Aufbau einer Integrationsarchitektur für ein Auslastungsanalysetool in Grafana

---

Roland Odorfer

01.03.2020

Version: 1.0

Technische Hochschule Nürnberg Georg Simon Ohm

TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Fakultät Informatik

Bachelorarbeit

Aufbau einer Integrationsarchitektur für ein Auslastungsanalysetool in Grafana

Roland Odorfer

Erstbetreuer Prof. Dr. Matthias Teßmann
Fakultät Informatik
Technische Hochschule Nürnberg Georg Simon Ohm

Zweitbetreuer Prof. Dr. Peter Rausch
Fakultät Informatik
Technische Hochschule Nürnberg Georg Simon Ohm

Betreuer Unternehmen Dr. Eldar Sultanow
Capgemini Deutschland GmbH

01.03.2020

Roland Odorfer
Aufbau einer Integrationsarchitektur für ein Auslastungsanalysetool in Grafana
Bachelorarbeit, 01.03.2020

Technische Hochschule Nürnberg Georg Simon Ohm
Fakultät Informatik
Hohfederstraße 40
90489 Nürnberg

Abstract

Im Rahmen der Digitalisierung steigt bei vielen Organisationen die Anzahl der Komponenten ihrer IT-Infrastruktur stark an. Dies geht mit einer erhöhten Komplexität der IT-Landschaft einher, was eine ganzheitliche Überwachung dieser mit traditionellen Überwachungstools erschwert. Im Rahmen der vorliegenden Arbeit soll eine Anwendung mit einer neuartigen 3D-Visualisierung, die das oben geschilderte Problem zu lösen versucht, in das Auslastungsanalysetool Grafana integriert werden. Das Ziel dieser Arbeit ist es zu untersuchen, inwiefern sich eine Integrationsarchitektur für Grafana konstruieren lässt, welche einerseits einen adäquaten Grad der Integration in Grafana ermöglicht, aber gleichzeitig eine angemessene Performanz und die Erweiterbarkeit der Architektur für ähnlich gelagerte Anwendungsfälle sicherstellt. Im Anschluss an die Umsetzung wird die fertige Architektur auf die geforderten Eigenschaften hin untersucht.

Abstract

In the context of digitization, the number of components in the IT infrastructure of many organizations is increasing rapidly. This is accompanied by an increased complexity of the IT landscape, which makes it difficult to monitor it holistically using traditional monitoring tools. In the context of this thesis, an application with a novel 3D visualization, which tries to solve the problem described above, shall be integrated into the workload analysis tool Grafana. The goal of this thesis is to investigate to what extent an integration architecture for Grafana can be constructed, which on the one hand allows an adequate degree of integration into Grafana, but at the same time ensures adequate performance and extensibility of the architecture for similar use cases. Following the implementation, the finished architecture is examined for the required characteristics.

Inhaltsverzeichnis

Abkürzungsverzeichnis **ix**

Abbildungsverzeichnis **xi**

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Zielsetzung . . . 3
1.3 Aufbau der Arbeit . . . 3

2 Grundlagen **5**
2.1 IT Operations Analytics . . . 5
2.2 Integrationsarchitektur . . . 5

3 Analyse **9**
3.1 Prototyp . . . 9
3.1.1 Architektur . . . 9
3.1.2 Benutzeroberfläche . . . 11
3.1.3 Datenanbindung . . . 13
3.1.4 Server . . . 18
3.2 Grafana . . . 19
3.2.1 Architektur . . . 19
3.2.2 Benutzeroberfläche . . . 20
3.2.3 Datenanbindung . . . 21
3.2.4 Plugin-Schnittstelle . . . 22
3.3 Anforderungen an die Integrationsarchitektur . . . 23

4 Implementierung **27**
4.1 Architektur . . . 30
4.2 Datenanbindung . . . 34
4.3 Benutzeroberfläche . . . 42
4.3.1 Panel . . . 44
4.3.2 Zeitauswahl . . . 45
4.3.3 Einstellungen . . . 47
4.3.4 Datenaktualisierung . . . 49
4.4 Server . . . 51

4.5 Grafana-Server 52
4.6 Bereitstellung 53

5 Ergebnisse **55**

6 Zusammenfassung und Ausblick **59**
6.1 Zusammenfassung 59
6.2 Ausblick 60

Literatur **63**

Abkürzungsverzeichnis

BA Bundesagentur für Arbeit 1
CORS Cross-Origin Resource Sharing 10
CSS Cascading Style Sheets 42
DBMS Datenbankmanagementsystem 36
HTTP Hypertext Transfer Protocol 9
IT Informationstechnologie 1
ITOA IT Operations Analytics 1
JSON JavaScript Object Notation 15
ML Machine Learning 1
REST Representational State Transfer 10
SRP Single-Responsibility-Prinzip 11
URL Uniform Resource Locator 13

Abbildungsverzeichnis

1.1 3D-Visualisierung 2

2.1 Typischer Prozess von IT Operations Analytics 5
2.2 Arten von Integrationsarchitekturen 6
2.3 Ebenen der Applikationsintegration 7

3.1 Architektur des Prototyps 10
3.2 Startseite des Prototyps 12
3.3 2D-Visualisierung des Prototyps 13
3.4 Sequenzdiagramm der Datenanbindung des Prototyps 14
3.5 Datenstruktur der Rohdaten im Prototyp 15
3.6 Organisation der IT-Infrastruktur 16
3.7 Datenstruktur für die 3D-Visualisierung 17
3.8 Architektur von Grafana 19
3.9 Einstiegsseite von Grafana 20
3.10 Illustration von CORS 22

4.1 Integrationsarchitektur 30
4.2 Adapter Pattern 35
4.3 Interface für die Datenquellen 37
4.4 Zusätzlicher Solr-Core für Vorhersage 39
4.5 Sequenzdiagramm der Datenanbindung des Plugins 41
4.6 Panel 44
4.7 Zeitauswahl 46
4.8 Übersicht der Interfaces für das Einstellungsmenü 47
4.9 Einstellungsmenü 48
4.10 Lifecycle-Methoden in Grafana 50

1 Einleitung

Das Messen von Verfügbarkeit und Auslastung von geschäftskritischen Anwendungen gewinnt durch die zunehmende Digitalisierung der Geschäftswelt immer mehr an Bedeutung. Eine eingeschränkte Verfügbarkeit von Unternehmensanwendungen führt oftmals zu empfindlichen Umsatzeinbußen [Ell14] und Reputationsschäden für das betroffene Unternehmen [Gro17].
Deshalb sollten Unternehmen in der Lage sein, Ausfälle in ihrer IT-Landschaft zu erkennen und im Optimalfall zeitnah beheben zu können. Hierbei kann Software aus dem Bereich *IT-Operations Analytics*[1] (ITOA) unterstützen. Durch neue Ansätze, wie *Machine Learning* (ML), können Lastspitzen vorhergesagt werden, um durch die rechtzeitige Bereitstellung von ausreichenden Ressourcen potentiellen Dienstausfällen vorzubeugen [Miy+19]. Die oben diskutierten Aspekte verdeutlichen den Bedarf nach einer effektiven ITOA-Lösung für den Unternehmensalltag.

1.1 Motivation

Eine intuitive Darstellung der zu überwachenden und analysierenden Daten zählt zu den Schlüsselfunktionalitäten einer Monitoring-Software, da sie eine effiziente und effektive Suche nach Anomalien ermöglicht.
Im Rahmen eines Forschungsprojekts bei der Bundesagentur für Arbeit (BA) wurde ein neuer Ansatz zur Visualisierung von Log-Daten aus Rechenzentren entwickelt [Chi+19]. Dabei wurde die in Abbildung 1.1 gezeigte 3D-Visualisierung konstruiert, welche die *Stadt-Metapher* mit dem sogenannten *Packing-Layout* nutzt [Sch13].
Hiermit lässt sich die hierarchische Beziehung zwischen Server, Cluster und Rechenzentrum übersichtlich darstellen. Ein weiterer Vorteil gegenüber der klassischen 2D-Visualisierung von Zeitreihendaten ist die Möglichkeit zur gleichzeitigen Darstellung einer Metrik von vielen unterschiedlichen Servern. Eine Befragung von Anwendern bestätigte die verbesserten Eigenschaften der neu entwickelten 3D-Visualisierung gegenüber der etablierten 2D-Visualisierung mit Zeitreihen [Chi+19].

Die etablierten Lösungen zur Analyse und Visualisierung von Log-Daten bieten eine solche 3D-Visualisierung derzeit nicht an. Dies ist auch bei dem ITOA-System

[1] „IT operation analytics (ITOA) is used for discovering complex patterns in data from IT systems.“ [DJ19]

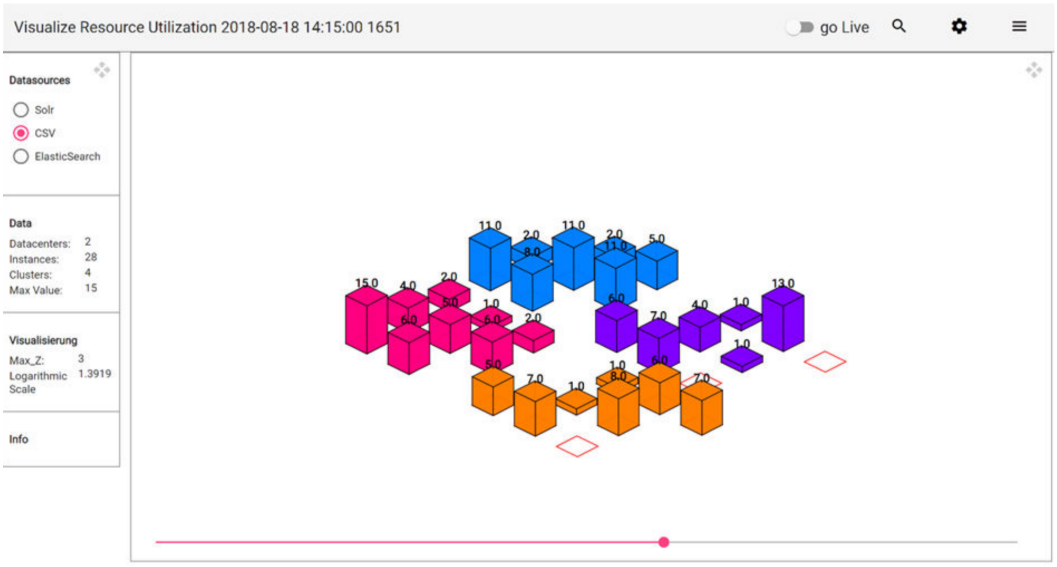

Abbildung 1.1: Die 3D-Visualisierung eines Rechenzentrums mit mehreren Clustern und Servern [Chi+19].

