein effizientes Ablesen der geforderten Informationen. Zur Umsetzung der Forderung nach einem schnelleren Erkennen von Anomalien durch Nutzung fortgeschrittener Visualisierungstechnologien, soll die Visualisierung in einer dreidimensionalen Darstellung umgesetzt werden. Die Nutzung einer dritten Dimension ermöglicht es, mehr Informationen in einer einzigen Visualisierung darzustellen.

Die Darstellung wird in einer Stadtmetapher umgesetzt. Hierbei repräsentiert jedes Gebäude einen Server, dabei werden die Häuser als Blöcke erfasst, welche die Cluster darstellt. Die Höhe der Gebäude entspricht dann der Auslastung einer Metrik der Server [Chircu et al. 2019, S. 17].

Mit der gewählten Stadtmetapher geht einher, dass der Zeitpunkt, welcher dargestellt wird, gewechselt werden können muss. So stellt die Visualisierung nur die Auslastung des Daten Centers zu einem bestimmten Zeitpunkt dar, dieser muss konfigurierbar sein, sodass eine Entwicklung der Auslastung betrachtet werden kann. Auch muss im System definierbar sein, wie breit beziehungsweise lang die Stadt wird. Also wie groß darf jeweils ein Block sein. Bezogen auf die Server heißt das, wie viele können nebeneinander dargestellt werden.

Im Folgenden werden die involvierten Akteure und die aus den Anforderungen abgeleiteten Use-Cases genauer beschrieben und analysiert.

2.1.1 Involvierte Akteure

Zur Aufstellung der Use Cases werden die aus der Anforderung entnommenen, am System beteiligten Akteure beschrieben.

Es gibt nur einen menschlichen Akteur, den User in nur einer Ausprägung.

Des Weiteren gibt es einen Systemakteur. Das ist die Datenbank, welche die zu visualisierenden Daten persistiert. Diese dient dazu, dem System die notwendigen Daten zur Verfügung zu stellen.

2.1.2 Use Cases

Aufgrund der analysierten Anforderungen werden im Folgenden adäquate Use-Cases identifiziert. Diese können der Abbildung 2 entnommen werden.

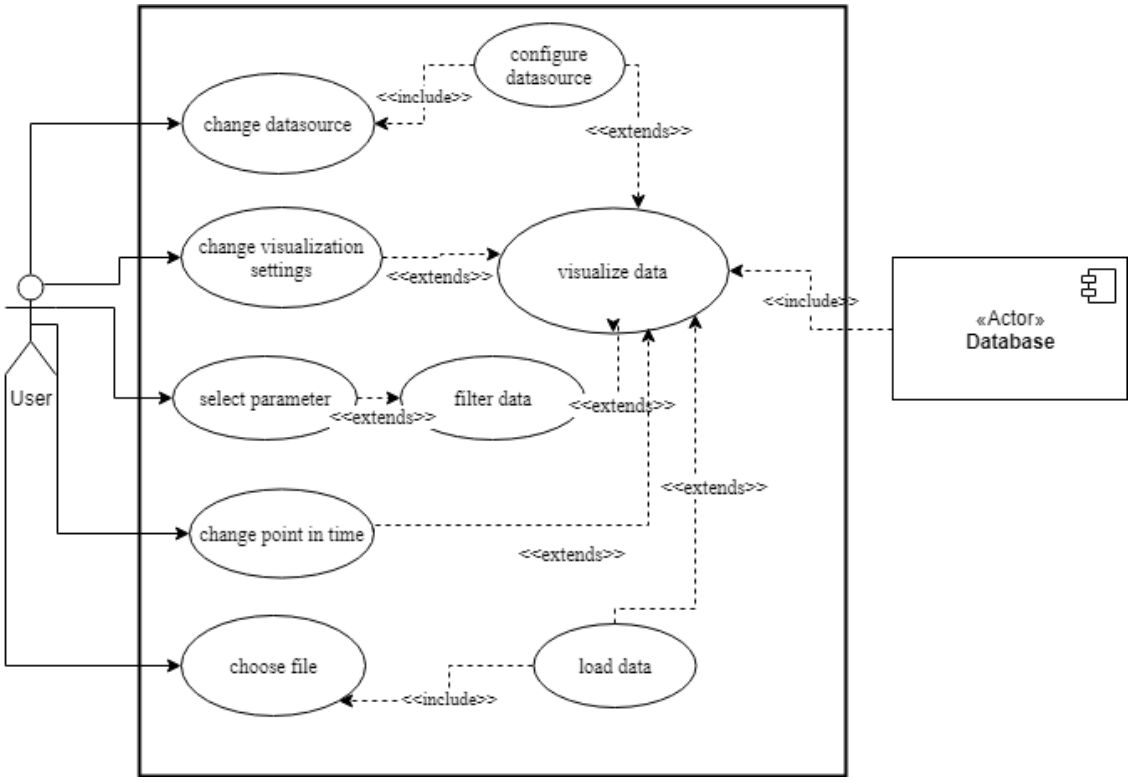

Abbildung 2: Use-Case Diagramm [eigene Darstellung]

2.1.3 Use-Case Spezifikation

Die Use-Cases „User change visualization settings“ und „User change point in time“ wurden verkürzt dargestellt. Es könnten beispielsweise weitere Use-Cases daraus abgeleitet werden, wie die Visualisierung angepasst werden soll. Eine Möglichkeit wäre „change color“ bei welcher die Farben der Darstellung angepasst werden können, oder auch „change scale“ wodurch die Skalierung der Visualisierung verändert werden kann. Gleiches gilt für „User change point in time“ bei welchem der Zeitpunkt durch eine Eingabemöglichkeiten verändert werden könnte.

Da in dieser Untersuchung jedoch ein Fokus auf die Implementierung, die verwendeten Technologien und die Architekturentwicklung gelegt werden soll, werden diese Use-Cases nicht in einer kleineren Detailstufe beschrieben, da dies auf die im Vordergrund stehende Architekturentwicklung und auch die verwendete Technologie keinen weiteren Einfluss hat.

2.2 Nicht funktionale Anforderungen

Wie bereits in Kapitel 2.2 beschrieben, dienen nicht funktionale Anforderungen der Feststellung des Ausmaßes der Qualität, in welcher die funktionalen Anforderungen umgesetzt werden müssen. Zur Einordnung werden im Folgenden Qualitätsmaßstäbe für Software beschrieben, diskutiert und analysiert. Diese sind in der ISO Norm 25010 beschrieben und dienen der Einordnung von Qualität einer Software als Produkt.

Es handelt sich bei den Qualitätsmerkmalen einer Software als Produkt um die acht Merkmale „Functional Suitabillity“, „Performance Effenciency“, „Compatibility“, „Usability“, „Reliability“, „Security“, „Maintainability“ und „Portability“ [Iso.org ISO/IEC 25010].

Dieser Darstellung folgend und den nachfolgend beschriebenen Anforderungen, werden die NFA‘s an die zu implementierende Anwendung benannt.

Sie orientieren sich in erster Linie an den bereits identifizierten funktionalen Anforderungen. Diese bestehen, wie in Kapitel 2.1 vorgestellt, aus einer Auswahl von

Datenquellen, der Anwendung von Filtern, einer Zeitpunktauswahl sowie der dreidimensionalen Visualisierung selbst.

Die Auswahl der Datenquelle muss es ermöglichen, die zu analysierenden Daten aus beliebiger Quelle zu gewinnen. Dies erfordert einen gewissen Grad an Portabilität und Wartbarkeit. Es muss ermöglicht werden, Daten zeitnah aus den Quellen zu gewinnen, um entsprechende Analysen garantieren zu können. Dies geht auch einher mit dem Qualitätsmerkmal Performance. So muss es auch dem User ermöglicht werden, verschiedenste Quellen ohne Probleme anzubinden, was eine entsprechende Usability voraussetzt.

Auf die zu visualisierenden Daten sollten diverse Filter angewandt werden, um spezifizieren zu können, welche Daten visualisiert und auch analysiert werden sollen. Diese Anforderung bedarf dabei der Erfüllung einer Performance aufgrund der Analysefunktionen, sowie eine angemessene Nutzbarkeit der Filterung selbst. Ebenso muss diese Filterung auf Daten einer beliebigen Quelle anwendbar sein, da diese auch austauschbar sein kann. Die gleichen Anforderungen lassen sich auf die Auswahl eines Zeitpunktes anwenden, da diese in der Erfüllung der nichtfunktionalen Anforderungen eine weitere Filterung darstellt.

Die Visualisierung muss es ermöglichen, verschiedensten Informationen graphisch darzustellen. Aus diesem Grund muss sie auch austauschbar sein. Auch muss die Visualisierung, da es sich bei den möglichen Daten um potentiell große Datenmengen handelt, einen besonderen Grad der Performance erfüllen. Die Darstellung muss dann zeitnah umgesetzt werden, um auch die Möglichkeit einer Ad-hoc-Analyse garantieren zu können. Auch die Visualisierung fordert den Anspruch nach Austauschbarkeit der Datenquelle.

Daraus lässt sich schließen, dass für die Erfüllung der funktionalen Anforderungen ein besonderes Augenmerk auf die Erfüllung der Qualitätsmerkmale Maintainability, Usability, Performance, Reliability und Portability gelegt werden muss.

Security nimmt eher eine nachrangige Rolle ein, da das System in der Bundesagentur für Arbeit nur intern eingesetzt wird und nur intern zugänglich ist.

3 Vorhandene Visualisierungs- und Analyselösungen

In diesem Abschnitt werden ausgewählte Anwendungen zur Visualisierung von Auslastungszeitreihen vorgestellt, analysiert und diskutiert. Darüber hinaus werden diese Anwendungen zugleich auf ihren Aufbau und ihre Möglichkeiten in Bezug auf die an das System gestellten Anforderungen untersucht und verglichen.

3.1 Grafana

Bei Grafana handelt es sich um eine Open-Source Metriken-Analyse und Visualisierungssuite. Eines der Haupteinsatzgebiete ist die Visualisierung von Zeitreihendaten zur Infrastruktur und Anwendungsanalyse [Grafana Docs]. Grafana baut als Open Source Anwendung unter der Apache 2 Lizenz auf einen breiten Support mit ca. 900 Contributern und rund 22.000 Commits auf Github [Grafana license]. Auch kann Grafana zugleich auf eine große Anzahl an namhaften Nutzern wie zum Beispiel Paypal, Intel, oder auch Ebay zurückgreifen.

Auf der technischen Seite unterstützt Grafana verschiedene Datenquellen wie zum Beispiel Graphite, Prometheus, MySQL und auch AWS Cloudwatch. Wie eingangs schon erwähnt unterstützt Grafana jedoch nicht die Darstellung multipler Werte zu einem bestimmten Zeitpunkt. Dies liegt an einer fehlenden dreidimensionalen Darstellungsform, welche durch die zusätzliche Dimension mehr Informationen transportieren kann, aber auch am Aufbau der Grammatik unter welcher die Zeitreihen in Grafana definiert werden.

Es wird in der Definition der zu visualisierenden Daten eine zeitliche Spanne angegeben, welche verschiedene Zuordnungen beinhaltet. Dies lässt dann die Darstellung von Veränderungen nur einer Instanz zu einem bestimmten Zeitpunkt zu.

Zwar machen der breite Support aus der Community der Open Source Lösung Grafana, aber auch die namenhaften Nutzer Grafana zu einer verlässlichen Lösung, jedoch werden hier nicht alle geforderten funktionalen Anforderungen erfüllt.

3.2 Kibana

Kibana ist eine in Javascript geschriebene Analyseplattform und eine der am meisten verbreiteten Open Source Visualisierungs-Komplettlösungen. Mit ca. 400 Contributern und ca. 26.000 Commits auf Github baut es auch auf einen breiten Support und lässt dadurch auf eine langfristige Weiterentwicklung schließen. [elastic/kibana] Auch Kibana, das zusammen mit ElasticSearch den Elastic Stack bildet, kann auf eine hohe Anzahl von Industriegrößen verschiedenster Branchen als Nutzer verweisen. Hierzu zählen auch die Telekom, Volkswagen, Fitbit und Blizzard u.a. [ELK Stack Erfolgsstories | Elastic Customers]. Die weite Verbreitung positioniert Kibana als quasi Industriestandard und lässt die mögliche Verwendung bei Analyse und Visualisierungsproblemen unumgänglich werden.

Da Kibana im festen Zusammenspiel mit ElasticSearch arbeitet, ist die Verwendung weiterer Datenquellen eingeschränkt. So verlangt schon das Deployment von Kibana eine speziell definierte ElasticSearch Instanz.

Kibana bietet zudem eine Vielzahl an Plugins, welche zusätzliche Funktionen bieten. Die Zeitreihen können üblicherweise nicht dreidimensional dargestellt werden, jedoch kann diese Funktion durch einige Plugins zum System hinzugefügt werden. Nachteilig ist dabei die damit einhergehende Abhängigkeit von den Herstellern der Plugins und somit kann hier eine dauerhafte Unterstützung nicht gewährleistet werden.

Kibana stellt insgesamt unter Betrachtung des Supports und der Positionierung als quasi Industriestandard eine passable Lösung für die Visualisierung und Analyse dar, jedoch fällt die Beschränkung der Datenquelle, sowie die nur teilweise unterstützte dreidimensionale Visualisierung negativ ins Gewicht.

3.3 Bewertung der diskutierten Lösungen

Die vorgestellten und diskutierten Lösungen zur Visualisierung und Analyse können im Bezug auf die gestellten Anforderungen in nachfolgender Tabelle betrachtet und verglichen werden.