Grafana[2] der Fall. Aus diesem Grund wurde die neu entwickelte 3D-Visualisierung vorerst in Form einer eigenen unabhängigen Anwendung implementiert, um die Umsetzbarkeit und die verbesserten visuellen Eigenschaften anhand eines Prototyps zu demonstrieren.

Die BA und andere große Organisationen betreiben oftmals bereits eine sogenannte ITOA-Lösung (z.B. Kibana[3] oder Grafana), um Ihre IT-Landschaft zu überwachen. Außerdem haben Unternehmen und Organisationen das Ziel die Komplexität ihrer Anwendungslandschaft durch die Konsolidierung einzelner Anwendungen zu verringern, um die IT-Kosten zu senken und um wettbewerbsfähig zu bleiben [FP14]. Deshalb bietet sich die Integration des Prototyps in bestehende ITOA-Anwendungen an.
Ein weiteres Argument für die Integration ist die Visualisierung aller entscheidungsrelevanten Metriken an einem zentralen Ort, somit muss der Anwender nicht zwischen verschieden Anwendungen wechseln.
Außerdem müssen die Anwender keine neue Benutzeroberfläche erlernen, wenn eine Integration auf Ebene der Benutzeroberfläche stattfindet.
Aufgrund der oben genannten Argumente soll im Rahmen dieser Arbeit geklärt werden, inwiefern sich eine Integration des Prototyps in die jeweils bereits operativ in einer Organisation verwendete ITOA-Lösung - hier Grafana - realisieren lässt.

[2]https://grafana.com/ (besucht am 28. Nov. 2019)
[3]https://www.elastic.co/products/kibana (besucht am 29. Nov. 2019)

1.2 Zielsetzung

Die vorliegende Arbeit hat den Aufbau einer Integrationsarchitektur für ein Auslastungsanalysetool in Grafana zum Ziel. Bei dem Auslastungsanalysetool handelt es sich um einen Prototypen, der die in Abbildung 1.1 gezeigte 3D-Visualisierung enthält. Dieser Prototyp soll im Rahmen dieser Arbeit in das ITOA-System Grafana integriert werden. Die Integrationsarchitektur soll sich reibungslos in bestehende Systeme einbetten lassen und auch bei großen Datenmengen eine adäquate Performanz gewährleisten. Die Verfeinerung der oben genannten Ziele in konkrete Anforderungen wird in Abschnitt 3.3 erfolgen.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit ist folgendermaßen aufgebaut: In Kapitel 2 werden die für das Verständnis der nachfolgenden Kapitel notwendigen theoretischen Grundlagen vermittelt. Daraufhin wird in Kapitel 3 eine Analyse der einzelnen beteiligten Softwaresysteme und Komponenten vorgenommen. Mit den in Kapitel 3 gewonnen Erkenntnissen wird in Kapitel 4 die Implementierung der Integrationsarchitektur beschrieben. In Kapitel 5 sollen die Ergebnisse und Erkenntnisse dieser Arbeit dokumentiert werden. Hierbei wird insbesondere die Zielerreichung und die erfolgreiche Umsetzung der Anforderungen aus Abschnitt 3.3 überprüft werden. Das letzte Kapitel der vorliegenden Abschlussarbeit setzt sich aus einer Zusammenfassung und einem Ausblick zusammen.

2 Grundlagen

Zum besseren Verständnis der in der vorliegenden Arbeit verwendeten Terminologie, sollen in den folgenden Abschnitten einige Begrifflichkeiten und Technologien näher erklärt und wo nötig voneinander abgegrenzt werden.

2.1 IT Operations Analytics

IT Operations Analytics hilft dabei Anomalien und Muster in Betriebsdaten von operativen IT-Systemen zu entdecken [DJ19]. ITOA-Systeme werden von Mitarbeitern, die für die IT-Infrastruktur ihrer Organisation zuständig sind, für komplexe Datenanalyseaufgaben eingesetzt [DJ19]. Hierunter zählen unter anderem die Ursachenanalyse bei Fehlern oder Ausfällen und die Leistungskontrolle oder Wirkungsanalyse von Services [DJ19].

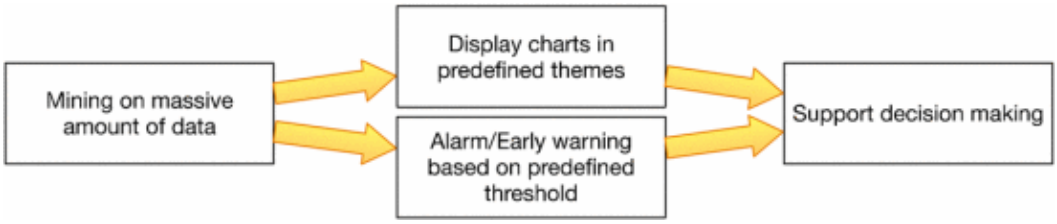

Abbildung 2.1: Ein typischer Prozess von IT Operations Analytics [He+16].

Abbildung 2.1 zeigt beispielhaft wie ITOA als Prozess in Organisationen ablaufen kann. Informationen, welche mithilfe von ITOA-Systemen gewonnen wurden, werden sowohl für die operative Geschäftsabwicklung als auch für den IT-Betrieb genutzt [He+16].

2.2 Integrationsarchitektur

Laut [Lui14] ist der Begriff *Integrationsarchitektur* folgendermaßen definiert:

> "The discipline of integration architecture essentially involves the touch points that exist between different components, whether they are software to software, hardware to hardware, or software to hardware. Integration architecture (aka middleware) is responsible for standards

and frameworks pertaining to communications and interfaces between application systems, operating system environments, and technologies."

[Lui14] beschreibt mehrere verschiedene Arten von Integrationsarchitekturen. Hierzu zählen die *Punkt-zu-Punkt-Architektur*, die *Hub-and-Spoke-Architektur* und *Serviceorientierte Architekturen*. Abbildung 2.2 bietet eine grafische Gegenüberstellung der 3 Architekturarten, welche im Folgenden näher erläutert werden sollen.

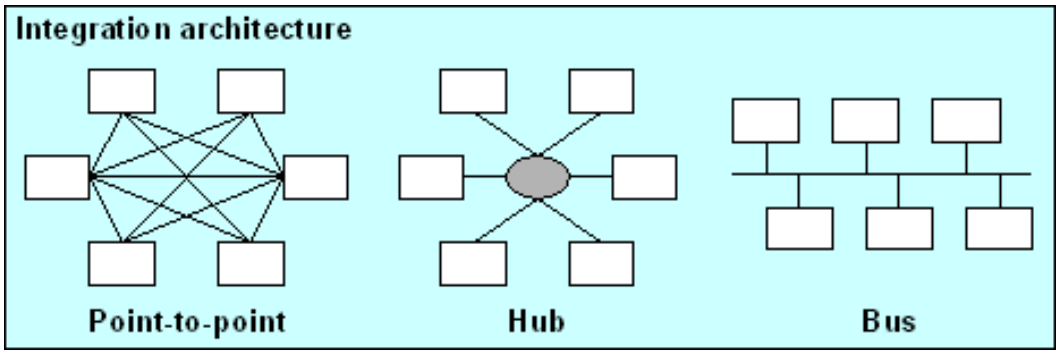

Abbildung 2.2: Die unterschiedlichen Arten von Integrationsarchitekturen [Cli05].

Bei der Punkt-zu-Punkt-Architektur kommunizieren die einzelnen Anwendungen direkt miteinander [Lui14].
Diese Architektur ist nur bei einer niedrigen Anzahl an zu integrierenden Anwendungen zu empfehlen, da für jede Verbindung zwischen den Anwendungen teils redundanter Programmcode implementiert werden muss [Gle05]. Dies kann die Wartbarkeit des Programmcodes verschlechtern [Gle05]. Außerdem führen die vielen Verbindungen und Schnittstellen schnell zu einer hohen Komplexität [Lui14].
Ein Vorteil liegt in der Möglichkeit der parallelen Entwicklung an verschiedenen Integrationsvorhaben, da beteiligte Entwickler keine gemeinsamen Schnittstellen berücksichtigen müssen [Gle05]. Des Weiteren ist es möglich, durch die Nutzung spezifischer Dateiformate oder Technologien eine bessere Performanz der Integrationsarchitektur zu erreichen [Gle05].

Die Hub-and-Spoke-Architektur sorgt für eine signifikante Verringerung der für eine Integration von mehreren Anwendungen benötigten Verbindungen [Lui14]. Dabei kommunizieren die einzelnen Anwendungen nicht mehr direkt miteinander, sondern über einen sogenannten *Hub* [Lui14].
Der Nachteil dieser Art von Integrationsarchitektur ist, dass der Hub als zentrales Element einen einzelnen potentiellen Ausfallspunkt darstellt. Zusätzlich gestaltet sich die Skalierung von Hub-and-Spoke-Architekturen schwierig [Lui14]. Schließlich ist noch zu erwähnen, dass Verbindungen, welche über einen Hub erfolgen, langsamer als eine einfache Punkt-zu-Punkt-Verbindung sind [Tro+04].

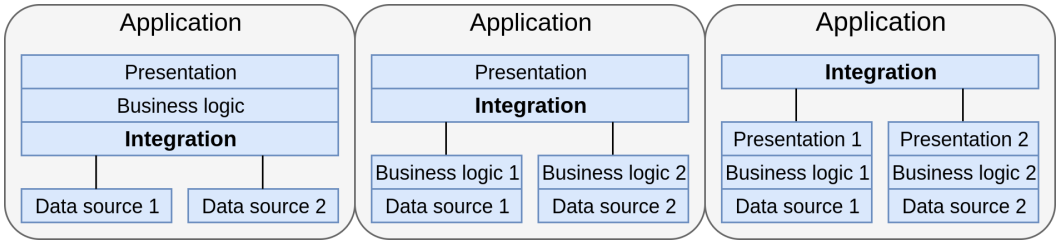

Abbildung 2.3: Die obere Grafik stellt die 3 unterschiedlichen Ebenen der Applikationsintegration gegenüber. Links ist die Integration auf *Datenebene* zu sehen. Mittig lässt sich die Integration auf *Geschäftslogikebene* betrachten. Rechts ist die Integration auf *Präsentationsebene* zu sehen.

Die Integration von Anwendungen mithilfe einer serviceorientierten Architektur umfasst die Kommunikation der einzelnen Anwendungen über einen *Enterprise Service Bus*, welcher die einzelnen Dienste einer Anwendung als Webservices bereitstellt [Lui14].
Der Vorteil dieser Art von Integration liegt unter anderem in der einfachen Wartbarkeit, der Flexibilität der Infrastruktur und der Skalierbarkeit [Mah07]. Außerdem sorgt die lose Kopplung von Anwendungen dafür, dass die IT-Systeme der jeweiligen Organisation auf Veränderungen des Geschäftsumfeldes agiler reagieren können [Mah07].
Die Einführung einer serviceorientierten Architektur in einer bestehenden Systemlandschaft bringt einen hohen initialen Aufwand mit sich [Mah07]. Es kann eine lange Zeit dauern bis sich die aus dem Einsatz der Architektur erhofften Einsparungen realisieren lassen [Mah07].

Im Rahmen dieser Arbeit soll eine geeignete Art von Integrationsarchitektur gefunden und implementiert werden. Die Argumente und Gründe, welche zu bestimmten Architekturentscheidungen geführt haben, sollen in den folgenden Kapiteln dokumentiert werden.

Eine Integrationsarchitektur umfasst laut [Lui14] folgende Bereiche:

- Nutzerintegration
- Prozessintegration
- Applikationsintegration
- Datenintegration
- Partnerintegration

In den folgenden Kapiteln wird der Fokus auf der Applikationsintegration liegen. Das hat den einfachen Grund, dass die fertige Integrationsarchitektur möglichst unabhängig von den organisationalen und informationstechnischen Rahmenbedingungen einer bestimmten Organisation oder eines Unternehmens funktionieren sollte.

Die Applikationsintegration ist prinzipiell auf 3 verschiedenen Ebenen möglich [Dan+07]. Namentlich sind dies die *Datenebene*, die *Geschäftslogikebene* und die *Präsentationsebene*. In Abbildung 2.3 findet sich eine grafische Übersicht der unterschiedlichen Ebenen.

Für das vorliegende Integrationsvorhaben gilt es zu untersuchen auf welcher Ebene die Applikationsintegration durchgeführt werden sollte. Hierbei gilt es die Erreichung, der im vorherigen Abschnitt definierten Zielsetzung, bestmöglichst sicherzustellen.

3 Analyse

Das folgende Kapitel beschäftigt sich mit der Analyse der an der Integration beteiligten Applikationen, deren Komponenten und Schnittstellen. Die Ergebnisse dieser Analyse werden im folgenden Kapitel in die Umsetzung der Integrationsarchitektur einfließen. Das Kapitel wird durch die systematische Dokumentation der Anforderungen an die zu entwickelnde Integrationsarchitektur komplementiert.

3.1 Prototyp

Im Nachfolgenden soll der im Rahmen des Forschungsprojektes [Chi+19] entwickelte Prototyp mit seiner neuartigen 3D-Visualisierung analysiert werden. Die Analyse wird sich auf die integrationsrelevanten Aspekte des Prototyps konzentrieren.

3.1.1 Architektur

Die Architektur des Prototyps orientiert sich an dem klassischen Client-Server-Modell, wobei der Client mithilfe von React[1] und der Server unter Zuhilfenahme der Laufzeitumgebung node.js[2] implementiert worden ist. Für die Datenanbindung kann ein beliebiger Solr-Datenbankserver konfiguriert werden. Abbildung 3.1 zeigt die Gesamtarchitektur des Prototyps.
Die Kommunikation zwischen Client, Server und Solr-Server (vgl. Abbildung 3.1) geschieht über das Protokoll HTTP. Der Zugriff auf den Client funktioniert über einen Webbrowser.

Der Client (vgl. Abbildung 3.1) befasst sich mit der Visualisierung der Zeitreihendaten und der Bereitstellung der gesamten Benutzeroberfläche des Prototyps. Hierzu zählen unter anderem die Schaltflächen zur Anpassung der Visualisierungen. Darüber hinaus ist die Konfiguration der Datenbank (Solr) aus der Benutzeroberfläche heraus möglich. Des Weiteren kann die Vorhersagefunktionalität in Form eines Python-Skripts (vgl. *ML Script* in Abbildung 3.1) aus dem Client heraus angestoßen werden.
Die Daten für die Visualisierung bezieht der Client von einem sogenannten *Core*,

[1] https://reactjs.org/ (besucht am 02. Dez. 2019)
[2] https://nodejs.org/ (besucht am 02. Dez. 2019)

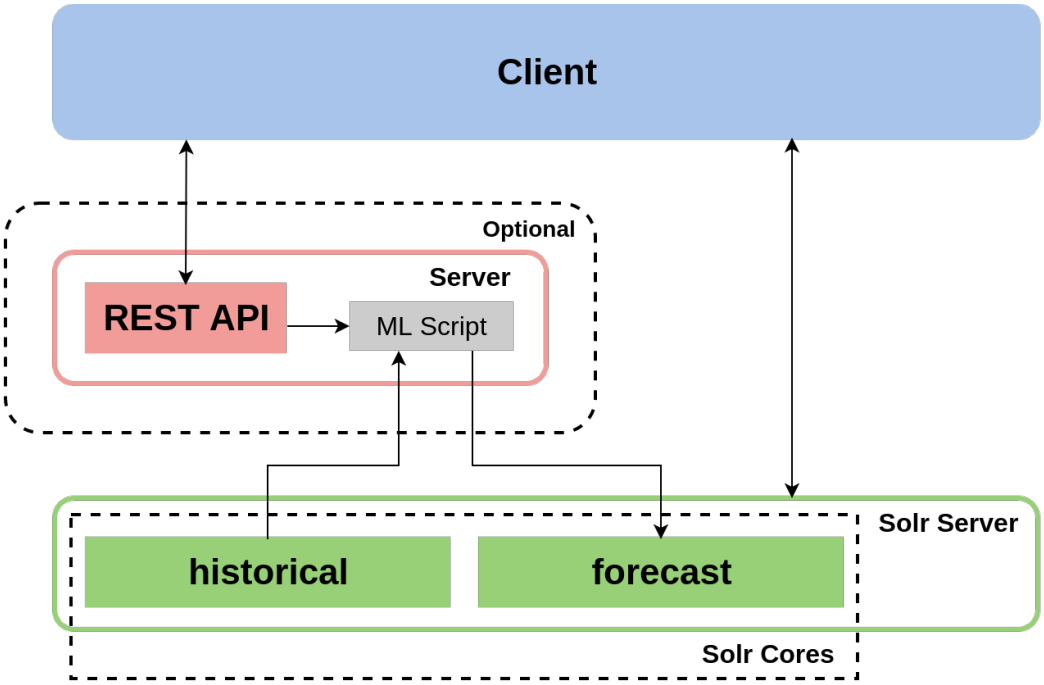

Abbildung 3.1: Die Architektur des Prototyps.

welcher auf einem Solr-Server liegt. Dabei handelt es sich um die Instanz einer einzelnen Solr-Datenbank.

Die REST-Schnittstelle (vgl. Abbildung 3.1) ist mithilfe von node.js umgesetzt worden und wird durch den Server bereitgestellt. Diese dient der Bereitstellung der Vorhersagefunktionaliät des Prototyps. Dabei handelt es sich um einen Machine-Learning-Algorithmus, dessen Ziel es ist, anhand der vorhanden Zeitreihendaten aus der Vergangenheit, eine Prognose für die Zunkunft zu erstellen. Hierbei ist insbesondere die Vorhersage von potentiellen Auslastungsspitzen von Interesse, da anhand dieser dynamisch die benötigten Ressourcen bereitgestellt werden können, um eine möglichst hohe Serviceverfügbarkeit sicherzustellen.

Der Solr-Server (vgl. Abbildung 3.1) fungiert als Datenquelle in der Architektur. Damit die Anbindung einer externen Solr-Datenbank an den Client erfolgen kann, muss der Solr-Server so konfiguriert werden, dass die ausgehenden HTTP-Header die notwendigen Felder enthalten, um die *CORS-Anfragen* des Clients zu erlauben. Eine ausführliche Beschreibung und Erklärung des Sicherheitsmechanismus *Cross-Origin Resource Sharing* (CORS) folgt im Unterkapitel 3.2.3.

Im Rahmen der Entwicklung des Prototyps wurde aus Gründen des Zeitdrucks wenig Augenmerk auf Aspekte wie die Performanz oder die Skalierung der Anwendung gelegt. Dies manifestiert sich unter anderem in Verzögerungen beim Ladevorgang der Visualisierungen.

Der Grund für die mangelhafte Performanz liegt unter anderem an der Tatsache, dass sehr viele aufwändige Berechnungen und Aggregierungen, welche für die Visualisierungen benötigt werden, im Client des Prototyps und damit im Browser des Anwenders stattfinden. Dies ist insbesondere bei großen Datenmengen problematisch, da die Rechenleistung der Clients im Vergleich zu Servern oftmals deutlich geringer ist.
Eine weitere Ursache für Verzögerungen bei der Ausführung des Prototyps stellen redundante Funktionsaufrufe und Variablenberechnungen dar. Ein Beispiel hierfür ist das mehrfache Durchlaufen des selben Arrays oder die wiederholte Berechnung des Durchschnittswerts derselben Zeitreihe.