Tabelle 1: Vergleich vorhandener Lösungen [eigene Darstellung]

Anforderungen	Kibana	Grafana
wählbare Datenquelle	x*	✓
Filterung	✓	✓
3D Visualisierung	x	x
Zeitpunktauswahl	✓	✓

Der erste Aspekt, der auffällt, ist die fehlende Möglichkeit einer dreidimensionalen Darstellung innerhalb beider Systeme, da diese eine der Hauptanforderungen ist, welche an das System gestellt werden.

Auch stellt sich zumindest unter Kibana die Frage nach einer frei konfigurierbaren Datenquelle, da Kibana fest an eine ElasticSearch Instanz gebunden ist, und auch nur diese als Datenquelle verarbeiten kann. Dies widerspricht der geforderten Anforderung an eine frei wählbare Datenquelle, nicht zuletzt um auch eine maximale Integrationsbreite gewährleisten zu können.

Grafana unterstützt nur die Darstellung einer Wertveränderung zu einem gewählten Zeitpunkt, was wiederum einer Forderung der Darstellung widerspricht, die für alle bestehenden Instanzen die aktuellen Auslastungswerte darstellt und dementsprechend vergleichbar macht.

Was bei beiden Anwendungen im Vergleich auffällt, ist die Abhängigkeit, welche zum gewählten System besteht. Bei Kibana wird ein kompletter Stack geliefert, welcher nicht in eine andere Anwendung weiter integriert werden kann, genau wie bei Grafana. Beide Anwendungen erschweren somit einen Wechsel zu einer anderen Lösung zu einem späteren Zeitpunkt.

* Die Datenquelle ist in Kibana frei konfigurierbar unter der Verwendung von Vega (vgl. Kapitel 6.2)

4 Softwarearchitektur

In der Literatur existiert keine einheitliche Definition von Softwarearchitektur. Insbesondere nicht wo die Grenzen zwischen Architekturdesign und Anwendungsdesign liegen. [Fritz Solms 2012]

Eine mögliche Definition weist Softwarearchitektur als eine Beschreibung aus, die ein System aus Komponenten, Verbindungen und Einschränkungen abbildet, die eine Bedarfserklärung der Stakeholder und eine Begründung, die zeigt, dass das System, würde es implementiert werden, diese Anforderungen erfüllen würde, enthält [Boehm et al. 1995, S. 2]

Ein weiterer Ansatz sagt, dass das Design der Software bzw. der Implementierung der Erfüllung der funktionalen Anforderungen dient. Wohingegen das Design der Softwarearchitektur einen direkten Einfluss auf die Qualität der nicht funktionalen Anforderungen eines Systems hat [McGovern 2004, S. 76].

Folgt man diesen Definitionen, stellen sie die Qualitätssicherung als das eigentliche Ziel einer Softwarearchitektur in den Mittelpunkt. Auch wenn der Prozess der Implementierung eigentlich mit der Entwicklung einer Architektur beginnt, so handelt es sich dabei jedoch um einen iterativen Prozess, welcher im Entwicklungsprozess immer wieder Anpassungen bedarf. Die den zu Grunde liegenden Designentscheidungen werden in diesem Abschnitt diskutiert. Auch wird auf einer höheren Abstraktionsebene die Funktionsweise des Gesamtsystems sowie das Zusammenspiel der in ihm enthaltenen Komponenten dargestellt, beschrieben und diskutiert.

Im Ergebnis dieser Untersuchung wird ein Grobentwurf für die Architektur vorgestellt und diskutiert.

4.1 Anforderung an eine Architektur

Die Anforderung an die Architektur orientiert sich an der Erfüllung einer Qualität von Software, wie sie im Kapitel 2.2 beschrieben wurde.

In Anlehnung an die in Kapitel 4 vorgestellten Definitionen von Softwarearchitektur dient das Design der Softwarearchitektur zur Erfüllung der Qualität und insbesondere der Qualität der nicht funktionalen Anforderungen des zu implementierenden Systems und wird auch in der Entwicklung der einzelnen Designentscheidungen berücksichtigt. Die Vorgehensweise orientiert sich an den in Kapitel 2.2 beschriebenen Qualitätsmerkmalen einer Software als Produkt nach ISO 25010.

Die zu entwickelnde Softwarearchitektur muss gewährleisten, dass das Produkt wartbar ist. Das heißt wie komplex ist es, Änderungen vorzunehmen. Seien es Änderungen aufgrund neuer Spezifikationen, Ausbesserungen oder Änderungen folgend aus einer Änderung der Anwendungsumgebung. Weiter muss eine hinreichende Bedienbarkeit gewährleistet werden. Das heißt, die Benutzung der Software durch den Benutzer. Ein weiterer Faktor betrifft die Funktionalität. Dies bezieht sich hier auf die fachlichen und funktionalen Anforderungen an das System. Insbesondere muss ein Schwerpunkt auf die Erfüllung der Performance gelegt werden, da explizit gewährleistet sein muss, Anomalien zeitnah ablesen zu können. Darüber hinaus existiert bereits eine Kibana Anwendung, bei welcher es wünschenswert wäre, dass das neue System für eine Integration in diese Anwendung offen ist. Auch soll ein nachhaltiges Produkt entwickelt werden, sodass ein langfristiger Support der verwendeten Technologien gewährleistet werden muss.

Wie in Kapitel 2.2 diskutiert, wird bei der Entwicklung einer weiteren Architektur ein besonderes Augenmerk auf die Erfüllung der ISO 25010 Qualitätsmerkmale Maintainability, Portability Usability und Performance Efficiency sowie Reliability gelegt.

4.2 Referenzarchitekturen und Architekturmuster

Eine Referenzarchitektur beschreibt ein Muster, welches für bestimmte Probleme herangezogen wird und für diese Klasse an Problemen als Musterlösung gilt. Bei der Visualisierung von Zeitreihen gibt es zwei etablierte Softwarelösungen. Kibana und Grafana sind bewährte browserbasierte Anwendungen. Beide basieren auf dem Architekturmuster des Model View Controllers, welches von dem Entwickler Trygve Reenskaug 1979 das erste Mal beschrieben wurde. Es dient dazu, den Datenzugriff von

der fachlichen Logik sowie von der Darstellung zu entkoppeln. Es besteht aus den Komponenten Model, View und dem Controller. Innerhalb der Model Komponente werden die Daten repräsentiert. Die View Komponente stellt diese Daten dar. Der Controller sorgt dafür, dass die Handlungen der User vom Model durchgeführt und von der View dargestellt werden [Robert Eckstein 2007].

Den Referenzprodukten Kibana, sowie Grafana folgend sollte sich die zu implementierende Visualisierungs- und Analyseanwendung nach dem Model-View-Controller als Muster ausrichten. Der Vorteil dieser Struktur liegt in einer modularen Erweiterbarkeit und Veränderbarkeit der Komponentenzusammensetzung. Besonders zu betrachten ist die entkoppelte View Komponente. Eine der Hauptanforderungen der Software liegt in der Veränderbarkeit der Darstellungsformen. Dies wird nur durch eine von der Modell- und Datenebene entkoppelten Ansichtsebene ohne größere Probleme ermöglicht.

4.3 Single-Page Applikation

So wie der Referenzarchitektur zu entnehmen ist, wird die Anwendung als Webanwendung implementiert, bei welcher das MVC Muster auch als Standard etabliert ist. Betrachtet man die Anforderungen an eine Architektur, respektive hier die Qualitätsanforderungen an Software, wird die Webanwendung als Single-Page-Applikation implementiert. Diese birgt verschiedene Vorteile, welche direkten Einfluss auf die Qualität haben. Zum einen wird eine Plattformunabhängigkeit garantiert. Dies generiert die Freiheit, die Anwendung in verschiedene Systeme zu integrieren oder auch für verschiedene Endgeräte nutzbar zu machen.

Zum anderen verbessert die Nutzung einer SPA die Qualitätsanforderung der Performance nach ISO 25010 gegenüber einer klassischen Webanwendung. Das Nachladen von Daten entfällt in der klassischen SPA. Sie besteht nur aus clientseitiger Technologie, welche aus HTML, JS und CSS besteht. Somit ist die Applikation auch offlinefähig. Darin liegt ein weiterer architektonischer Vorteil. Die benötigten Daten werden über Services abgerufen, was eine sehr hohe Flexibilität darstellt, da diese Services über Schnittstellen austauschbar werden. [Steyer & Softic 2015, 32f]

4.4 Komponenten

Im Folgenden werden die einzelnen benötigten Komponenten der zu entwickelnden Visualisierungsanwendung vorgestellt.

Die *View* Komponente bzw. die Visualisierung gliedert sich im Folgenden in die Teilaspekte des UI und der Visualisierung der Auslastungsdaten.

Das *Userinterface* dient der Interaktion des Users mit dem System. Darüber soll es dem User ermöglicht werden, Anforderungen an die Visualisierung an das System zu übermitteln

Die *Visualisierung der Auslastungsdaten* verlangt die Forderung einer dreidimensionalen Darstellung. Dort werden die vom System zur Verfügung gestellten Daten entsprechend der Anforderung des Users visualisiert.

Der *Controller* stellt dem View die zu visualisierenden Daten zur Verfügung. Auch verarbeitet er in erster Instanz die eingestellten Konfigurationen des Users.

Der *Model* Layer stellt verschiedene Schnittstellen zur Verfügung, über welche die Daten entsprechend den Anforderungen aufbereitet werden. Es werden folgend die drei Hauptkomponenten des Datenparsers, des Dataservice und Data Customize vorgestellt.

Der *Dataservice* dient dem System als Schnittstelle zu verschiedenen Datenquellen, entsprechend der Anforderung nach einer frei wählbaren Datenquelle. Er dient dazu, die Logik der Datenquelle von der Visualisierung fachlich zu trennen, sodass eine Modularität entsprechend Iso 25010 gewährleistet wird.

Der *Datenparser* hat die Aufgabe, die im Dataservice empfangenen Daten in ein für das System lesbares Format zu bringen. Das heißt, es wird hier eine einheitliche Datenstruktur aufgebaut.

Innerhalb der *Data Customize* Komponente werden der aufgebauten Datenstruktur, entsprechend der Anforderungen des Nutzers, Teilmengen entnommen und an das View übergeben.

4.5 Grobstruktur einer Softwarearchitektur

Der analysierten Referenzarchitektur, den identifizierten Komponenten sowie den ermittelten Anforderungen an das System folgend wird hier ein architektonischer Grobentwurf entwickelt. Der strukturelle Aufbau der entwickelten Softwarearchitektur wird in einer schematischen fachlichen Architekturdarstellung vorgestellt.

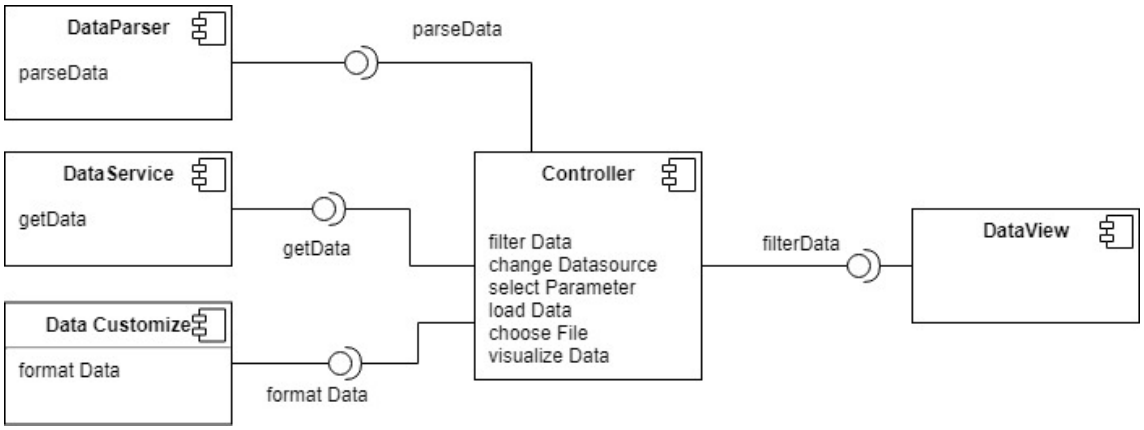

Abbildung 3: Architektur Grobentwurf [eigene Darstellung]

Das zu implementierende System basiert hier auf dem Model View Controller Muster und wird als Single Page Webapplikation umgesetzt.

Der Aufbau der Anwendung wird in drei Layer unterteilt. Die View Komponente ist für die Darstellung der Daten entscheidend. In Abbildung 3 ist dies durch die Komponente View beschrieben. Die Entkopplung der View Komponente ermöglicht es, die Anforderung nach verschiedenen Darstellungsformen implementieren zu können. Der Controller stellt die Schnittstellen bereit, über welche ein Großteil der Anforderungen durch den User getriggert werden können. Diese sind zum Beispiel wie in Kapitel 2.1 beschrieben „filter Data“, change Datasource, select Parameter usw. Innerhalb des Modellayers werden Komponenten an den Controller angebunden, welche für verschiedene Operationen zuständig sind. Diese lassen sich wie in der Abbildung zu sehen in drei Hauptkomponenten differenzieren. Über eine Komponente, welche dafür verantwortlich ist, Daten einer Datenquelle abzurufen und welche visualisiert werden

sollen, wird garantiert, dass es möglich ist, diese durch eine beliebige andere Datenquelle zu ersetzen. So muss diese lediglich innerhalb dieser Komponente neu definiert werden.

Die Komponente „DataParser“ führt die Modularität der Datenquelle fort, indem die Datenquelle fachlich von der Logik der Datenstruktur getrennt wird.

Innerhalb der Komponente „Data Customize“ werden die geparsten Daten entsprechend der Eingaben durch den Nutzer für die Präsentation innerhalb der Viewkomponente aufbereitet, in Abbildung 3 beispielhaft durch die Schnittstelle „format Data“ definiert.

5 Entwurf und Implementierung

Dieser Abschnitt beschäftigt sich mit dem Entwurf und der Implementierung des Systems. Die zentralen Aspekte richten sich nach den Fragen womit die Anforderungen umgesetzt werden, wie und warum. Deshalb wird nachfolgend auf den verwendeten Technologie Stack eingegangen. Das heißt zum Beispiel, welche Programmiersprachen, Frameworks oder anderen technischen Hilfsmittel werden verwendet, um das System zu implementieren. Auch wird diskutiert, warum sich für beziehungsweise gegen die jeweilige Technologie entschieden wurde. Anschließend wird die umgesetzte Softwarearchitektur beschrieben und diskutiert. Es werden Designentscheidungen erläutert, sowie Gesamtzusammenhänge der Systemkomponenten beschrieben und in Bezug auf die Umsetzung der Anforderungen an das System diskutiert. In diesem Kapitel wird der erste im Zuge dieser Arbeit entwickelte Prototyp vorgestellt. Ausgehend von diesem Prototyp werden in Kapitel 6 weitere Architekturen vorgestellt.

In der folgenden Tabelle 2 sind die verschiedenen entwickelten Ansätze zu erkennen. Sie werden unterschieden entlang des Model View Controller Patterns, erweitert um den zur Visualisierung verwendeten Renderer beziehungsweise die Visualisierungsbibliothek. Ein weiterer Vergleichsparameter ist der Support von Web.gl, welcher sich auf den verwendeten Renderer bezieht. Diese werden anschließend vorgestellt und diskutiert, insbesondere deren Vor- und Nachteile.

Tabelle 2: Architekturstacks [Eigene Darstellung]

#	View	Visualizegrammar	Renderer	Web.gl support
1	Angular Marterial	Custom	D3_3d	Nein
2	Angular Marterial	Custom	Three.js	Ja
3	Kibana	Kibana Vega Plugin	D3.js	Nein
4	Angular Marterial	Vega	Sanddance, Deck.gl	Ja
5	React	Custom	Three.js	Ja

5.1 Erster Prototyp

Hier wurde ein erster Prototyp implementiert, welcher die grundlegenden Funktionen des Systems darstellen soll. Die entwickelte Architektur orientiert sich an der in Kapitel 4.5 entwickelten Grobstruktur des Systems. Da die Architekturentwicklung in einem iterativen Prozess erfolgt, ist die entwickelte Architektur des Prototyps nicht final. Jedoch wirkt sie richtungsweisend für die nachfolgenden Designentscheidungen.

Der Prototyp besteht aus einem Technologiestack von Angular 6, Typescript, Material, Node.js, D3_3d.js und Solr.

Im weiteren Verlauf werden die verwendeten Komponenten kurz beschrieben und das Zusammenwirken dieser aufgezeigt.

5.2 Verwendete Technologien

Innerhalb dieses Abschnittes werden die zur Umsetzung eines ersten Prototyps, verwendeten Technologien und Frameworks vorgestellt. Sie werden hinsichtlich ihrer Eigenschaften gegenüber der Erreichung der Qualität von Software untersucht und es wird begründet, weshalb sich für die jeweilige Technologie entschieden wurde.

Angular ist ein von Google entwickeltes, auf Typescript basierendes Framework. Mit 966 Contributern und 14800 Commits ist es eines der beliebtesten Open Source Frameworks

zur Erstellung von Client Applikationen. Angular läuft unter der MIT Lizenz [Angular Github].

Eine Angular Anwendung besteht aus verschiedenen Components, welche verschiedene Views darstellen. Diese Views beinhalten Templates zur Darstellung aber auch Programm Logik, welche die Darstellung verändern kann. Components greifen auf Services zurück, welche die einzelnen Components erweitern, aber nicht direkt Einfluss auf die Darstellung hat. Die Templates werden in HTML geschrieben, welche die Darstellung eines Components beschreibt. Die Logik selbst wird in Typescript geschrieben [Angular Architecture overview].

Typescript ist eine von Microsoft entwickelte Sprache, welche auf Javascipt aufsetzt. Es wird zur Laufzeit in Javascript kompiliert. Dies bringt den Vorteil, dass mit Typescript verschiedene Elemente zu Javascript hinzugefügt werden, welche das Entwickeln robuster gegenüber Fehlern machen als mit Javascript. So verfügt Typescript über eine statische Typisierung, welche Typfehler zur Laufzeit verhindern kann.

Material ist eine von Google entwickelte Designbibliothek, welche direkt in Angular eingebunden ist. So stellt Material fertig gestylte Komponenten Angular zur Verfügung, sodass dies bei der Implementierung einen erheblichen Vorteil bietet. Es können die bereits gestylten voll funktionsfähigen Komponenten in ein Angular Projekt integriert werden.

Bei *Node.js* handelt es sich um eine JavaScript Laufzeitumgebung, welche dazu entwickelt wurde, skalierbare Netzwerkanwendungen zu entwickeln [About | Node.js]. Außerdem ist Node.js eine der Standards und wird von über 2500 Entwicklern entwickelt [Node.js Github]. Desweiteren liefert Node.js den Node Package Manager (NPM) mit welchem sich Javascript Bibliotheken in das Projekt integriere lassen. Der Package Manager von Node.js ist mit über 1.000.000 Javascript Bibliotheken eines der größten Package-Management Systeme überhaupt [Modulecount].

D3.js ist eine Javascript Bibliothek zur Darstellung von dokumentenbasierten Daten mithilfe von HTML, SVG und CSS [Bostock 2019]. D3.js selbst ist nicht in der Lage Daten dreidimensional darzustellen. Daher wird zur Visualisierung der Stadtmetapher

Visualisierung das Plugin *D3_3d* verwendet, welches die Bibliothek dahingehend erweitert [Niekes/d3-3d]

Solr ist ein Such Server mit einer REST-Api. Es bietet Features wie Volltextsuche und unterstützt Standardformate wie XML, JSON und http. Außerdem unterstützt Solr eine nahezu Echtzeit-Indexierung, welche dafür sorgt, dass die Daten sich sehr schnell aktualisieren können und das Abrufen dieser unter dem Gesichtspunkt der Performance einen Vorteil gegenüber einer klassischen Datenbank hat. Des Weiteren unterstützt Solr die sogenannte JSON-Facetten, welche dem User ermöglichen, Daten über die REST Schnittstelle in einer gewünschten Form zu erhalten [Apache Solr - Features].

5.2.1 Aufbau des Prototyps

Die Architektur des Prototyps zielt auf eine möglichst hohe Wiederverwendbarkeit der Komponenten bei der weiteren Entwicklungsarbeit. Das heißt, sie soll so modular wie möglich aufgebaut sein. Der architektonische Aufbau des Prototyps folgt der in Kapitel 4.5 entwickelten Struktur und wird im Folgenden anhand von Architekturbildern und schematischen Abbildungen aufgezeigt und diskutiert. Insbesondere wird dabei auf die Erfüllung der an das System gestellten Anforderungen, welche in Kapitel 2 beschrieben wurden, eingegangen.

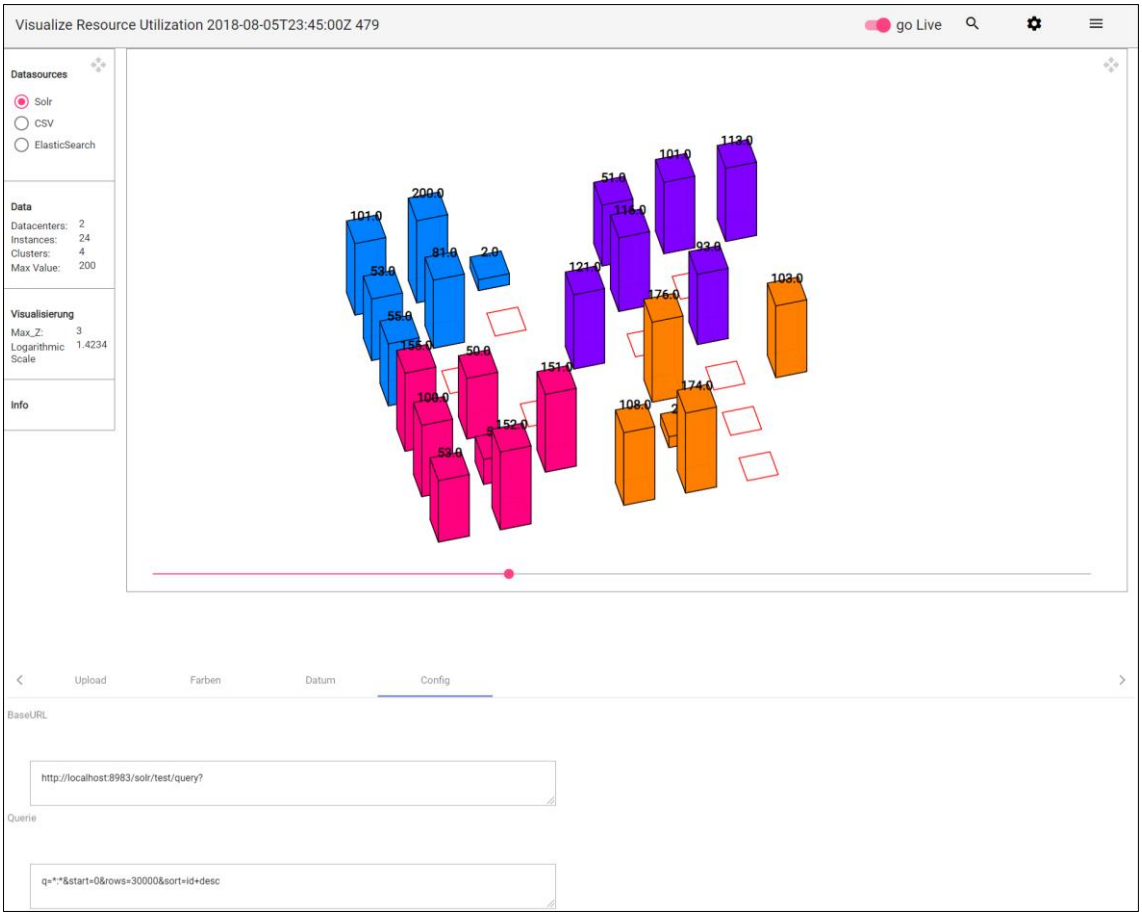

Abbildung 4: UI und Visualisierung des Prototyps [eigene Darstellung]

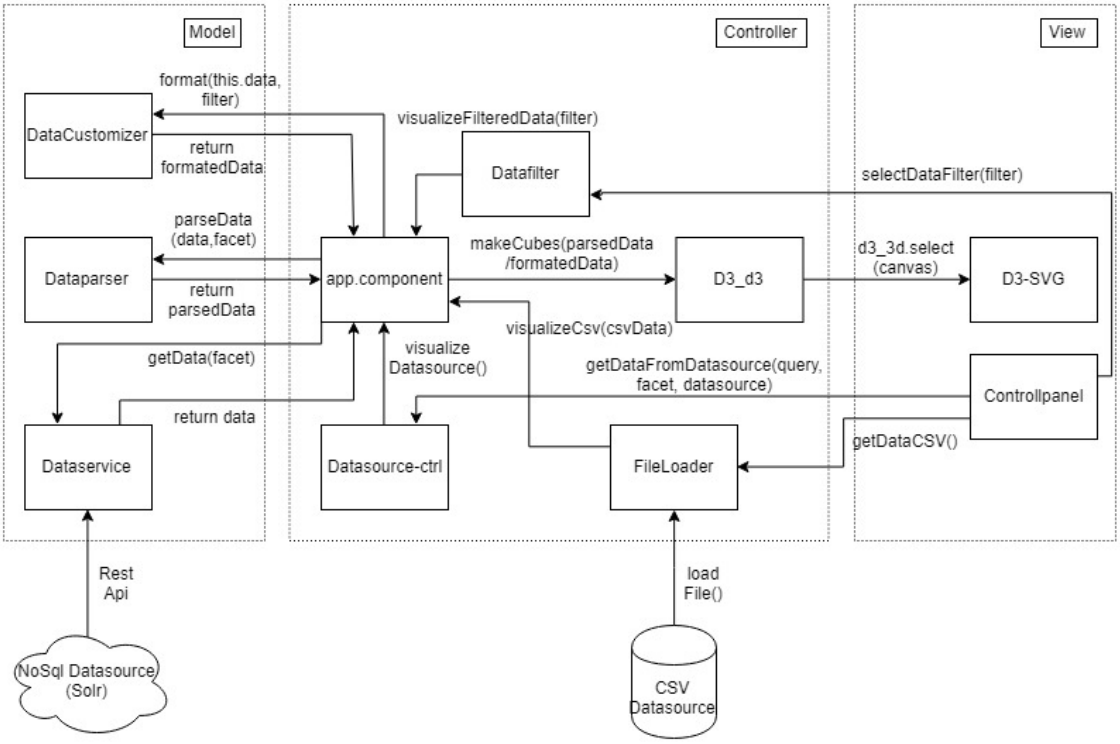

Abbildung 5: Architektur des Prototyps [eigene Darstellung]

Die Abbildung 5 zeigt die entwickelte Architektur des Prototyps auf. Diese orientiert sich an der in Kapitel 4.5 entwickelten Grobstruktur des Systems, welche sich nach dem MVC Muster richtet. So wird die Anwendung in die drei Layer Model, View und Controller unterteilt. die zentrale Komponente innerhalb des Controllers ist die Angular Hauptkomponente die app.component.ts. Diese ist die standardmäßige root Komponente einer Angular Anwendung. Sie ist im Aufbau der Architektur das Zentrale Bindeglied zwischen den weiteren Komponenten des Systems. Die Daten werden über zwei mögliche Wege an das System gebunden. Der erste ist das Laden einer CSV File, welches über das UI, in dem Controllpanel gestartet werden kann. Dort kann vom User eine lokale Datei ausgewählt werden, welche vom FileLoader geladen wird. Diese wird dann über die app.component an das Modellayer weitergereicht, wo sie mithilfe des Dataparser in ein einheitliches Format gebracht wird. Diese Daten werden zurückgegeben und mithilfe des Renderers D3_3d mittels select innerhalb eines Elements des DOM visualisiert.