3.1.2 Benutzeroberfläche

Die Benutzeroberfläche des Prototyps wurde mithilfe des JavaScript-Frameworks React und der Programmiersprache TypeScript entwickelt. Im Rahmen der Entwicklung des Prototyps wurde ein papierbasierter Mockup zur Demonstration der vollständigen Benutzeroberfläche erstellt. Im Anschluss wurden die einzelnen Elemente der Benutzeroberfläche des Mockups - mittels der in beschriebenen [Inc19] Vorgehensweise - in eine sogenannte *Komponentenhierarchie* überführt. Hierbei fand das Single-Responsibility-Prinzip (SRP) Einsatz. Dabei handelt es sich um eine Methodik, bei der die Benutzeroberfläche solange in einzelne Komponenten und Subkomponenten aufgeteilt wird, bis jede Komponente ausschließlich einem Zweck dient [Inc19]. Dieses Vorgehen gewährleistet eine hohe Modularität der zu entwickelnden Software [Mar03]. Je modularer eine Anwendung gestaltet ist, desto geringer sind die Auswirkungen der Änderung an einer Komponente auf die anderen Komponenten. [ISO11]. Diese Erkenntnis sollte die Integration in Grafana erheblich erleichtern.

Nachfolgend sollen die wichtigsten Bestandteile der Benutzeroberfläche des Prototyps näher beschrieben werden. Dabei wird auch auf Komponenten und Funktionalitäten eingegangen, welche bisher zwar noch nicht in der Anwendung enthalten sind, zukünftig jedoch im Rahmen der Entwicklung der Integrationsarchitektur implementiert werden sollen.

Die Kopfleiste (vgl. Abbildung 3.2) enthält mehrere Menüelmente, welche nur als *Mockup* in der Anwendung vorhanden sind. Diese Elemente enthalten noch keine Funktion und dienen ausschließlich der visuellen Demonstration.
Ein solches Element ist die Schaltleiste zur Datumsauswahl. Hiermit soll sich die temporale Eingrenzung, der vom Server in den Client zu ladenden Rohdaten, einrichten lassen. Das Zeitintervall lässt sich zwar bereits über das Element einstellen und auch in einer Zustandsvariable speichern, jedoch hat dies noch keinen Effekt auf den Service, welcher die Daten vom Datenbankserver holt. Dieser verfügt nämlich noch

Abbildung 3.2: Die Startseite des Prototyps.

über keine Methode, welche die dafür notwendige Programmlogik implementiert. An dieser Stelle ist zu erwähnen, dass es sich hierbei um die Auswahl des Zeitintervalls für die Daten handelt, welche die Anwendung von Solr holt. Dagegen lassen sich über die 2D-Visualisierung (vgl. Abbildung 3.3) die Daten, welche bereits lokal in den Client geladen wurden weiter selektieren.
Die voll funktionsfähigen Elemente der Kopfzeile umfassen eine Schaltfläche zur Aktivierung der Vorhersagefunktionalität und eine Schaltfläche für die manuelle Aktualisierung der lokalen Datenbasis.

Die 3D-Visualisierung (vgl. Abbildung 3.2) bildet den Kern der Anwendung und ist deshalb zentral in der Benutzeroberfläche platziert. Zur Implementierung der 3D-Visualisierung wurde die JavaScript-Bibliothek three.js[3] verwendet. Sie dient der Erstellung von 3D-Visualisierungen in Browsern und baut auf WebGL[4] auf [Dir18]. Der Anwender kann mithilfe der Pfeiltasten und der Maus die 3D-Visualisierung erkunden. Daneben besteht auch die Möglichkeit zum Zoomen.
Die beiden Dropdown-Elemente am oberen Rand der 3D-Visualisierung (vgl. Abbildung 3.2) sind Mockups. Sie sollen einmal den dynamischen Wechsel zwischen den verschiedenen Aggregationstypen (Minimum, Maximum, Mittelwert und Summe) und den jeweils verfügbaren Kennzahlen ermöglichen.

Die 2D-Visualisierung (vgl. Abbildung 3.2) dient neben der übersichtlichen Darstellung der Aggregierungen (jeweils Maximum, Mininum und Mittelwert über alle Werte einer Kennzahl zu einem bestimmten Zeitpunkt) der jeweils geladenen Zeitreihendaten, auch der Selektion jener. Dabei ist es möglich einen konkreten Zeitpunkt

[3] https://threejs.org/ (besucht am 05. Dez. 2019)
[4] https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API (besucht am 05. Dez. 2019)

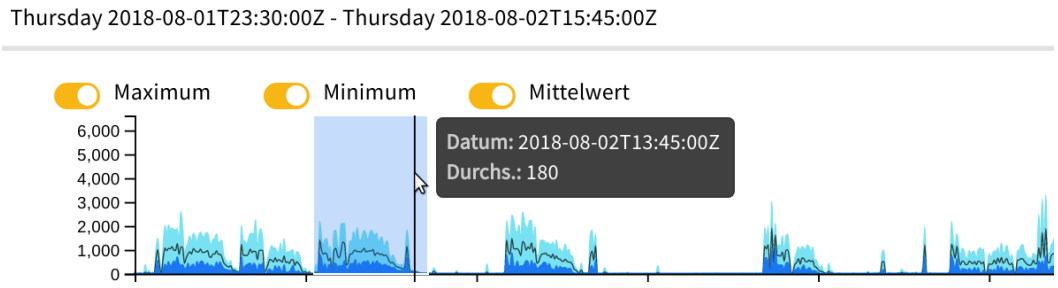

Abbildung 3.3: Die 2D-Visualisierung hilft unter anderem bei der übersichtlichen Darstellung der Maxima, Minima und Mittelwerte der geladenen Zeitreihendaten. Entlang der x-Achse sind Zeitstempel notiert, wohingegen die y-Achse die aggregierten Werte einer Kennzahl anzeigt.

oder ein Zeitintervall auszuwählen. Die korrespondierenden Daten werden dann entsprechend in der 3D-Visualisierung dargestellt. Bei der Auswahl eines Zeitintervalls (vgl. Abbildung 3.3) werden derzeit nur die Werte eines Zeitpunkts angezeigt. In der fertigen Integrationsarchitektur soll automatisch eine Aggregation der Daten in der 3D-Visualisierung erfolgen. Diese Funktionalität gilt es im Rahmen des Aufbaus der Integrationsarchitektur noch umzusetzen.
Die Implementierung der 2D-Visualisierung erfolgte unter der Verwendung der JavaScript-Bibliothek D3.js[5].

Über die Seitenleiste (vgl. Abbildung 3.2) gelangt der Nutzer unter anderem zu der Einstellungseite für die Datenquellen. Hierin finden sich datenbankspezifische Einstellungen. Beispiele dafür umfassen unter anderem die Auswahl eines Solr-Cores oder die Änderung des Uniform Resource Locator (URL) des Solr-Servers.
Aktuell lassen sich diese und andere Konfigurationen in der Anwendung nur während einer Browser-Session nutzen, da der Prototyp zum Zeitpunkt der Analyse über keinen Persistenzmechanismus für die Einstellungsdaten verfügt.

3.1.3 Datenanbindung

Der Prototyp unterstützt als Datenquelle derzeit nur Solr. Eine dynamische Anbindung verschiedener DBMS ist derzeit nicht ohne Weiteres möglich, da die Architektur des Prototyps hierfür nicht ausgelegt ist. Dies liegt unter anderem an der Tatsache, dass die für die Datenanbindung verantwortliche Klasse `DataService` nicht generisch und erweiterbar gestaltet worden ist.
Die neu entwickelte Visualisierung von IT-Betriebsdaten des vorliegenden Prototyps wird aus Zeitreihendaten erstellt und bedarf einer umfangreichen Transformationslogik. Im Folgenden sollen die daran beteiligten Transformationsoperationen und Datenstrukturen beschrieben und in Hinsicht auf die Integration analysiert werden.

[5] https://d3js.org/ (besucht am 07. Dez. 2019)

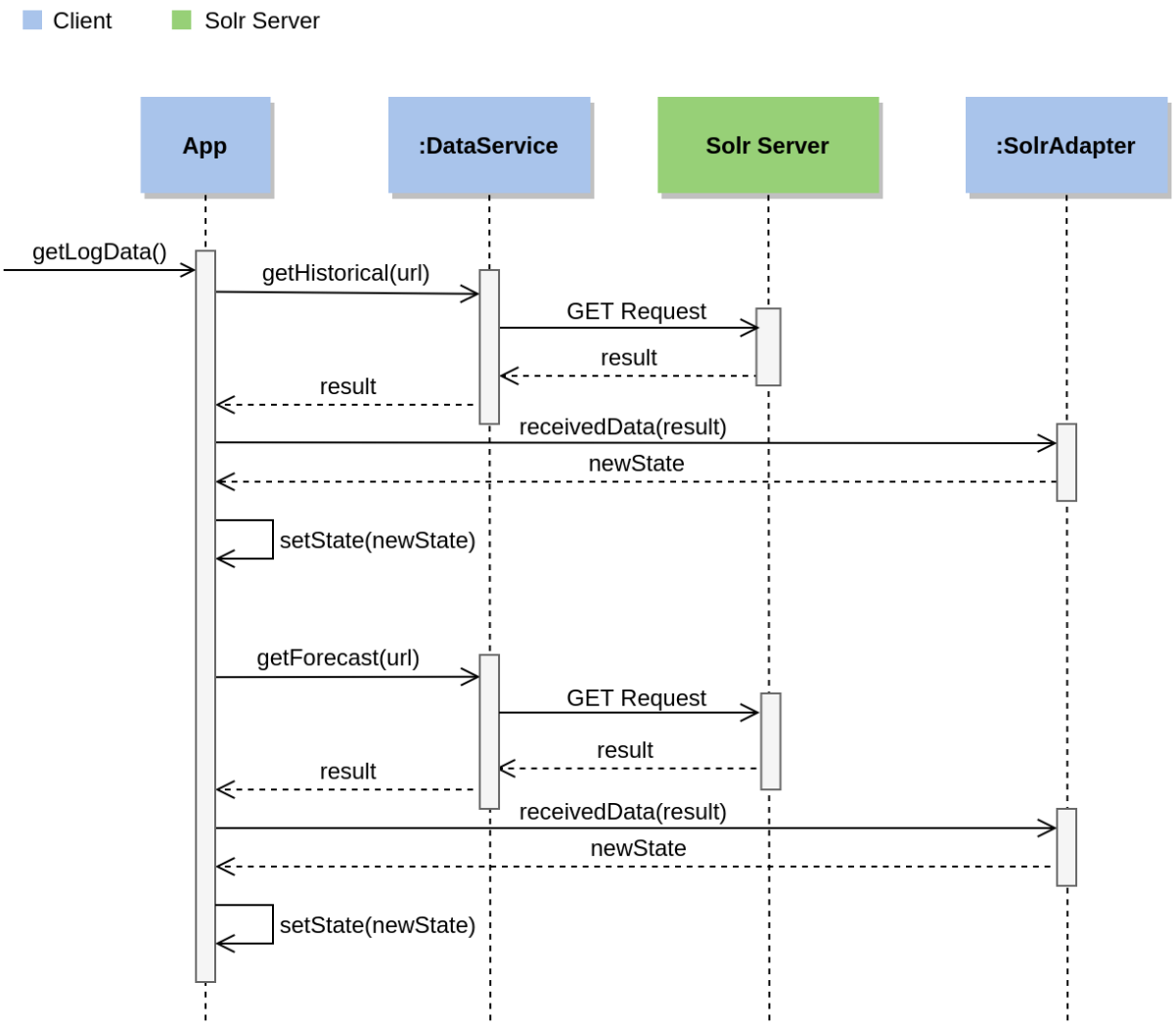

Abbildung 3.4: Das Sequenzdiagramm zeigt das Zusammenspiel der an der Datenanbindung beteiligten Module und Methoden.

Abbildung 3.4 bietet eine Übersicht über den zeitlichen Ablauf dieser Transformationen und der beteiligten Module und Methoden. Außerdem werden die untereinander bestehenden Abhängigkeiten transparent gemacht.

Die Datenanbindung wird durch den Aufruf der Methode `getLogData` der Wurzelkomponente `App` initalisiert. Dieser Aufruf findet unter anderem beim Start der Anwendung und bei der Aktualisierung der Daten statt. In dieser Methode wird eine Objektinstanz der Klasse `DataService` erstellt.
Im Anschluss wird die Methode `getHistorical` der Objektinstanz aufgerufen. Dabei wird der Methode eine URL übergeben. Diese setzt sich aus der Adresse des Solr-Servers, dem Namen des Solr-Cores mit den historischen Daten und einer Datenbankabfrage zusammen. Im vorliegenden Fall enthält die voreingestellte Datenbankabfrage keinerlei Selektionskriterium, jedoch lässt sich dies bei Bedarf über

die Einstellungsseite für Solr durch ein einfaches Textfeld ändern.


```
[
    {
        "timestamp":"2018-08-01T00:00:00Z",
        "host":"0",
        "cluster":8,
        "dc":0,
        "perm":2,
        "instanz":"8",
        "verfahren":"0",
        "service":"0",
        "response":200,
        "count":1,
        "minv":152,
        "maxv":152,
        "avg":152.0,
        "var":0.0,
        "dev_upp":152.0,
        "dev_low":152.0,
        "perc90":152.0,
        "perc95":152.0,
        "perc99":152.0,
        "sum":152,
        "sum_of_squares":23104,
        "server":"PBZ08E00_PERM02_S08_OSB",
        "id":"1c9c75bd-4f1c-4154-862b-1399b93bc055",
        "_version_":1652096008355577856
    },
    ...
]
```


Abbildung 3.5: Die Abbildung zeigt einen beispielhaften Ausschnitt der Rohdaten, die im Prototyp transformiert werden müssen, damit daraus die Visualisierungen erzeugt werden können.

Anschließend sendet die Methode `getHistorical` eine HTTP-Anfrage mit der Anfragenmethode GET an die als Argument übergebene URL. Der Solr-Server nimmt die Anfrage entgegen und führt die Datenbankabfrage auf dem in der URL spezifizierten Solr-Core aus. Das Ergebnis dieser Anfrage (`result` in Abbildung 3.4) bildet ein Datum, welches im Datenaustauschformat JSON formatiert ist. Ein beispielhaftes Datum ist ausschnittsweise in Abbildung 3.5 zu sehen.

Die Struktur und der Aufbau des Ergebnisses ist sehr einfach gehalten. Es handelt sich um ein Array, welches einzelne Objekte mit verschiedenen Schlüssel/Wert-Paaren beinhaltet. Die Struktur der einzelnen im Array enthaltenen Objekte ist flach und verfügt damit über keine Verschachtelungen oder Ähnliches.
In jedem Objekt sind verschiedene Kennzahlen wie zum Beispiel *verfahren* (Auslastung der Instanz) als Schlüssel und die jeweils korrespondierenden Kennziffern als Wert enthalten. Außerdem verfügt jedes Objekt über einen Zeitstempel und Identifikatoren für das Rechenzentrum, das Cluster und die Instanz aus welcher die gemessenen Kennzahlen stammen.
Die Datenstruktur der Rohdaten ist besonders wichtig, denn Sie bildet das datentechnische Fundament des Prototyps. An vielen Stellen im Quellcode wird direkt auf

Elemente der Struktur zugegriffen. Deshalb ist es notwendig, dass diese zur Laufzeit auch vorhanden sind. Die Daten stammen aus dem Rechenzentrums der BA und wurden für die Extraktion in die Solr-Datenbank an einigen Stellen bereinigt und angepasst.

Nachdem die Abfrage auf dem Solr-Core erfolgreich durchgeführt wurde, sendet der Solr-Server das Ergebnis an den Client zurück. Im Anschluss wird das Ergebnis in der Methode `getHistorical` in Empfang genommen und dient als Argument der Methode `receivedData`, welche Teil einer neuen Objektinstanz der Klasse `SolrAdapter` ist (vgl. Abbildung 3.4). Bei dieser Klasse handelt es sich um eine Sammlung von Methoden zur Transformation der vorliegenden Rohdaten (vgl. Abbildung 3.5) und zur Berechnung von Werten für die Visualisierungen.
Ein Beispiel hierfür ist die Höhe der Balken in der 3D-Visualisierung. Diese wird logarithmiert dargestellt, um Ausreiserwerte anwenderfreundlich darstellen zu können. Für die Berechnung des Logarithmus ist unter anderem die Berechnung der maximalen Höhe (der größte Messwert zu einem bestimmten Zeitpunkt) mithilfe der Methoden des Adapters von Nöten.

Die Notwendigkeit einer Transformation der Rohdaten durch den Adapter liegt in der Beschaffenheit der Rohdaten begründet, denn diese entsprechen nicht der für die 3D-Darstellung notwendigen Datenstruktur.

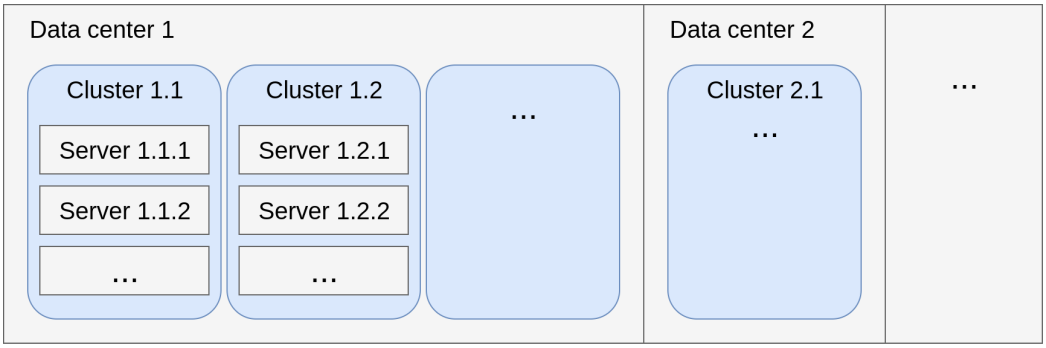

Abbildung 3.6: Die Hierarchie zwischen Rechenzentrum, Cluster und Server.

Der 3D-Visualisierung liegt eine spezielle Datenstruktur zu Grunde, welche die in Abbildung 3.6 zu sehende natürliche Hierarchie zwischen Servern, Cluster und Rechenzentren abbildet. Sie erlaubt eine dynamische Konstruktion der 3D-Visualisierung von IT-Infrastrukturen mithilfe von three.js. Der Adapter ist in der Lage die Datenstruktur der Rohdaten (vgl. Abbildung 3.5) in diese Form zu überführen. Dabei werden die Server, Cluster und Rechenzentren einer IT-Landschaft in eine Hierarchie eingeordnet und über eindeutige Identifikatoren miteinander assoziiert. Diese Organisation der IT-Infrastruktur entstammt dem Rechenzentrum der BA. Abbildung 3.7 zeigt die TypeScript-Interfaces der Datenstruktur.
Im Anschluss an die erfolgreiche Ausführung der Berechnungen und Transforma-

```
interface TimeSeries {
    timeSeries: Map<string, DCState>
}

interface DCState {
    datacenters: Map<string, Datacenter>;
}

interface Datacenter {
    clusters: Map<string, Cluster>;
}

interface Cluster {
    instances: Map<string, Instance>;
}

interface Instance {
    measure: number;
}
```

Abbildung 3.7: Die Datenstruktur für die 3D-Visualisierung.

tionen des Adapters werden die Ergebnisse (`newState` in Abbildung 3.4) zentral als Zustandsvariablen der `App`-Komponente gespeichert. Somit können die in der Komponentenhierarchie nachgelagerten Komponenten auf die Zustandsänderung reagieren und die aktualisierten Daten verarbeiten.

Der oben beschriebene Vorgang bezieht sich auf die Anbindung der gemessenen IT-Infrastrukturdaten an den Prototyp. Im Anschluss an diesen Vorgang findet analog dazu dasselbe für die Anbindung der aus den vorhandenen Infrastrukturdaten generierten Vorhersagewerte statt (vgl. Abbildung 3.4).