Ähnlich verhält es sich bei dem Aufruf getDataFromDatasource. Dort wird noch vom User die entsprechende Datenquelle ausgewählt und die Facettierung eingestellt, mit

welcher die Daten der Datenquelle entnommen werden sollen. Diese Informationen können im Controllpanel eingestellt werden und werden dann vom Datasource-ctrl an die app.component.ts übergeben, welche den Dataservice aufruft, sodass die entsprechenden Daten der ausgewählten Quellen entnommen werden können. Diese werden dann auch mittels des Dataparsers in die gewünschte Form gebracht und mit D3_3d visualisiert. Dies erzeugt ein Canvas, welches mittels D3_3d.select an ein Element des DOM gebunden wird.

Eine weitere Anforderung wird durch Filter Data beschrieben. Über das Controllpanel können verschiedene Metriken und Filterungen vorgenommen werden, wie sie schon in Kapitel 2.1.2 beschrieben wurden. Dazu gehören unter anderem change Visualization setting, filter data oder auch change point in time. Diese Filterinformationen werden über selectFilterData an die app.component.ts übergeben und die vorhandenen Daten werden entsprechend der Filterung über die Komponente Data Customizer an die Anforderungen bzw. Filter angepasst, zurückgegeben und wieder mittels D3_3d visualisiert. Auch hier bindet die D3_3d das dabei entstehende Canvas an ein DOM Element.

Die Aufteilung der Komponenten in Model, View und Controller hat die in Kapitel 4.2 diskutierte Vorteil gebracht, nach welchem insbesondere die Entkopplung der Darstellung von den Zugriffen und der fachlichen Logik eine einfache Erweiterbarkeit der Anwendung ermöglicht. So muss beispielsweise in der vorgestellten Architektur, um die Anbindung einer weiteren Datenquelle zu ermöglichen, lediglich die Dataservice Komponente um eine Methode erweitert werden, da durch die Modularität die Herkunft der Daten für die weitere Verarbeitung (Dataparser, DataCustomizer) aber auch für die Darstellung (View D3_3d-SVG) keine Rolle spielt.

5.2.2 Kritik des Prototyps

Der umgesetzte Prototyp weist zwei größere Schwachstellen auf. Zum einen wurde bei einem Lasttest mit mehr als 200 zu rendernden Balken die Performance getestet. Das führte dazu, dass das Rendern der Balken selbst mit hohem Zeitaufwand verbunden war, aber auch das Verändern des Blickwinkels auf den dreidimensionalen Raum mittels draggen nur mit starker Verzögerung möglich war. Dies stellt eine Nichterfüllung einer

der Hauptanforderungen nach ISO 25010 (Software-Qualitätseigenschaften) Performance Efficiency dar. Des Weiteren unterliegt die Umsetzung des Userinterfaces und die Anwendung verschiedener Filterungen auf die Daten in der Implementierung einem relativ hohen Zeitaufwand. Hierbei wäre es nützlich, bereits etablierte Lösungen nutzbar zu machen, sodass eine Eigenentwicklung nicht mehr zwingend notwendig für die Nutzung ist. Ein weiterer Grund, der es notwendig macht, weitere Möglichkeiten der Umsetzung zu diskutieren, liegt in der Modularität des Systems. So sind die einzelnen Komponenten, aufgrund der Architektur von Angular (vgl. Kapitel. 5.1.1), innerhalb der Anwendung selbst sehr modular aufgebaut, und somit auch austauschbar nach ISO Norm 25010 Maintainabillity. Jedoch ist die gesamte Anwendung, aufgrund der verwendeten Technologien, nicht in die Anwendungslandschaften von Grafana oder auch Kibana effizient integrierbar. Es wird nur eine Stand Alone Deployment Methode zugelassen. Zwar wäre theoretisch die Integrierung als Plugin zum Beispiel unter der Verwendung von Kibana möglich, würde jedoch zu dem gewünschten Mehrwert der Verwendung des Userinterfaces von Kibana führen.

6 Weiterentwicklung der Architektur

Aufgrund der in Kapitel 5.2.2 diskutierten Nachteile des Prototyps in dem Bezug auf die Performance, sowie auch auf die Modularität werden nachfolgend weitere mögliche Architekturen sowie Designentscheidungen aufgezeigt und diskutiert.

Diese Designentscheidungen wurden iterativ getroffen und bauen bezüglich ihrer Entscheidungsgrundlage aufeinander auf. Der Ablauf der Beschreibung folgt dem Prozessablauf der getroffenen Entscheidungen.

6.1 Angular und Three.js

Aufgrund der festgestellten Performanceprobleme, welche bei der Durchführung eines Lasttests unter Benutzung des Prototyps erkannt wurden, wurde die bestehende Architektur hinsichtlich der Visualisierungskomponente untersucht. Bei der in dem Prototypen verwendeten Bibliothek handelt es sich um die D3_3d. Es wurden weitere mögliche Visualisierungsbibliotheken gesucht. Diese sollten eine höhere Performance bieten können. Aufgrund dieser Anforderung, bietet sich eine Visualisierungstechnologie an, welche das Interface von Web.GL nutzt.

WebGL ist eine von der Mozilla.org entwickelten Schnittstelle, mithilfe derer 3D Grafiken hardwarebeschleunigt im Client visualisiert werden können [WebGL: 2D and 3D graphics for the web]. Durch die Verwendung einer GPU beim Rendern, kann die Performance extrem gesteigert werden. Eine Bibliothek, welche Web.Gl verwendet, um Daten zu rendern, ist Three.js. Es unterliegt der MIT Lizenz und der Code steht somit zur freien Verfügung. Mit 1115 Contributern auf Github ist es ein breit unterstütztes Projekt [three.js Github].

Eine Three.js Visualisierung besteht aus drei Hauptkomponenten. Zur Visualisierung wird eine „Camera“ initialisiert, welche einen bestimmten Blickwinkel zugeteilt bekommt, sowie entsprechende Positionen auf den Achsen. Daneben wird eine Scene benötigt. Die Scene stellt einen Container für die eigentlich zu rendernden Daten dar. Wie beispielsweise die Geometry. Diese Elemente werden einer Scene wird dann hinzugefügt und die Szene wird gemeinsam mit der Camera mithilfe des Three.js Renderer gerendert.

```
var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera( 75, window.innerWidth /
window.innerHeight, 0.1, 1000 );

var renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild( renderer.domElement );
```

Abbildung 6: three.js Codebeispiel [Quelle: Creating a scene – three.js docs]

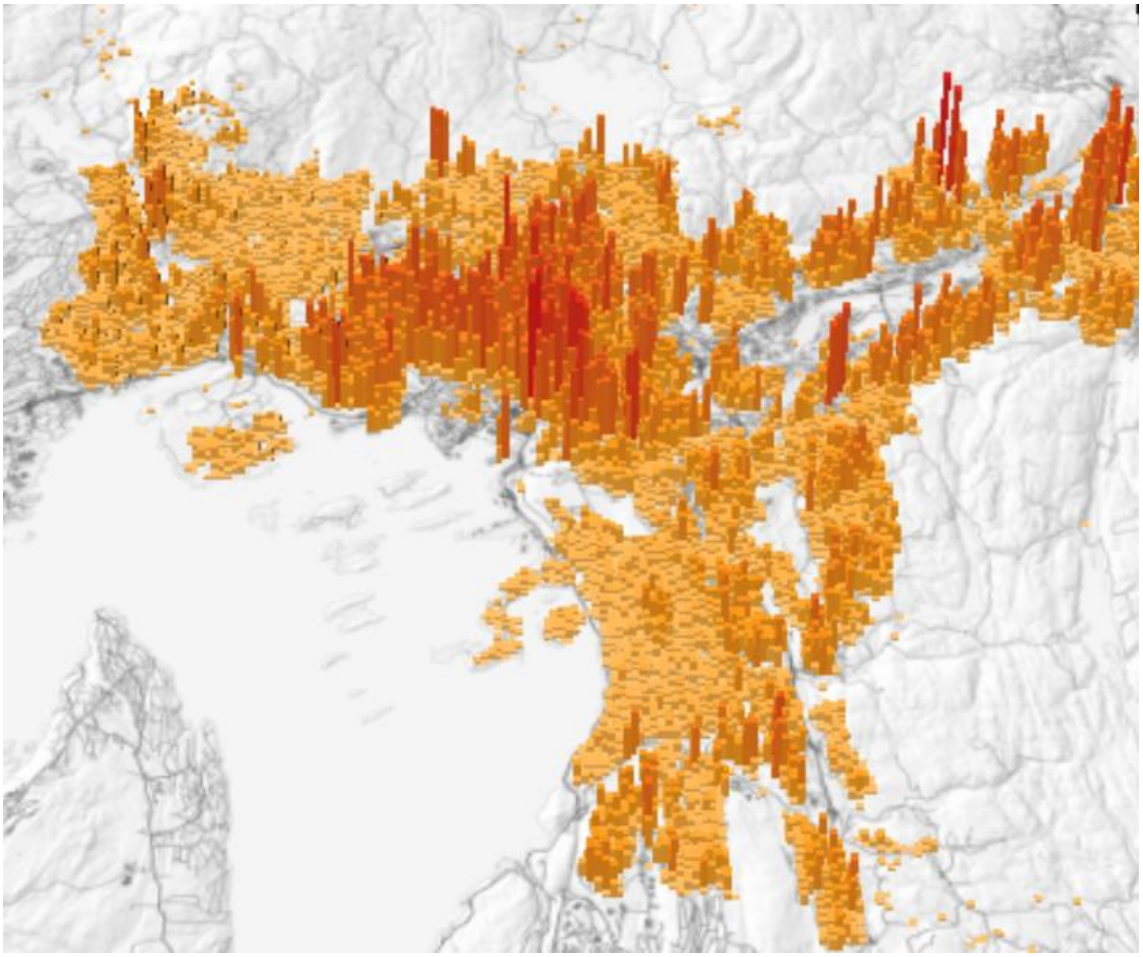

Abbildung 7: Github veiledning09 [Quelle: Bjorn Sandvik 2015]

Der Abbildung 7 lässt sich entnehmen, dass es ohne weiteres möglich ist, unter der Benutzung von Three.js, eine hohe Anzahl an Instanzen zu rendern. Auch gleicht die

gerenderte Form der im Beispiel verwendeten Instanzen, somit denen der gewünschten Darstellung von dreidimensionalen Würfeln.

Three.js lässt sich in die entwickelte Architektur des bestehenden Prototyps integrieren und stellt einen Ersatz für D3_3d dar, der bezüglich der gerenderten Datenmenge eine höhere Performance liefert.

6.1.1 Aufbau der Architektur

Der architektonische Aufbau der Anwendung gleicht dem, der im Prototpyen entwickelten Architektur (vgl. Kapitel 5.2.1). Diese Architektur wird ausschließlich hinsichtlich einer verwendeten Komponente überarbeitet, die des verwendeten Renderers D3_3d durch Three.js. Die genannten modularen Vorteile des MVC gelten auch für diese Architektur. Three.js wird auch innerhalb des DOM in eine SVG oder Canvas gerendert.

6.1.2 Bewertung

Diese Implementierung sorgt für eine extrem skalierbare Lösung der eigentlichen Datenvisualisierung, da unter Einbezug eines Grafikprozessors deutlich mehr Daten gerendert werden können als ohne, wie in Abbildung 7 zu erkennen ist. Jedoch bleibt die Anwendung unter Beibehaltung der restlichen Technologie starr gegenüber einer Integration der Visualisierung in ein anderes System und somit wird auch die Verwendung anderer Userinterfaces und Steuerungselemente verhindert.

6.2 Kibana und Vega

Um das Problem der fehlenden Integrationsmöglichkeit zu lösen, kann die Anwendung auch mithilfe von Kibana umgesetzt werden. Kibana ist hinsichtlich der Integration einer Datenquelle auf ElasticSearch eingeschränkt. Allerdings beinhaltet Kibana auch eine sehr große Anzahl an Steuerungselementen und Filterfunktionen, welche, würden sie nutzbar gemacht werden, einen hohen Mehrwert für den Funktionsumfang der Anwendung beinhalten

Es wurde eine Möglichkeit gesucht, Kibana verwenden zu können und trotzdem die Integration verschiedener Datenquellen beibehalten zu können.

Seit der Version 6.2 hat Kibana Vega integriert [Vega Graphs | Kibana User Guide [6.2] | Elastic]. Vega ist eine Visualisierungsgrammatik, mithilfe welcher es ermöglicht wird, in einem JSON Format die Anordnung und das Verhalten einer Visualisierung innerhalb eines Canvas oder SVG zu beschreiben.