Bei der Analyse der Datenanbindung wurden einige Schwächen der aktuellen Implementierung des Prototyps offensichtlich. Diese sollen im Nachfolgenden beschrieben und begründet werden.

In der Praxis unterscheidet sich der Aufbau von IT-Infrastrukturdaten teilweise erheblich und kann von Anwendung zu Anwendung als auch Organisation zu Organisation sehr stark voneinander abweichen. Die aktuelle Gestaltung der Anwendung erschwert eine Anpassung an Datenstrukturen, welche von der in Abbildung 3.5 gezeigten Struktur abweichen. Ein Beispiel hierfür ist, der direkte Zugriff auf Schlüsselwerte der Rohdaten.

Jeder Aufruf der Methode `getLogData` verursacht eine Anfrage, welche den gesamten Datenbestand der Solr-Cores in den Client lädt. Dies sorgt bei größeren Datenmengen zu spürbaren Verzögerungen, denn die gesamte Transformations- und

Berechnungslogik muss dabei jedes Mal durchlaufen werden. Eine Selektion eines bestimmten Zeitraumes ist derzeit in der Anwendung nicht vorhanden.

Ferner werden bei jedem Aufruf von `getLogData` die Daten der Vorhersage in den Client geladen. Dies geschieht unabhängig davon, ob der Vorhersagemodus in der Anwendung überhaupt vom Nutzer aktiviert wurde und stellt somit einen unnötigen Verbrauch von begrenzter Rechnerkapazität dar.

Ein weiterer Nachteil ist die getrennte Speicherung von Historie- und Vorhersagedaten in zwei getrennten Cores auf dem Solr-Server. Dies sorgt ebenfalls für Einbußen bei der Performanz des Prototypen, da bei jedem Aufruf der Methode `getLogData` die Ergebnisse und Variablen des Solr-Adapters für die Vergangenheit und die Vorhersage verschmolzen werden müssen, damit die korrekten aggregierten Werte berechnet werden können.

Außerdem ist die flexible Erweiterung der Anwendung um zusätzliche Datenquellen wie zum Beispiel Prometheus oder Elasticsearch derzeit nicht ohne Weiteres möglich. Dies liegt an der Strukturierung des Services, welche keinen dynamischen Wechsel der Datenquelle sowohl zur Laufzeit als auch zur Entwicklung erlaubt.

Eine weitere Limitierung des Prototyps stellt die fehlende Möglichkeit der Parameterisierung der zu visualisierenden Kennzahl dar. Derzeit ist ausschließlich die Auswertung der Auslastung (Schlüssel `count` in Abbildung 3.5) von Instanzen möglich, denn diese Variable ist über die Anwendung verteilt fest in die Programmlogik codiert worden.

3.1.4 Server

Auf dem Server (vgl. Abbildung 3.1) wurde eine REST-Schnittstelle implementiert. Diese enthält verschiedene Endpunkte, welche dazu dienen den Prozess der Vorhersage anzustoßen. Hierbei wird das ebenfalls auf dem Server befindliche Machine-Learning-Skript (vgl. *ML Script* in Abbildung 3.1) ausgeführt. Das Skript nutzt einen Machine-Learning-Algorithmus, um anhand der historischen Daten aus dem Solr-Core `historical` (vgl. Abbildung 3.1) eine Vorhersage für die Zukunft zu generieren. Die prognostizierten Zeitreihendaten werden daraufhin in einem neuen Solr-Core namens `forecast` (vgl. Abbildung 3.1) gespeichert. Dieser Solr-Core enthält ausschließlich die zuletzt generierten Werte der Vorhersage.

Derzeit ist die Vorhersage von Zeitreihendaten nur für eine Kennzahl, nämlich die Auslastung (vgl. `count` in Abbildung 3.5) möglich, um die Vorhersage von weiteren Kennzahlen zu ermöglichen müsste der Programmcode an einigen Stellen angepasst

werden. Unter anderem müssten alle verfügbaren Kennzahlen in der Anwendung erfasst werden. Dies liegt daran, dass die Programmlogik derzeit keine Möglichkeit zur Festlegung der jeweils verfügbaren Kennzahlen bietet.

Eine weitere Schwachstelle des Servers stellt die nicht vorhandene Option zur Persistierung der Einstellungen des Anwenders dar. Dies führt dazu, dass bei jedem Neustart der Anwendung die getroffenen Einstellungen (zum Beispiel für die Datenbanken) verworfen werden. Der Anwender muss die Einstellungen also bei jedem Neustart erneut vornehmen.

3.2 Grafana

Anknüpfend an die Analyse des Prototyps, soll in diesem Kapitel das ITOA-System Grafana und seine Komponenten näher untersucht werden. Die Untersuchung soll sich auf die für die Integration relevanten Aspekte fokussieren.

3.2.1 Architektur

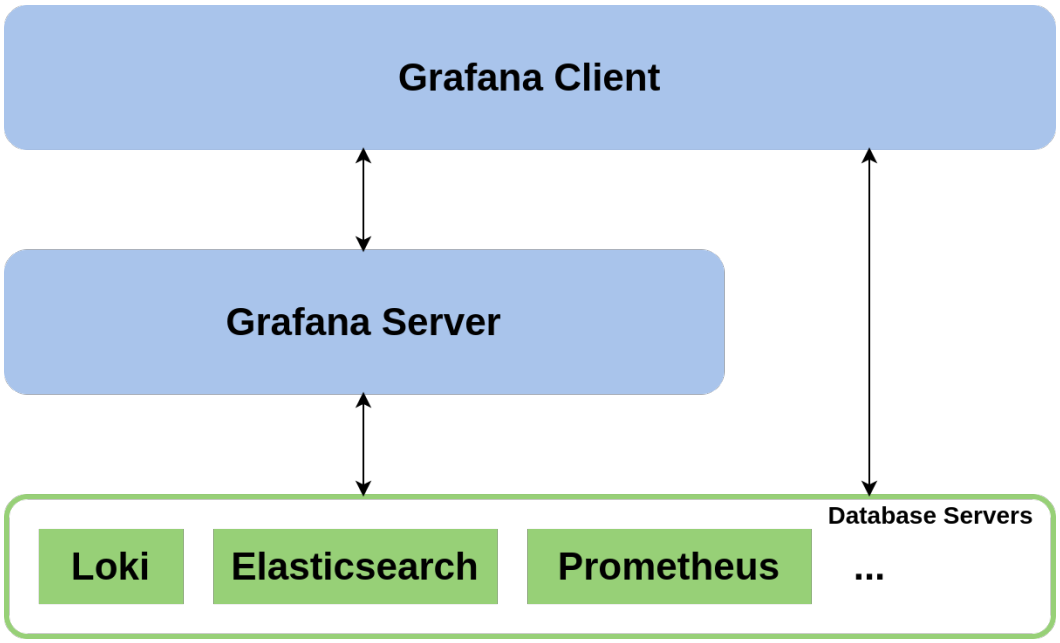

Abbildung 3.8: Die Architektur von Grafana.

Die Architektur von Grafana entspricht analog zum Prototypen ebenfalls weitestgehend der Client-Server-Architektur (vgl. Abbildung 3.8), wobei der Grafana-Client mit der Visualisierung der Daten betraut ist und der Grafana-Server sich um die Persistierung von Einstellungen und Bereitstellung von Services kümmert. Die Kommunikation zwischen Client und Server findet über HTTP statt [Lab19b].
Der Grafana-Client wurde mithilfe der beiden JavaScript-Frameworks React und

Angular[6] umgesetzt [Öde18]. Währenddessen ist der Grafana-Server unter Verwendung der Programmiersprache Go[7] entwickelt worden [Bin15]. Die Datenbankserver werden über eine HTTP-Schnittstelle an Grafana angebunden. Die Anbindung erfolgt wahlweise direkt an den Grafana-Client oder indirekt mit dem Grafana-Server als zwischengeschalteten Proxy (vgl. Abbildung 3.8) [Lab19e].
Der Client wird - analog zum Prototyp - über einen Webbrowser geöffnet [Lab19h].

3.2.2 Benutzeroberfläche

Die Benutzeroberfläche von Grafana setzt sich aus unterschiedlichen Komponenten zusammen. Diese und der allgemeine Aufbau der Benutzeroberfläche sollen nachfolgend näher erläutert werden.

Abbildung 3.9: Die Einstiegsseite von Grafana.

Abbildung 3.9 zeigt den Einstiegspunkt der Benutzeroberfläche von Grafana. Es ist das Dashboard zu sehen, dieses besteht aus einzelnen Panels, welche in der Regel die Daten aus den angebundenen Datenquellen visualisieren.

In der Kopfzeile (vgl. Abbildung 3.9) befindet sich unter anderem ein Dropdown-Menü, welches die Auswahl eines bestimmten Zeitintervalls für die dargestellten Daten ermöglicht. Dieses Intervall bezieht sich auf die Daten, welche von der jeweiligen Datenbank geladen werden. Das ausgewählte Zeitintervall gilt grundsätzlich für alle Panels des Dashboard, jedoch lassen sich bei Bedarf auch für jedes Panel spezifische Zeitintervalle festlegen.

[6] https://angular.io/ (besucht am 12. Dez. 2019)
[7] https://golang.org/ (besucht am 12. Dez. 2019)

Grafana bietet spezifische Seiten mit Einstellungsmöglichkeiten für Panels, Dashboards und die gesamte Grafana-Instanz. Im Einstellungsmenü eines Dashboards lässt sich unter anderem ein Aktualisierungsintervall für die vorhandenen Zeitreihendaten festlegen. Über die Seitenleiste lässt sich zu den anwendungsweit gültigen Einstellungen wechseln.

Die Benutzeroberfläche von Grafana wurde ursprünglich in Angular entwickelt [Öde18]. Jedoch findet derzeit eine Migration von Angular zu React statt [Öde18]. Für den Umstieg sprach die bessere Zukunftsperspektive, größere Popularität und einfachere Handhabung von React [Öde18]. Die Plugin-Schnittstelle von Grafana ist nun auch für React-Anwendungen verfügbar [Hol19].