Vega selbst befindet sich in der Version 5.4.0 und läuft unter der BSD 3-Clause Lizenz, welche eine kommerzielle Nutzung ermöglicht [Jeffrey Heer 2018]. Außerdem ist es ein Projekt mit steigender Nachfrage, wie schon an der Integration durch Kibana selbst zu erkennen ist. Die Nutzung von Vega innerhalb Kibana ermöglicht es, Kibana für weitere Datenquellen als ElasticSearch zu öffnen. Vega rendert Daten aus verschiedenen Formaten. Unterstützt werden unter anderem JSON, CSV, TSV und DSV. Diese Formate werden von verschiedenen Datenquellen unterstützt. Insbesondere das JSON Format, da es von vielen NOSQL Datenbanken unterstützt wird. NoSql Datenbanken können Daten auf verschiedene Arten persistieren hinsichtlich ihrer Struktur. Jedoch sind die am häufigsten gebrauchten Arten, das Key-Value speichern, dokumentenbasiertes speichern sowie Extensible Records. Bei der Key-Value Methode, wird zu jedem indexierten Key ein Value gespeichert. Ähnlich verhält es sich bei der dokumentenbasierten Speicherung, bei welcher die Values aber wiederum aus weiteren Key-Value Paaren bestehen können [Rick Cattell 2010, S. 3]. Durch diese Datenstruktur lassen sich die in den Datenquellen gespeicherten Daten leicht in das von Vega verlangte JSON Format transformieren. JSON selbst ist eine aus Javascript entnommene Sprache zur Darstellung von Datenstrukturen. Die Struktur einer JSON gleicht der einer dokumentenbasierten Speicherung. Jedes Value kann ein bis beliebig viele mögliche Values haben [Thomas Brey 2019, 1ff].

Das Datendesign von Vega funktioniert über zwei Keys. Es ist das „$schema“ über welches definiert wird, wie die Daten visualisiert werden sollen. Die Daten selbst werden über den Key „data“ definiert [Vega Parser API]. Dort können Listen mit verschiedenen Daten definiert werden. Diese modulare Definition der Daten ermöglicht es der Vega Grammer Daten aus einer beliebigen Datenquelle übergeben zu können.

```
{
  "$schema": "https://vega.github.io/schema/vega/v5.json",
  "width": 400,
  "height": 200,
  "padding": 5,

  "data": [
    {
      "name": "table",
      "values": [
        {"category": "A", "amount": 28},
        {"category": "B", "amount": 55},
        {"category": "C", "amount": 43},
        {"category": "D", "amount": 91},
        {"category": "E", "amount": 81},
        {"category": "F", "amount": 53},
        {"category": "G", "amount": 19},
        {"category": "H", "amount": 87}
      ]
    }
  ],
```

Abbildung 8: Vega Grammar Beispiel [Quelle: Vega Bar Chart Example]

6.2.1 Aufbau der Architektur

Der Aufbau der Architektur wird in Kibana eingebettet. Dies ist erforderlich, da das Ziel angestrebt wird, Funktionen von Kibana für die Anwendung nutzbar zu machen. Deshalb wird eine Komponente entwickelt, welche es ermöglichen soll, der von Kibana zur Verfügung gestellten Vega Instanz, die notwendigen Daten in einer für Vega verständlichen Form zu übergeben. Außerdem muss für die Verwendung von Vega ein Schema entwickelt werden, welches die Visualisierungsstruktur definiert. Diese zwei Komponenten lassen sich aus der Anforderung von Vega ableiten, welche benötigt werden, um Daten zu visualisieren. Der Aufbau der Integration einer Kibana Visualisierung durch Vega zeigt die Darstellung in Abbildung 8.

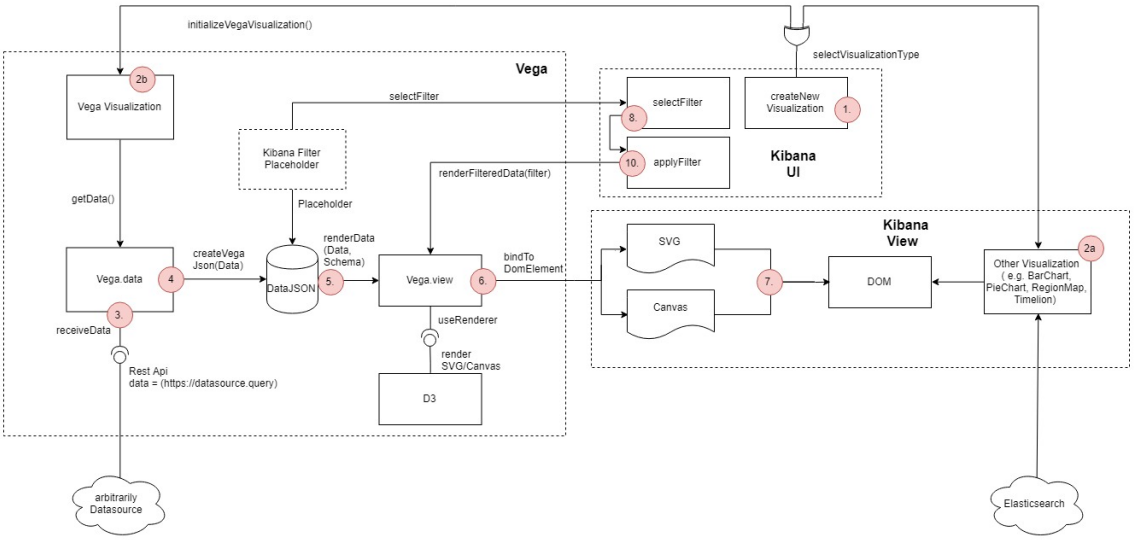

Abbildung 9: Vega Kibana Integration [eigene Darstellung]

Bei 1. Wird in createNewVisualization über das Userinterface mittels selectVisualizationType von Kibana eine neue Visualisierung getriggert. Dort kann man zwischen verschiedenen Visualisierungsformen wählen. 2a Other Visualization erstellt eine Standart Kibana Visualisierung wie zum Beispiel einem Bar Chart, Pie Chart oder einer Timelion Visualisierung, welche der Visualisierung von Zeitreihen dient. Diese Darstellung ist jedoch in zwei Dimensionen gehalten. Diese Visualisierungen nutzen die Standarddatenquelle von Kibana, Elasticsearch und sind auch ausschließlich an diese gebunden.

2b Darüber hinaus kann man eine in Kibana integrierte Vega Visualisierung initialisieren. Über diese Visualisierung wird es ermöglicht, andere Datenquellen anzuzapfen als Elasticsearch. Dies geschieht 3. über die Definition der Variable „data“, welcher eine URL übergeben werden kann, welche JSON daten zurückliefern muss. In der Konsequenz können alle Datenquellen angebunden werden, welche über eine Restschnittstelle verfügen.

Die empfangenen Daten werden durch den Aufruf von createVegaJson in das Vegagrammar-Format übertragen; zur Ermöglichung der Verwendung von Filtern, kann der Entwickler diese, durch Verwendung von Placeholdern, welche von Kibana zur Verfügung gestellt werden, die jeweiligen Keys der zu filternden Daten erweitern.

Die daraus resultierenden JSON Daten werden 5. über den Aufruf renderData() mit einem definierten Vega Schema, welches definiert, in welcher Form die Daten visualisiert werden sollen, Über Vega.view durch den von Vega verwendeten Renderer D3.js gerendert. Dieser liefert die Visualisierung in ein SVG oder Canvas Objekt. Dieses Objekt wird durch den Aufruf 6. bindToDomElement an ein Element des 7. DOM's, wo die Visualisierung sichtbar wird.

Über das von Kibana zur Verfügung gestellte Userinterface können 8. die in die Daten eingefügten Placeholder als Filter ausgewählt werden und 9. auf die Daten angewandt werden. Daraufhin werden die gefilterten Daten gerendert und das resultierende SVG beziehungsweise Canvas an das DOM gebunden.

6.2.2 Bewertung

Durch die hohe Verbreitung von Kibana und die vielfältigen Steuerungsmöglichkeiten ist die Möglichkeit der Integration in Kibana ein wünschenswertes Ziel für die Architektur der Anwendung. Unter der Verwendung von Kibana mit Vegaintegration stehen die vielen Designmöglichkeiten des Kibana UI zur Verfügung. Auch wird eine Modularität (Wiederverwendbarkeit) im Hinblick auf die Integrationsmöglichkeit verschiedener Datenquellen zur Anwendung durch Vega garantiert, welche der Qualitätsanforderung Modularität dient. Dies jedoch nur teilweise, denn es ist nicht möglich, eine Stand-alone-Variante der Anwendung mit dem vorgeschlagenen Stack zu deployen. Es entsteht eine Abhängigkeit von Kibana. Auch von der verwendeten Vega Version in Kibana. So weist Kibana darauf hin, dass die Integration von Vega in Kibana, experimentell ist, und dass es möglicherweise in zukünftigen Versionen wieder entfernt wird [Vega Kibana V 7.2]. Zudem muss an diesem Punkt erwähnt werden, dass Kibana mit Vega 4.3 eine alte Version von Vega implementiert. Die aktuelle Version ist 5.4.1. Hinzu kommt, dass der von der in Kibana integrierten Vega Version verwendete Renderer D3 nicht das Rendern einer dritten Dimension ermöglicht [Vega axis documentation]. Der Renderer unterstützt keine Z-Achse, weshalb es mit dieser Lösung nicht möglich ist, die an das System gestellten Anforderungen einer dreidimensionalen Darstellung der Auslastungsdaten umzusetzen.

6.3 Angular und SandDance

Unter Berücksichtigung der Vorteile und der Simplizität, mit welcher Vega es ermöglicht, eine Visualisierung zu beschreiben, wurde eine Möglichkeit gesucht, unter welcher es Vega ermöglicht wird, die in Kapitel 6.2 beschriebene Probleme zu lösen.

Da der Renderer von Vega selbst keine Möglichkeit bietet, eine Z-Achse darzustellen, ist es für eine weitere Verwendung unter der Prämisse, dass eine dreidimensionale Darstellung der Auslastungszeitreihen umgesetzt werden soll, unumgänglich, dass der native Renderer von Vega durch einen anderen ersetzt werden muss.

Die Problematik des Renderns einer dritten Dimension innerhalb von Vega wurde auch von zwei Entwicklern von Microsoft aufgegriffen [3D data visualization issue]. Sie entwickelten deshalb den JavaScript Visualisierungs-Stack SandDance.

SandDance vereint die Layoutgrammatik von Vega, wie bereits in Kapitel 6.2 diskutiert, mit Deck.gl als ein Web.gl basierenden Renderer [Sanddance].

Deck.gl wird von Uber entwickelt und der Code steht mit der MIT Lizenz zur freien Verfügung [uber/deck.gl]. Es handelt sich dabei um ein auf Web.gl basierenden Framework zur Visualisierung von großen Datenmengen. Diese werden in der Standardversion von Deck.gl besonders auf Karten dargestellt [Large-scale WebGL-powered Data Visualization].

Abbildung 10: Deck.gl Beispiel-Visualisierung [Quelle: Deck.gl-Example]

Wie in Abbildung 10 zu sehen ist, ist Deck.gl in der Lage eine sehr große Menge an Daten zu rendern. Außerdem orientiert sich das Beispiel an dem Design der Visualisierungskomponente des Prototyps, welches auch an der Darstellung einer Stadt mit verschiedenen Blöcken und Höhen der Gebäude zu erkennen ist. Auch ist das Rendern von dreidimensionalen Würfeln problemlos möglich.

Durch die Verwendung des Deck.gl Renderers anstatt des nativen, von Vega gelieferten, wird es ermöglicht, mittels Vega die zu rendernden Punkte auf allen drei Achsen zu definieren und durch die Nutzung des Deck.gl Renderers zu visualisieren.

Mithilfe von Vega, wird eine höhere Modularität garantiert im Bezug auf die Integration der Visualisierung in andere Systeme, wie zum Beispiel Kibana. Wird der Renderer, welcher von Vega verwendet wird und bereits in Kibana integriert ist, um die Funktion für eine dritte Achse erweitert, wäre eine Integration der Visualisierung selbst in Kibana umsetzbar.

Der SandDance Stack wird in eine Angular Anwendung eingebunden. Dies ist möglich über die Einbindung folgender Packages über Node.js.

- vega-lib
- SandDance
- deck.gl
- luma.gl

Die SandDance Anwendung selbst wird darüber initialisiert, dass SandDance eine Vega Instanz, deck.gl und layers, welche Teil von Deck.gl sind und luma.gl übergeben wird. Luma.gl ist der Web.gl basierende Renderer, welcher von Deck.gl verwendet wird. Schließlich wird der SandDance Instanz noch ein Presenter hinzugefügt, welchem ein DOM Element übergeben wird, in welchem die Visualisierung gerendert wird. Der Aufbau solch einer Instanz lässt sich in der folgenden Abbildung 11 ablesen. Innerhalb der Funktion werden dann die zu rendernden Objekte, sowie weitere Metadaten wie zum Beispiel die Kameraperspektive auf die Visualisierung definiert.

```
initCube(cubedata:any): void{

  var cubeTest;
  console.log(cubedata);
  (function (cubeTest) {
    SandDance.use(vega, deck, layers, luma);
      var colors = {
          red: [255, 0, 0],
          green: [0, 255, 0],
          blue: [0, 0, 255],
          gray: [128, 128, 128]
      };
      cubeTest.presenter = new SandDance.VegaDeckGl.Presenter(document.querySelector('#vis'));
      var stage = {
          cubeData: cubedata
```

Abbildung 11: Codebeispiel SandDance [eigene Darstellung]

6.3.1 Aufbau der Architektur

Der Architekturaufbau des beschriebenen Stacks wird in diesem Abschnitt aufgezeigt und beschrieben. Dabei wird das Zusammenspiel der einzelnen Komponenten diskutiert und an einem Beispiel erläutert.