Grafana bietet eine eigene Programmbibliothek mit fertigen Komponenten zur beschleunigten Entwicklung von Benutzeroberflächen für Plugins an. Diese Komponenten sind in React geschrieben und können somit einfach in den Programmcode des React-Plugins eingebunden werden.

Grafana bietet derzeit keine Visualisierung an, mit welcher man die gesamte IT-Infrastruktur einer Organisation auf einen Blick überwachen und analysieren könnte. Die in [Chi+19] entwickelte 3D-Visualisierung ist hierzu jedoch in der Lage.

3.2.3 Datenanbindung

Grafana ist in der Lage die Daten zur Visualisierung aus einer Reihe unterschiedlicher Datenquellen zu beziehen. Die offiziell unterstützten Datenquellen umfassen folgenden Systeme [Lab19b]:

- CloudWatch
- Elasticsearch
- Graphite
- InfluxDB
- OpenTSDB
- Prometheus

Darüber hinaus unterstützt Grafana die Anbindung weiterer Datenquellen über spezielle Plugins [Lab19e].

Grafana stellt für jedes DBMS einen individuellen Query-Editor bereit, der auf das jeweilige System zugeschnitten ist und eine Datenexploration in Echtzeit ermöglicht. Darüber hinaus lassen sich eigene Variablen definieren und verschachtelte Datenstrukturen in Zeitreihendaten transformieren. Die benutzerspezifischen Da-

tenbankabfragen werden in einer JSON-Datei des Dashboards gespeichert und auf dem Grafana-Server persistiert [Lab19d]. In dieser Datei werden zusätzlich alle Einstellungs- und Metadaten eines Dashboards und seiner Panels gespeichert.

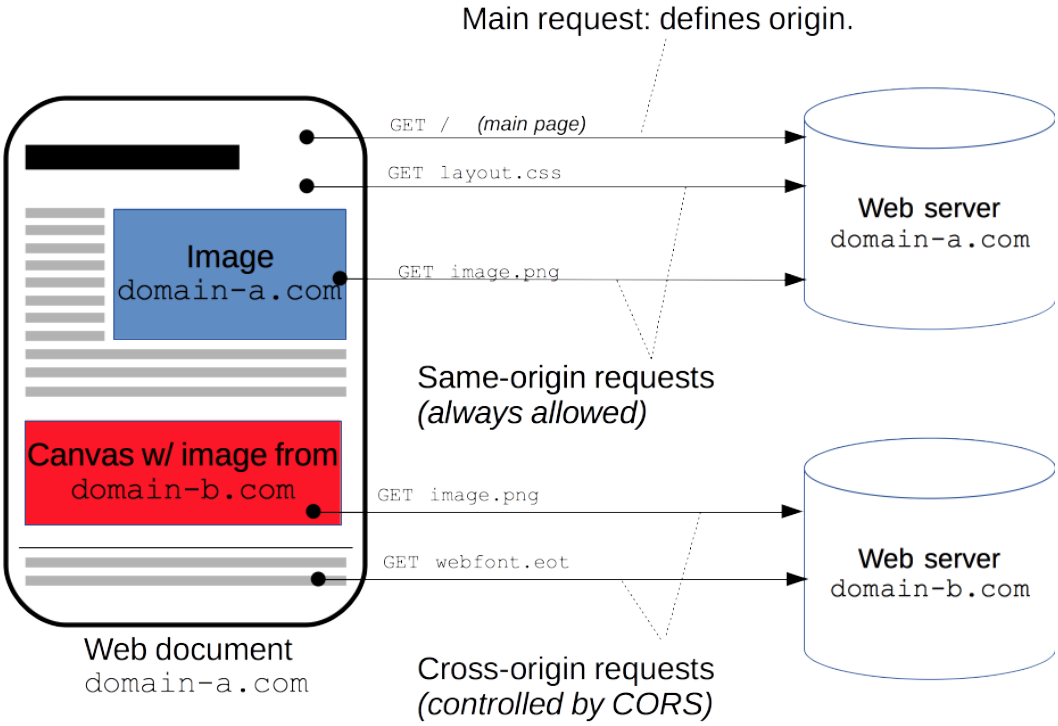

Abbildung 3.10: Die Darstellung illustriert die Funktionsweise von CORS [Cor19].

Die Datenbanken können direkt aus dem Grafana-Client heraus abgefragt werden, sofern dies durch HTTP-Header des auslieferndenden Servers erlaubt ist. Ist dies nicht der Fall, so besteht die Möglichkeit zur Konfiguration des Grafana-Servers als Proxy, welcher die Anfragen an die jeweilige Datenbank weiterleitet und somit als Intermediär zwischen Client und Datenbank dienen kann (vgl. Abbildung 3.8).
Dies kann insbesondere dann hilfreich sein, wenn der Grafana-Server sich hinter einem Reverse-Proxy befindet und dieser kein CORS erlaubt. Wenn jedoch der Grafana-Server als Proxy konfiguriert ist und somit alle Datenbankabfragen über den Grafana-Server geleitet werden, dann liegen nur noch unproblematische *same-origin* HTTP-Anfragen vor.
Bei CORS handelt es sich um einen Sicherheitsmechanismus in Browsern, welcher es Webseiten verbietet Ressourcen von einer anderen Domain als der eigenen zu laden. Abbildung 3.10 versucht die Funktionsweise von CORS anschaulich darzustellen.

3.2.4 Plugin-Schnittstelle

Grafana bietet Entwicklern eine Plugin-Schnittstelle an [Lab19g]. Mit ihrer Hilfe lässt sich der bestehende Funktionsumfang Grafanas für zusätzliche Anwendungsfälle erweitern [Lab19g]. Die Schnittstelle ist in Form verschiedener JavaScript-Bibliotheken

und -klassen entwickelt worden. Damit ist grundsätzlich jede Programmiersprache, welche sich zu JavaScript kompilieren lässt, zur Entwicklung von Grafana-Plugins geeignet [Lab19g].

Grafana unterscheidet zwischen 4 verschiendenen Arten von Plugins: *Backend-Plugins*, *App-Plugins*, *Panel-Plugins* und *Datasource-Plugins* [Lee16].

Datasource-Plugins dienen der Anbindung zusätzlicher Datenbanken an Grafana [Lab19f]. Die bereits von Grafana unterstützten Datenbanken sind ebenfalls mithilfe von Datasource-Plugins umgesetzt worden [Lee16]. Damit nutzen alle angebundenen Datenbanken dieselbe Schnittstelle.

Mithilfe von Panel-Plugins lassen sich weitere Visualisierungen inklusive den korrespondierenden Einstellungsmöglichkeiten zu Grafana hinzufügen [Lab19i].

Backend-Plugins dienen der Erweiterung des Grafana-Servers [Lab19a]. Dies kann unter anderem die Nutzung der Benachrichtigungsfunktion Grafanas umfassen. Des Weiteren ermöglichen Backend-Plugins die Umsetzung eines Caches für Datenbankabfragen, eines Proxies und die Implementierung von benutzerdefinierten Authentifizierungsmethoden [Lab19a].

App-Plugins stellen eine Kombination von Datasource-Plugins und Panel-Plugins dar. Darüber hinaus erlauben Sie die Erstellung benutzerdefinierter Seiten in der Benutzeroberfläche von Grafana [Lab19i]. Hierunter könnten Anmeldeformulare, Dokumentationen oder die Steuerung anderer Services über HTTP-Anfragen fallen [Lab19i].

3.3 Anforderungen an die Integrationsarchitektur

In Abschnitt 1.2 wurde bereits die grobe Zielrichtung dieser Arbeit umrissen. Aus dieser Zielsetzung und der vorangegangen Analyse sollen nun konkrete Anforderungen für die Integrationsarchitektur abgeleitet werden. Dabei sollen auch die im Rahmen der Erprobung des Prototyps gewonnenen Erkenntnisse mit einfließen.

Der Prozess der Anforderungserhebung orientierte sich an den beiden in [PEM03] beschriebenen Techniken *Interviews* und *Prototyping*. Im Rahmen des Prototyping wurde iterativ ein Prototyp entwickelt, anhand dessen die Anforderungen an die Integrationsarchitektur abgeleitet werden konnten. Dabei handelte es sich um einen sogenannten *evolutionären Prototyp*, welcher die Grundlage für die zu entwickelnde Integrationsarchitektur bildete.

Die Interviews fanden in Form von Textnachrichten und einer wöchentlich stattfindenden Telefonkonferenz statt. Hierbei konnten potentielle Anforderungen an das System mit Domänenexperten und den beteiligten Softwareentwicklern diskutiert werden. In diesem Zusammenhang ist zu erwähnen, dass es sich bei den Interviews um *offene Interviews* gehandelt hat. Das heißt, dass im Vorfeld keine Fragen oder ähnliches definiert wurden, stattdessen wurde frei und ohne spezielle Vorgaben diskutiert.

Die im Rahmen des oben beschriebenen Prozesses gewonnenen Anforderungen sind nachfolgend systematisch dokumentiert worden.

[R01] Erweiterbarkeit
Ein wichtiges Ziel ist die Sicherstellung der Erweiterbarkeit der Integrationsarchitektur für ähnlich gelagerte Anwendungsfälle. Hierzu könnte zum Beispiel die zusätzliche Integration in das ITOA-System *Kibana* zählen. Deshalb sollten die Komponenten der Integrationsarchitektur möglichst modular und wo möglich unabhängig von Grafana entwickelt werden. Bestenfalls sollte sich die 3D-Visualisierung einfach über das bestehende Plugin-System von Grafana einbinden lassen.

[R02] Einheitliche Benutzeroberfläche
Die Konfiguration der 3D-Darstellung sollte weitestgehend in die grafische Benutzeroberfläche von Grafana eingebettet und integriert werden, um eine einfach Bedienung für die Anwender sicherzustellen.