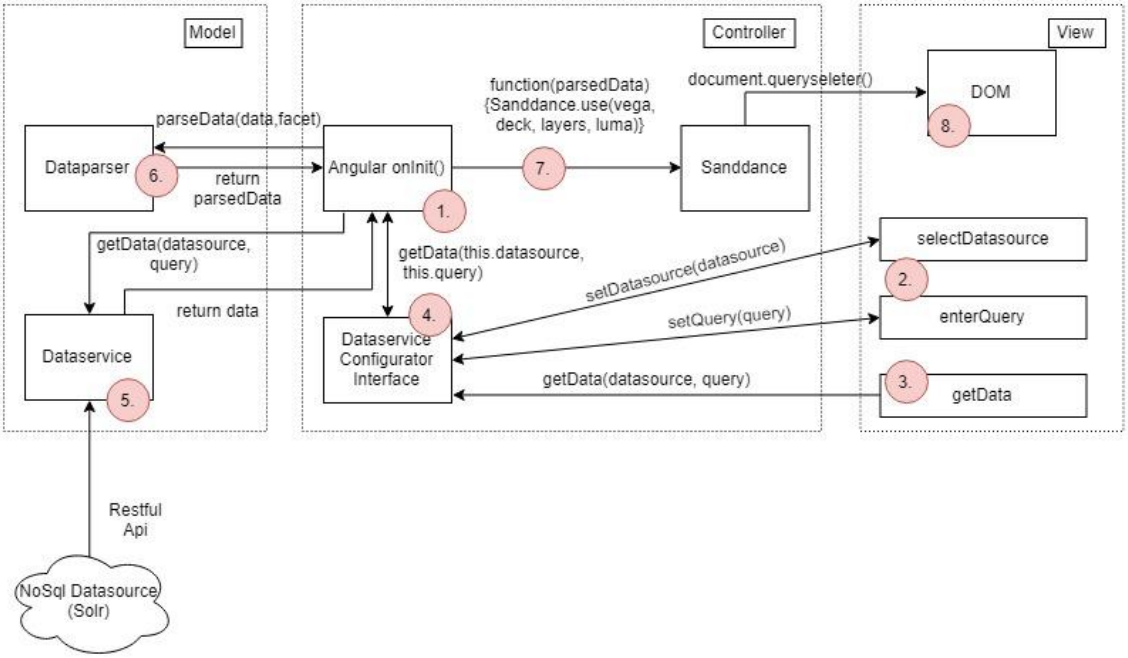

Abbildung 12: SandDance Angular Architektur [eigene Darstellung]

Es entsteht eine Standalone Webanwendung, welche der identifizierten Referenzarchitektur folgend (vgl. Kapitel 4.2) auf dem Pattern des Mode View Controller aufbaut. Dementsprechend sind die einzelnen Komponenten in die drei Layer unterteilt.

Bei 1. Beginnend wird beim Starten der Angularanwendung die implementierte Method onInit() aufgerufen. Über den Constructer der Hauptkomponente, welche bei einer Angularanwendung immer durch die app.component.ts definiert wird, wird der Service Dataservice injected. Dieser ruft bei onInit() über eine Restful-Schnittstelle die zu visualisierenden Daten ab. Dabei wird die bereits diskutierte Facettierung nutzbar gemacht, welche von, in diesem Beispiel Solr, zur Verfügung gestellt wird. Über ein 2. Selectfeld und eine Textbox, welches in der Viewkomponente verfügbar ist, kann der Nutzer die Datenquelle auswählen und die Abfragequery, also das gewünschte Ausgabeformat, welches die ausgewählte Datenquelle wiedergeben soll, definieren. Diese werden über das von Angular bereitgestellte two-way Databinding an im Angular model vorhandene Variablen gebunden. Das Abrufen der Daten wird 3. über einen Button im Userinterface getriggert. Mit der vorher definierten Datenquelle und der Query wird über das Interface 4. Dataservice Configurator der 5. Dataservice getriggert. Dort

werden aus der Datenquelle über eine Restful api mittels get abgerufen die Daten gesammelt und an die app.components.ts zurückgegeben. Diese ruft 6. die Komponente Dataparser auf und übergibt ihr die Daten sowie die JSON-Facette, welche aus der Query abgeleitet werden kann. Anhand dieser werden die Daten in die Struktur, welche die Vegagrammer erwartet, übertragen. Diese aufgebaute Datenstruktur wird zurück an den Controller gegeben. Nun können bestimmte Teile der Daten, je nach Filterung an SandDance übergeben werden. SandDance rendert mittels deck, lumar und layers die im Vega Format vorliegenden Daten 7. und übergibt diese an die View Komponente über die Funktion document.query.select() welcher eine Div Id übergeben wird 8..

6.3.2 Bewertung

Die entwickelte Architektur unter der Verwendung des Stacks Angular – SandDance bietet eine Vielzahl an Integrationsmöglichkeiten. Zum einen können die Komponenten Model sowie Controller in andere bestehende Anwendungen integriert werde, da Vega zum Beispiel von Kibana unterstützt wird, wenn auch momentan noch nicht mit einem Renderer, der fähig ist, eine Z-Achse zu rendern. Außerdem kann die gesamte Anwendung auch als Stand-alone-Variante deployt werden, somit ist sie zwar offen gegenüber Integrationen, aber nicht abhängig davon. Diese Modularität genügt dem Qualitätsanspruch nach Iso 25010 nach Wartbarkeit (vgl. Kapitel 4.1) in einem höheren Maß als die anderen vorgestellten Architekturen. Auch wird darüber hinaus das Qualitätskriterium Portabilität erfüllt, denn die Anwendung lässt sich auf mehrere Arten installieren, wie bereits diskutiert eingebettet in Kibana, aber auch als Stand-alone-Version. Zusätzlich ist, da SandDance von Microsoft entwickelt wird, auch eine Integration in PowerBI denkbar. All diese Möglichkeiten machen die Architektur offen gegenüber Änderungen, wie zum Beispiel das Einstellen einer Unterstützung verschiedener möglicher Zielsysteme (PowerBi, Kibana) Die Nutzung des Web.Gl basierten Renderers, sowie die problemlose Austauschbarkeit der verwendeten Datenquelle garantieren eine hohe Performance und Verlässlichkeit nach Iso 25010. Der Aufbau lässt auch eine problemlose Erweiterung der verwendeten Services zu, so kann das Model Layer auch um beliebige Anzahlen an weiteren Komponenten erweitert werden, sodass einer Erweiterung der Funktionen dann nichts mehr im Weg steht.

SandDance wird von Microsoft entwickelt. Betrachtet man allerding die hohe Anzahl an Projekten, welche von großen Unternehmen wie zum Beispiel Microsoft oder Google umgesetzt werden, aber die Entwicklung durch Neuorientierung nicht fortgesetzt werden, bedeutet dies nicht direkt eine Zukunftssicherheit der Technologie sondern impliziert sogar eine gewisse Unsicherheit gegenüber dieser. Zudem verwendet die aktuelle Version von SandDance eine veraltete Vega Version mit Version 4.4.0 gegenüber der aktuellen Version 5.4.1.

Die Anwendung lässt sich unter der Verwendung von Angular und SandDance, was eine dreidimensionale Darstellung unter der Verwendung von Vega ermöglicht, nicht in Kibana oder Grafana integrieren. Beide sehen für die Entwicklung eines Plugins für ihre Systeme die Verwendung von React.js vor.

6.4 React.js und Three.js

Aus den in Kapitel 6.3.2 genannten Gründen, wurde eine weitere Designänderung bezüglich der verwendeten Technologien entwickelt, welche das Ziel verfolgt, die Anwendung gegenüber einer Integration in Kibana und Grafana zu erlauben , aber auch eine Stand-alone-Variante ermöglichen zu können. Ein weiteres Ziel der Veränderung ist es, die Verwendung von SandDance zu umgehen, da es sich dabei zwar um eine Technologie handelt, welche die diskutierten Probleme der Datenquellenanbindung in Kibana mittels Vega, sowie das damit einhergehende dreidimensionale Darstellungsform Problem von Vega löst (vgl. Kapitel 6.3.2), jedoch aufgrund des noch sehr jungen Alters des Stacks kaum über entsprechende Dokumentation verfügt. Das kann bei der Implementierung zu erheblichem Mehraufwand führen. Auch gibt es eine gewisse Unsicherheit gegenüber der Zukunftssicherheit von SandDance einher. Zwar handelt es sich um ein Projekt, das von Microsoft entwickelt wird, jedoch besteht die Möglichkeit, dass die Entwicklung nicht fortgeführt wird. Aus diesen beiden Gründen wird eine weitere Implementierung designt. Diese besteht aus einer Anwendung, welche mittels React.js gebaut wird und zur Visualisierung auf dem auf Web.Gl basierende Visualisierungsframework Three.js aufbaut.

Es wäre auch eine Implementierung mithilfe des in Kapitel 6.3 vorgestellten Renderers Deck.gl möglich. Stellt man diese beiden Renderer sich jedoch gegenüber, so erkennt man sofort, dass Three.js der Vorrang zu geben ist. Der folgenden Tabelle 3 kann ein Vergleich der, in den verschiedenen Implementierungen benutzten Renderern entnommen werden. Diese Renderer werden verglichen hinsichtlich Ihrer Beliebtheit, welche an den jeweiligen Github Repository gemessen wird, was dann darüber Aufschluss gibt, wie zukunftssicher die verwendete Technologie ist. Auch ist aufgrund des Performanceanspruches der Anwendung eine Web.gl Fähigkeit zu vergleichen [d3.js Github], [deck.gl Github], [three.js Github], [Niekes/d3-3d].

Tabelle 3: Vergleich Renderer (Stand: 19.08.2019) [eigene Darstellung]

Kategorie	D3_3d	Deck.gl	Three.js	D3.js
Github Used by	26	969	22.105	66.793
Github Contributor	1	96	1134	121
Web.gl	Nein	JA	JA	Nein
Lizens	BSD-3	MIT	MIT	BSD-3
Created at	2017	2015	2010	2010
3D capable	JA	JA	JA	Nein

Wie die Tabelle zeigt, ist D3.js eine sehr gefragte Bibliothek zur Visualisierung und wird so von bereits 66.793 weiteren Projekten verwendet. Allerdings unterstützt D3.js keine Hardwarebeschleunigung der Visualisierung, wie sie durch das Interface von Web.gl implementiert wird. Auch ist es unter der Verwendung von D3 nicht möglich, dreidimensional zu visualisieren. D3 kann um diese Funktion durch das Plugin D3_3d erweitert werden. Im Vergleich zu den anderen vorgestellten Renderern verfügt D3_3d über eine geringe Verbreitung und wird auch nur von einem einzelnen Entwickler entwickelt. Ferner verfügt der Renderer nicht über ein Web.gl support. Dies führt zu, wie in Kapitel 5.2.2 beschrieben, Performanceproblemen der Visualisierung bei entsprechender Auslastung. Vergleicht man Deck.gl und Three.js wird auch in Bezug auf die Beliebtheit deutlich, das Three.js eine deutlich größere Verbreitung findet. So wird die Bibliothek in 22.000 Projekten verwendet, wohingegen Deck.gl auf ca. 1000 kommt. Zwar handelt es sich bei Deck.gl um das jüngere Projekt, jedoch geht mit der Verwendung von Three.js anstatt von Deck.gl trotzdem eine größere Zukunftssicherheit einher.

Bei React.js handelt es sich um eine von der Firma Facebook entwickelten, auf JavaScript basierenden Bibliothek zur Entwicklung von User Interfaces. Es wird als Open Source Projekt unter der MIT Lizenz entwickelt und steht zur freien Verfügung.

Die Implementierung der Anwendung mit React.js birgt den Vorteil der Öffnung gegenüber der Integration in Kibana sowie Grafana. Wie bereits in Kapitel 3 diskutiert, handelt es sich bei diesen beiden Systemen um zwei weit verbreitete Lösungen im Bereich der Visualisierung von Daten. Bei einer Integration der Anwendung in diesem System lässt sich die Anwendung in die Systemlandschaft vieler Unternehmen problemlos eingliedern. Auch in dem Bezug auf andere Qualitätsmerkmale nach ISO 25010 hat die Anwendung weitere Vorteile, so erhöht sich beispielsweise die Zukunftssicherheit des Systems, da mit Grafana, Kibana und als Stand-alone-Variante drei verschiedene Möglichkeiten zur Verfügung stehen, die Anwendung zu deployen. Eine Integration in beide Systeme wird durch Kibana und Grafana selbst ermöglicht, welche zur Verfügung stellen, für die jeweilige Anwendungen Plugins zu entwickeln, Kibana und Grafana führen Dokumentationsseiten, wie dies umgesetzt werden kann [Plugin Development | Kibana User Guide [7.3] | Elastic], [Grafana Developer Guide].

Grafana dokumentiert die Pluginentwicklung direkt für React.js, sowie auch für Angular. Diese ist bei Kibana nativ auf eine React.js Anwendung eingestellt. Hier lässt sich erkennen, dass Kibana im Masterbranch ein Skript zur Verfügung stellt, welches es einem ermöglicht, mittels der Eingabe einiger Parameter, ein leeres Plugin für Kibana zu erstellen. Dieses Plugin basiert jedoch auf einer React.js Anwendung [spalger].

Als Äquivalent für die Funktion von SandDance, wurde sich erneut für Three.js entschieden, welches verschiedene Vorteile mit sich bringt, zum Beispiel bezüglich der Performance (vgl. Kapitel 6.1)

6.4.1 Aufbau der Architektur

Der Aufbau der Stand-alone-Variante ist stark an dem des Prototyps (Kapitel5.2.1) orientiert. Es werden die gleichen Komponenten benötigt. Nur wird anstelle von Angular hier React.js verwendet. Dies bringt die Möglichkeit der Integration der Anwendung in Kibana, welche in folgender Darstellung Abbildung 13 vorgestellt.

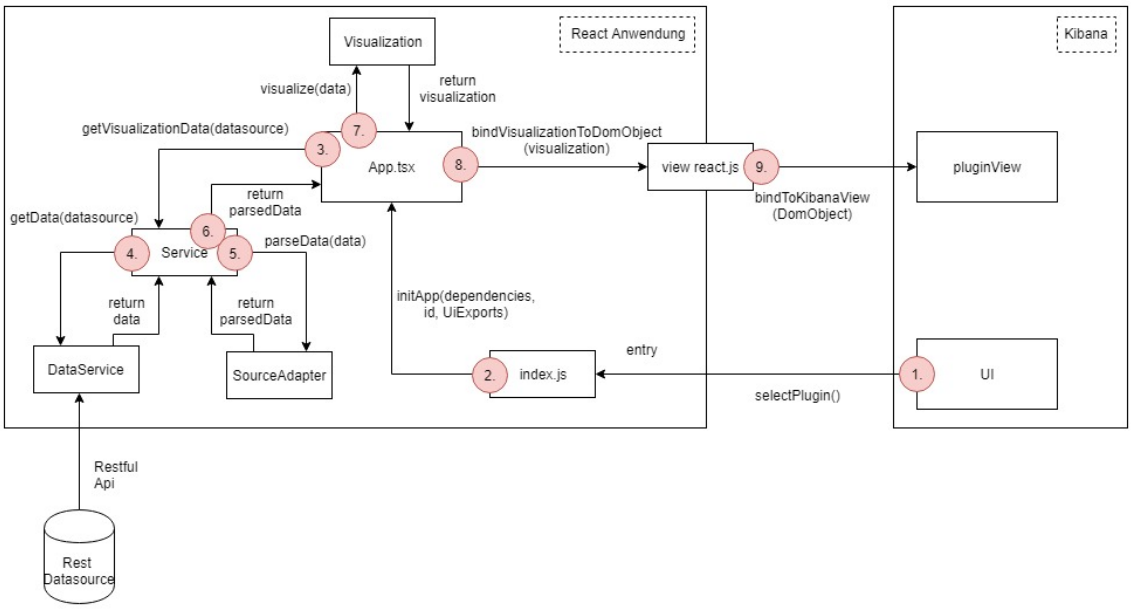

Abbildung 13: Integrationsarchitektur React.js und Three.js Kibanaplugin [eigene Darstellung]

Die Abbildung zeigt den Aufbau der Integration der Anwendung als Plugin für Kibana. Der initiale Eintrittspunkt in die Anwendung wird über die index.js realisiert. 1. Ruft der Benutzer das Plugin in Kibana auf, wird auf diesen Punkt referenziert. Innerhalb der index.js wird die Applikation initialisiert und zum Beispiel Dependencies oder auch UserInterface Exports definiert. 2. Die eigentliche Anwendung wird dann durch den Aufruf von initApp in der App.tsx gestartet. 3. Die App.tsx ruft die zu visualisierenden Daten über einen Service mittels einer 4. DataService Komponente ab. 5. Diese werden dann über den SourceAdapter in ein definiertes Zielformat geparst. 6. Die geparsten Daten werden vom Service an die App.tsx zurückgegeben und in der 7. Visualization Komponente durch den Renderer Three.js gerendert. Diese Visualisierung wird dann durch die App.tsx 8. mit dem Aufruf von bindVisualizationToDomObject an ein DOM Element der View Komponente gebindet. Diese View Komponente der Anwendung wird 9. in die Kibana Oberfläche integriert.

Des Weiteren wird eine Integration in die Visualisierungsanwendung Grafana ermöglicht. Dies kann auch über die Erstellung eines Plugins realisiert werden. Grafana selbst bietet eine ausführliche Dokumentation über die Erstellung von Plugins [Grafana Developer

Guide]. Diese können in Javascript implementiert werden. Wie auch bei Kibana gibt es einen Entrypunkt, über welchen ein Plugin exportiert wird. Dies ist im Fall von Grafana module.js. Grafana stellt verschiedene Events zur Verfügung, mithilfe derer ein Plugin auf Ereignisse in Grafana selbst reagieren kann.

6.5 Validierung der entwickelten Architekturen

In diesem Kapitel wird eine Validierung der entwickelten Architekturen durchgeführt. Diese orientiert sich insbesondere an den Qualitätskriterien, wie in Kapitel 2.2 beschrieben, Performance und Skalierbarkeit. Um die Anwendungen vergleichbar gegenüber der Performance, sowie Skalierbarkeit zu machen, werden im Folgenden für jede der entwickelten Architekturen 2000 Datensätze in Form von Balken gerendert. Dabei wird die Zeit gestoppt, welche benötigt wird, um die Daten zu visualisieren. Auch wird überprüft, wie sich die Rotation der Kamera durch die dreidimensionale Anwendung verhält.

Eine Validierung für die Architektur, welche in Kapitel 6.2 beschrieben wird, kann nicht durchgeführt werden, da das Rendern von dreidimensionalen Balken mit dem vorgeschlagenen Stack nicht möglich ist.

Die gemessenen Zeiten vom Aufruf des Renderns bis zur fertigen Visualisierung, sowie das Verhalten der Kamerarotation kann der folgenden Tabelle entnommen werden. Bei den angegebenen Zeiten handelt es sich jeweils um das gemessene Maximum bei jeweils zehn Versuchen.

Tabelle 4: Architekturen Performancevalidierung [eigene Darstellung]

#	Stack	rendering time in ms	rotation
1	Prototype	637	stark verzögert / kaum möglich
2	Angular und Three.js	12	minimal verzögert
3	Kibana und Vega	n/a	n/a
4	Angular und Sanddance	19	kaum verzögert
5	React und Three.js	9	minimal verzögert

Wie der Tabelle entnommen werden kann wird durch die Verwendung eines Renderers mit Web.gl Unterstützung generell ein enormer Performancezuwachs verzeichnet. So hebt sich die Zeit zum Rendern von 2000 Balken unter der Verwendung von D3_3d, welches nicht auf Web.gl basiert mit 637 Millisekunden enorm von den anderen Architekturen ab. Die schärfste Rotation der Kamera entstand unter der Verwendung von Angular und SandDance. Jedoch lag dort die Zeit des Renderings etwas höher als unter der Verwendung von Three.js, was die höchste Performance und somit auch Skalierbarkeit aufweist. Der gemessene Unterschied zwischen den beiden Architekturen, welche Three.js verwenden könnte an dem kompakteren Aufbau von React.js liegen, fällt jedoch so gering aus, dass er zu vernachlässigen ist. Die Rotation der Kamera war flüssig jedoch minimal verzögert bei beiden Three.js Architekturen.

6.6 Bewertung der entwickelten Architekturen

Die Entwicklung einer Architektur, sowie die Entscheidungsfindung bezüglich des zu verwendenden Technologie-Stacks durchlief mehrere Phasen, bei der verschiedene Ansätze entstanden, wie die Anwendung implementiert werden kann. Diese Designentscheidungen wurden getroffen, um jeweils ein höheres Maß an Qualität der Software garantieren zu können. Hinsichtlich der gesetzten Anforderung wurden verschiedene Wege aufgezeigt, welche die einzelnen teilweise konträren Ziele zu unterschiedlichen Graden erfüllen. Diese verschiedenen Architekturen sollen in Bezug auf die Erfüllung der Anforderungen Performance der Visualisierung, Integrationsmöglichkeit in bestehende Systeme, Zukunftssicherheit und Implementierungsaufwand verglichen werden. Die Bewertungen leiten sich aus den im Kapitel 6 diskutierten Vor- und Nachteilen der einzelnen Komponenten ab. Diese wurden im Bezug zur Erfüllung der Anforderung, insbesondere der der Erfüllung der Qualitätsmerkmal Maintainability, Usability, Performance, Portability und Reliability nach ISO 25010 beschrieben. Der Qualitätsvergleich wird in der folgenden tabellarischen Darstellung gezeigt.

Tabelle 5: Entwickelte Architekturen im Vergleich der Qualitätserfüllung [eigene Darstellung]

Qualität	Prototype	Angular/Three.js	Kibana/Vega	Angular/Sanddance	React.js/Three.js
Maintainability	✓	✓	✓	✓	✓
Usability	x	x	✓	✓	✓
Performance	x	✓	✓	✓	✓
Portability	x	x	x	✓	✓
Reliability	✓	✓	x	x	✓

Wie die Tabelle 5 zeigt, hat die Architektur um den Stack von React.js und Three.js gegenüber den gesetzten Qualitätsanforderungen den höchsten Erfüllungsgrad. Dies liegt im Besonderen an der verwendeten Technologie. Three.js weist zudem einen hohen Beliebtheitsgrad auf, wie in Kapitel 6.4 beschrieben. Auch garantiert die Verwendung von React.js die Möglichkeit, die entwickelte Anwendung in die Visualisierungs- und Analyse Anwendungen Kibana und Grafana zu integrieren. Dies sorgt für eine deutlich höhere Zuverlässigkeit, entsprechende Wartbarkeit, aber auch Benutzbarkeit. Denn diese Anwendungen zeichnen sich durch eine große Beliebtheit aus, was auf eine angemessene Nutzerfreundlichkeit schließen lässt.

7 Proof of Concept

In diesem Kapitel wird ein POC der entwickelten Architektur vorgestellt. Er soll zeigen, dass die entwickelte Architektur aus Kapitel 6 und die damit einhergehenden Designentscheidungen funktionsfähig sind die entwickelten Lösungen das Grundproblem der Visualisierung und Analyse von Zeitreihen im Untersuchungsgegenstand so implementiert werden kann.

Es wurde zur Implementierung des POC die Technologien Three.js und React.js verwendet. Abbildung 14 können die mittels Three.js gerenderten dreidimensionalen Cubes entnommen werden. Als Datenquelle wurde eine Solr Instanz genutzt und über eine Dataservice Komponente abgerufen. Diese Darstellung zeigt, dass mittels Three.js somit eine große Menge an Daten problemlos gerendert werden kann.

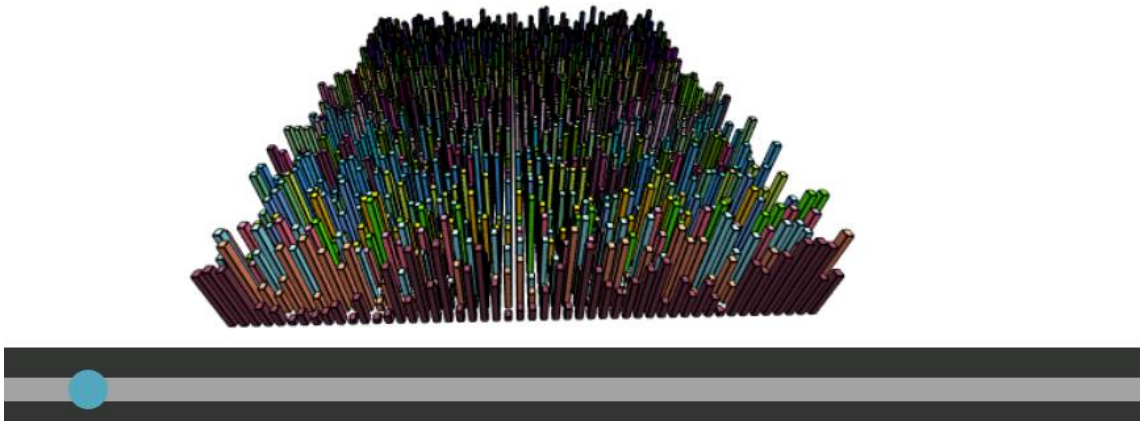

Abbildung 14: Three.js cubes Example [eigene Darstellung]

Die Auswahl der Architektur wurde aufgrund der möglichen Integration, aber auch der Stand-alone-Variante der Anwendung getroffen. Diese Stand-alone-Variante wird im Folgenden vorgestellt und diskutiert. Die Abbildung 14 zeigt das Userinterface des entwickelten POC, sowie die Visualisierung von Auslastungsdaten in einer dreidimensionalen Stadtmetapher mit Balken.

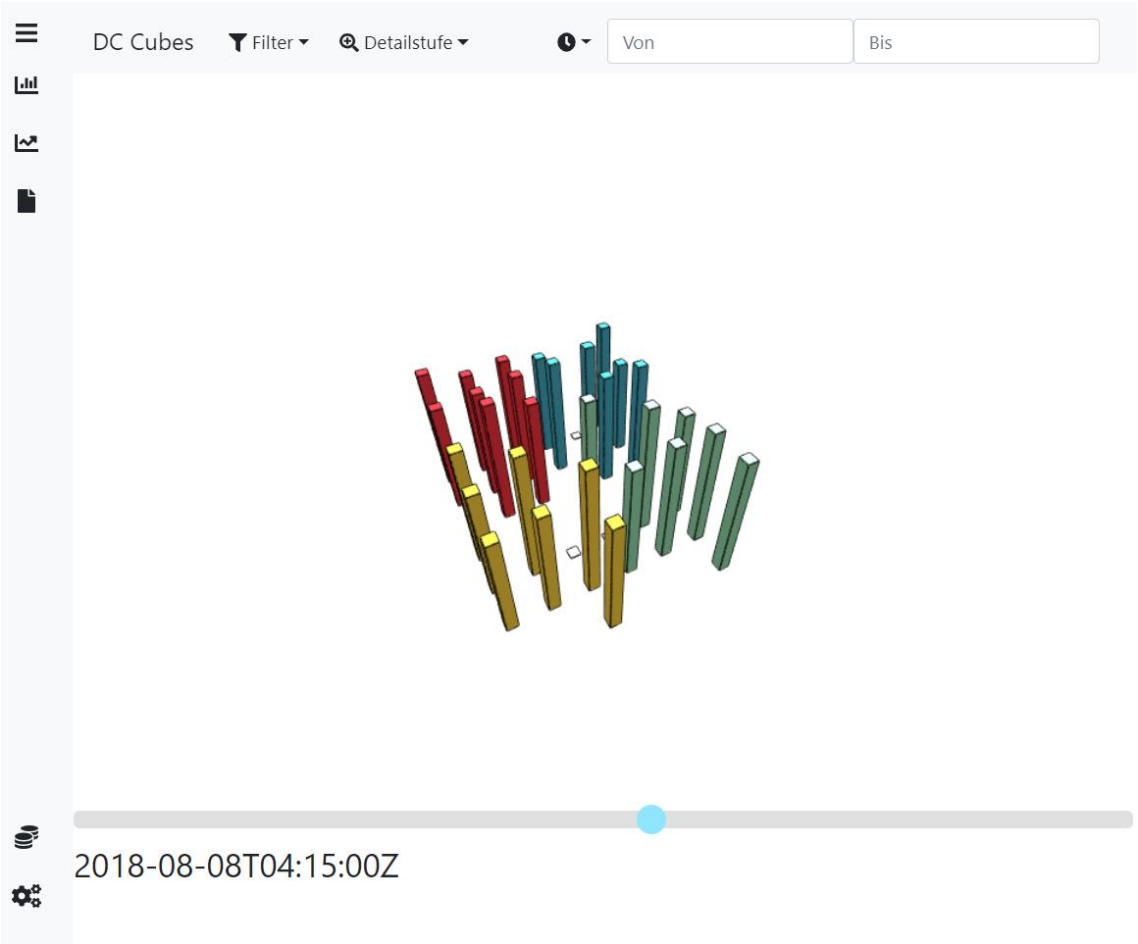

Abbildung 15: POC React.js, Three.js UI und Cubes [eigene Darstellung]

Wie die Entwicklung des POC aufweist, ist es möglich, die dieser Untersuchung zugrundeliegende Problemstellung mit dem hier entwickelten und vorgeschlagenen Technologiestack zu lösen und umzusetzen. Auch ist diese Softwarearchitektur stabil gegenüber den gestellten Kriterien bezüglich Integration und Performance. Des Weiteren kann aufgrund der Untersuchungsergebnisse in Kapitel 6 auch eine Zukunftssicherheit an die verwendeten Technologien unterstellt (Reliability) werden. Dies zeigt sich auch dadurch, dass sowohl Three.js als auch React.js einen enorm hohen Anwendungsgrad besitzen. Dies zeigt sich auch an der Größe der Git Repositorys hinsichtlich der Anzahl von Contributern. Darüber hinaus steht hinter React.js mit Facebook ein Global Player, welcher ein Vorantreiben der Entwicklung zusätzlich forciert.

8 Zusammenfassung

Die Softwarearchitektur für ein System zur Visualisierung und Analyse von Auslastungszeitreihen muss verschiedenen Qualitätsmerkmalen im besonders hohen Maß genügen. Dadurch, dass es sich dabei um eine im Enterprise Bereich einzuordnende Software handelt, sollte diese auch gegenüber Integrationen so offen wie möglich sein. Dafür existieren bereits verschiedene Standards von Anwendungen zur Visualisierung und Analyse von Daten, in welche die in dieser Untersuchung entwickelte Applikation integrierbar sein sollte, um eine möglichst hohe Wiederverwendbarkeit zu garantieren.

Die hier entwickelte Softwarearchitektur und Implementierung sollte es in seiner Anwendung es dem Benutzer ermöglichen, Anomalien in der Auslastung seines Rechenzentrums anhand der Auslastungsdaten schnell abzulesen, um sofort Maßnahmen zu deren Behebung einleiten zu können. Eine der Hauptanforderungen in der Problemstellung ist zusätzlich die einer dreidimensionalen Darstellung der Daten. Damit wäre zugleich auch ein höherer Informationsgehalt gewährleistet. Darüber hinaus sollte die Darstellung im Hinblick auf die unterschiedlichen Metriken der Auslastung filterbar sein, um in der Darstellung möglichst viele Indikatoren zur Auslastung präsentieren zu können.

Mit Hilfe der hier entwickelten Softwarearchitektur lassen sich diese Anforderungen wie in Kapitel 7 gezeigt, erfüllen und sogar noch um weitere Funktionen ergänzen.

Die in dieser Untersuchung entwickelte Softwarearchitektur unter der Verwendung von React.js und Three.js garantiert außerdem eine hohe Integrationsmöglichkeit in bereits bestehende und bereits verwendete Systeme zur Visualisierung und Analyse wie Kibana oder Grafana. Diese ermöglicht auch eine problemlose Erweiterung in der Anwendung, zumal weitere Funktionen bereits in Kibana integriert sind.

Zudem wird durch die Verwendung von Three.js zum Rendern der Visualisierung ein hoher Grad an Performance geliefert, welche es ermöglicht, die hohe Anzahl an Instanzen, welche die Bundesagentur für Arbeit betreibt, gleichzeitig zu visualisieren und somit auch eine Vergleichbarkeit auf einen Blick zu realisieren.

Zusammenfassend wird festgestellt, dass es möglich ist, mithilfe der hier entwickelten Softwarearchitektur die Auslastungsdaten des Datencenters der Bundesagentur für Arbeit zu jedem beliebigen Zeitpunkt, den jeweils aktuellen Grad der Auslastung zu erfassen. Dies wird durch die dreidimensionale Visualisierung, der Darstellung von unterschiedlichen Metriken und der Datennutzung aus verschiedenen Quellen erreicht. So gelingt es, auftretende Anomalien in der Auslastung des Datencenters schnell identifizieren und kurzfristig Maßnahmen zu deren Ursachenbehebung einleiten zu können.

Die aufgezeigten und dargelegten Möglichkeiten bietet die hier entwickelte Softwarearchitektur, wie die gesamte Untersuchung zeigt, da neben der dreidimensionalen Präsentation der Daten, das Gesamtergebnis die gewünschten und geforderten Qualitätsindikatoren beinhaltet.

9 Ausblick

Die Anforderungen an die in dieser Untersuchung zur Architekturerstellung und Implementierung eines Systems zur Visualisierung und Analyse von Auslastungszeitreihen beinhalten über die Visualisierung selbst hinaus auch eine Möglichkeit der Integrierung zu Vorhersagemöglichkeiten von Störungsfällen und Anomalien. Diese lassen sich in die entwickelte Anwendung integrieren durch die Verwendung von Machine-Learning-Algorithmen.

Die Integration von solch einer Funktion ermöglicht es dem Nutzer unter Betrachtung der bereits festgestellten Ereignisse und Erkennung von wiederkehrenden Mustern im Ablauf von bereits aufgetretenen Anomalien eine Vorhersage zu liefern, welche in Zukunft auftretende Störfälle vorhersagt. So kann ein noch schnelleres Eingreifen ermöglicht oder Störungsvorfälle sogar gänzlich vermieden werden.

10 Literaturverzeichnis

3D data visualizations? · Issue #1738 · vega/vega, https://github.com/vega/vega/issues/1738; Zugriff am 27.07.2019.

About | Node.js, https://nodejs.org/en/about/; Zugriff am 27.07.2019.

Angular Architecture overview, https://angular.io/guide/architecture; Zugriff am 27.07.2019.

Angular Github, https://github.com/angular/angular; Zugriff am 27.07.2019.

Apache Solr - Features, https://lucene.apache.org/solr/features.html; Zugriff am 27.07.2019.

Bjorn Sandvik: Github veiledning09, http://geoforum.github.io/veiledning09/; Zugriff am 04.08.2019.

Boehm, B.; Clark, B.; Abd-Allah, A.; Gacek, C.: On the Definition of Software System Architecture, Los Angeles, CA, 90089-0781, 1995.

Böhm, R.; Fuchs, E.: System-Entwicklung in der Wirtschaftsinformatik. vdf Hochschulverlag AG an der ETH Zürich, Zürich, 2015.

Bostock, M.: D3.js - Data-Driven Documents, https://d3js.org/; Zugriff am 27.07.2019.

Chen, C.-h.; Härdle, W.; Unwin, A.: Handbook of data visualization. Springer, Berlin, 2008.

Chircu, A.; Sultanow, E.; Baum, D.; Koch, C.; Seßler, M.: Visualization and Machine Learning for Data Center Management. Informatik 2019 Konferenz (2019).

Creating a scene – three.js docs, https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene; Zugriff am 27.07.2019.

d3.js Github, https://github.com/d3/d3; Zugriff am 13.08.2019.

deck.gl Github, https://github.com/uber/deck.gl; Zugriff am 13.08.2019.

Deck.gl-Example, https://deck.gl/#/examples/core-layers/geojson-layer-polygons; Zugriff am 27.07.2019.

elastic/kibana, https://github.com/elastic/kibana; Zugriff am 04.08.2019.

Fritz Solms: What is Software Architecture? Paper, South Africa, 2012.

Gilbert, P.; Ramakrishnan, K.; Diersen, R.: From Data Center Metrics to Data Center Analytics: How to Unlock the Full Business Value of DCIM (2013).

Grafana Developer Guide, https://grafana.com/docs/plugins/developing/development/; Zugriff am 03.09.2019.

Grafana Docs, https://grafana.com/docs/v4.3/; Zugriff am 23.08.2019.

Grafana license, https://github.com/grafana/grafana/blob/master/LICENSE; Zugriff am 23.08.2019.

ISO/IEC 25010:2011(en), Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en; Zugriff am 28.07.2019.

Jeffrey Heer: Vega License, https://github.com/vega/vega/blob/master/LICENSE; Zugriff am 05.07.2019.

Large-scale WebGL-powered Data Visualization, https://deck.gl/#/documentation/overview/introduction; Zugriff am 03.09.2019.

McGovern, J.: A practical guide to enterprise architecture. Prentice Hall/Professional Technical Reference, Upper Saddle River, NJ, 2004.

Modulecounts, http://www.modulecounts.com/; Zugriff am 27.07.2019.

Niekes/d3-3d, https://github.com/Niekes/d3-3d; Zugriff am 23.08.2019.

nodejs Github, https://github.com/nodejs/node; Zugriff am 27.07.2019.

Plugin Development | Kibana User Guide [7.3] | Elastic, https://www.elastic.co/guide/en/kibana/current/plugin-development.html; Zugriff am 03.09.2019.

Rainer Schlittgen, B. S.: Zeitreihenanalyse (2001), S. 1.

Referenzen · ELK Stack Erfolgsstories | Elastic Customers, https://www.elastic.co/de/use-cases/; Zugriff am 04.08.2019.

Rick Cattell: Scalable SQL and NoSQL Data Stores (2010).

Robert Eckstein: Java SE Application Design With MVC, https://www.oracle.com/technetwork/articles/javase/mvc-136693.html; Zugriff am 14.07.2019.

Robertson, S.; Robertson, J.: Mastering the requirements process. Addison-Wesley, Upper Saddle River, NJ, 2011.

SandDance, https://microsoft.github.io/SandDanc; Zugriff am 27.07.2019.

spalger: generate kibana plugin, https://github.com/elastic/kibana/blob/master/scripts/generate_plugin.js; Zugriff am 04.08.2019.

Steyer, M.; Softic, V.: Angular JS: Moderne Webanwendungen und Single Page Applications mit JavaScript. O'Reilly, Beijing, 2015.

Thomas Brey: The JavaScript Object Notation (JSON) Data Interchange Format, https://buildbot.tools.ietf.org/pdf/rfc7158.pdf; Zugriff am 27.07.2019.

three.js Github, https://github.com/mrdoob/three.js/; Zugriff am 19.07.2019.

uber/deck.gl, https://github.com/uber/deck.gl; Zugriff am 03.09.2019.

Vega axis documentation, https://vega.github.io/vega/docs/axes/#axis-orientation-reference; Zugriff am 27.07.2019.

Vega Graphs | Kibana User Guide [6.2] | Elastic, https://www.elastic.co/guide/en/kibana/6.2/vega-graph.html; Zugriff am 26.07.2019.

Vega Graphs | Kibana User Guide [7.2] | Elastic, https://www.elastic.co/guide/en/kibana/current/vega-graph.html; Zugriff am 27.07.2019.

Vega Parser API, https://vega.github.io/vega/docs/api/parser/; Zugriff am 03.09.2019.

WebGL: 2D and 3D graphics for the web, https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API; Zugriff am 27.07.2019.

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit ohne Hilfe Dritter und ohne Zuhilfenahme anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Potsdam, den 06. September 2019

_____

Unterschrift