[R03] Einführung eines Persistenzmechanismus
Darüber hinaus sollten die vom Nutzer vorgenommenen Einstellungen über eine Sitzung hinweg verfügbar sein. Hierfür ist die Einbindung eines Persistenzmechanismus in Form einer Konfigurationsdatei oder einer Datenbank von Nöten.

[R04] Stabilität gegenüber Updates
Die Integrationsarchitektur sollte möglichst stabil gegenüber Updates oder Upgrades einzelner beteiligter Komponenten sein. Deshalb sollte die Anbindung an Grafana nicht zu eng sein. Dies ist insbesondere bei Grafana wichtig, da einige Schnittstellen, welche Grafana Entwicklern zur Verfügung stellt, aktuell noch einen Alpha-Status[8] haben. Das heißt, dass sogenannte *Breaking Changes* möglich sind. Aus diesem Grund sollte die Integration in Grafana nur über definierte Schnittstellen stattfinden. Modifikationen im Quellcode von Grafana sollten dagegen vermieden werden.

[R05] Flexible Datenanbindung
Des Weiteren sollten sich zusätzliche Datenquellen mit wenig Aufwand an die Integra-

[8] Ein früher Stand im Entwicklungsstadium einer Software.

tionsarchitektur anbinden lassen. In diesem Zusammenhang ist auch sicherzustellen, dass sich Zeitreihendaten mit beliebiger Datenstruktur in die Architektur einbinden lassen. Um dieses Ziel zu erreichen, muss eine flexible Datenanbindung in die Applikation eingebaut werden. Derzeit unterstützt der Prototyp nur die Solr-Plattform als Datenquelle. Für die Zukunft ist jedoch die Anbindung zusätzlicher Datenquellen wie zum Beispiel Prometheus[9], CSV-Dateien und Elasticsearch[10] geplant.

[R06] Dynamischer Wechsel von Kennzahlen und Aggregationstypen
Die fertige Integrationsarchitektur sollte den dynamischen Wechsel zwischen benutzerdefinierten Kennzahlen über die Benutzeroberfläche erlauben. Außerdem sollte bei der Auswahl eines Zeitraums über die 2D-Visualisierung automatisch eine Aggregation über diesen Zeitraum durchgeführt und in der 3D-Visualisierung dargestellt werden. Darüber hinaus sollte sich der Typ der Aggregation dynamisch über die Benutzeroberfläche festlegen lassen.

[R07] Funktionsumfang
Ferner sollte die bestehende Funktionalität des Prototyps vollständig von der Integrationsarchitektur abgebildet werden. Hierbei ist auch zu prüfen, ob redundante Funktionen, also Funktionen, welche sowohl im Prototyp als auch in Grafana vorhanden sind - wie zum Beispiel die Zeitauswahl in einer Komponente - konsolidiert werden können.

[R08] Performanz
Ein darüber hinaus wichtiger Punkt ist die Sicherstellung einer Performanz, die den Ansprüchen der Endanwender gerecht wird. Das heißt, dass nach dem initialen Laden der Daten keine spürbaren Verzögerungen bei der Nutzung der Anwendung auftreten sollten. Dies stellt insbesondere bei großen Datenmengen noch ein Kernproblem des Prototyps dar. Denn aktuell werden alle Zeitreihendaten bei der Initialisierung des Programms in den Client (Browser) des Anwenders geladen. Jedoch ist die Leistung der dort verfügbaren Hardware oftmals begrenzt. In der fertigen Integrationsarchitektur sollen deshalb immer nur die Zeitreihendaten in den Client geladen werden, welche für das aktuell ausgewählte Zeitintervall von Relevanz sind.

[R09] Bereitstellung
Die fertige Anwendung sollte in die Build-Pipeline integriert werden. Es sollte darauf geachtet werden, dass so wenig Code wie möglich kopiert wird. Stattdessen sollen die in der Integrationsarchitektur benötigten Module aus der Codebasis des Prototyps importiert werden.
Außerdem soll eine kurze Anleitung die Installation der fertigen Integrationsarchitektur für Anwender erleichtern.

[9] https://prometheus.io/ (besucht am 17. Dez. 2019)
[10] https://www.elastic.co/products/elasticsearch (besucht am 17. Dez. 2019)

4 Implementierung

Das folgende Kapitel beschäftigt sich mit der konkreten Implementierung der Integrationsarchitektur. Hierzu sollen beispielhaft die wichtigsten Komponenten und Funktionen der Integrationsarchitektur aufgezeigt werden. Zusätzlich sollen die während der Implementierung getroffenen Designentscheidungen und aufgetretenen Probleme aufgezeigt und erläutert werden.

In einem ersten Schritt musste die für das Integrationsvorhaben passende Integrationsart ausgewählt werden (vgl. Abschnitt 2.2). Für die vorliegende Integrationsarchitektur kam nur die Punkt-zu-Punkt-Architektur in Frage, da sowohl die Hub-and-Spoke-Architektur, als auch die serviceorientierten Architekturen hinsichtlich ihrer Umsetzung sehr komplex und damit zeitaufwendig sind. Bei der Punkt-zu-Punkt-Architektur kann hingegen die bestehende Plugin-Schnittstelle Grafanas genutzt werden, ohne einen zusätzlichen Webservice, oder eine extra Schnittstelle in Grafana für einen Service-Hub entwickeln zu müssen.

Im Anschluss an die Auswahl der Integrationsart erfolgte die Auswahl einer sinnvollen Integrationsebene.
Eine Integration auf der Datenebene macht hier keinen Sinn, da Grafana - abgesehen von einer Datenbank für Konfigurationen - über keine dedizierte Datenbank mit festem Schema oder Datensätzen verfügt. Die Daten werden über eine Schnittstelle an Grafana angebunden und liegen auf externen anwenderspezifischen Datenbankservern, welche keinem festen Schema folgen. Außerdem verfügt Grafana aktuell über keine Visualisierung, welche der in [Chi+19] entwickelten 3D-Visualisierung ähnelt, jedoch ist das vorliegende Integrationsvorhaben durch die Bereitstellung einer solchen Visualisierung ursprünglich motiviert worden. Deshalb muss eine Integration auf höherer Ebene durchgeführt werden, damit die neu entwickelte 3D-Visualisierung in der fertigen Integrationsarchitektur zur Verfügung steht.

Die zweite Option besteht in einer Integration ausschließlich über die Benutzeroberfläche. Dies ließe sich theoretisch durch einen *Inlineframe*[1] realisieren. Dabei müsste der gesamte Prototyp in den Inlineframe eingefügt werden, damit weiterhin die volle Funktionalität der 3D-Visualisierung zur Verfügung steht.
Das oben erläuterte Vorgehen bringt jedoch einige Nachteile mit sich. Einerseits

[1] Bei einem Inlineframe handelt es sich um ein spezielles HTML-Element, welches die Einbettung eines HTML-Dokuments in ein anderes HTML-Dokument ermöglicht [W3C19a].

würde mit einer solchen Integration eine Verschlechterung der Benutzerfreundlichkeit einhergehen, da zwei vollkommen unterschiedliche Benutzeroberflächen nebeneinander laufen müssten. Andererseits ist eine Konsolidierung redundanter Bedienelemente dann ebenfalls nicht möglich. Ein weiterer Negativpunkt stellt die begrenzte Positionierbarkeit des Inlineframes dar, denn bei Grafana handelt es sich um keine klassische statische HTML-Seite, sondern um eine Web-Applikation, welche größtenteils nur als JavaScript-Programmcode vorliegt. Eine Positionierung des Inlineframes in der Benutzeroberfläche von Grafana ist somit nicht ohne Weiteres möglich. Es ließe sich zwar theoretisch über die Plugin-Schnittstelle ein einfaches Plugin entwickeln, dessen einziger Zweck die Ausgabe des Inlineframes an einer bestimmten Stelle in der Benutzeroberfläche Grafanas wäre, jedoch würde es sich dabei streng genommen um eine Integration auf Applikationslogikebene handeln.

Schließlich gilt es noch die Integration auf Applikationslogikebene hinsichtlich der Eignung für das vorliegende Integrationsvorhaben zu prüfen. Eine solche Integration bietet sich im vorliegenden Fall an, da Grafana bereits eine hierfür notwendige dedizierte Schnittstelle für die Erweiterung der Anwendung zur Verfügung stellt.
Das oben erläuterte Vorgehen hat einige entscheidende Vorteile für die Integrationsarchitektur. Zum einen besteht bei einer Integration auf der Applikationslogikebene die Möglichkeit dazu redundante Komponente zu konsolidieren. Zum anderen ist es möglich viele Elemente der Benutzeroberfläche des Prototyps sowohl grafisch, mithilfe von Grafana zur Verfügung gestellten Bibliotheken, als auch funktional mithilfe der Plugin-Schnittstelle zu integrieren und somit eine einheitliche und konsistente Benutzeroberfläche über alle Elemente der integrierten Anwendung hinweg zu erreichen. Zu diesem Zweck stellt Grafana umfangreiche Programmbibliotheken zur Verfügung. Diese beinhalten die Plugin-Schnittstelle für die Integration der Applikationslogik, aber auch eine Bibliothek, welche fast alle Elemente der Benutzeroberfläche von Grafana in Form von konfigurierbaren React-Komponenten zur Verfügung stellt.
Aufgrund der oben dargelegten Vor- und Nachteile der einzelnen Integrationsebenen, fiel die Wahl auf eine Integration auf der Applikationslogikebene.

Nachdem sich für die Integration auf Applikationslogikebene entschieden wurde, gilt es die potentiellen Möglichkeiten der Anbindung des Prototyps an Grafana auf dieser Ebene zu prüfen.
Aufgrund der Quelloffenheit von Grafana besteht die Möglichkeit der Integration des Prototyps in Grafana durch Modifikation des Programmcodes. Dieses Vorgehen hätte jedoch zwei entscheidende Nachteile. Einerseits wäre der Zeitaufwand für das Finden der zu modifizierenden Stellen im Quellcode erheblich. Andererseits müsste bei einem Update von Grafana der Programmcode aufwendig analysiert und getestet werden, um die korrekte Funktionsweise der Modifikation zu gewährleisten. Weiterhin würde ein solches Vorgehen die Anforderung [R04] aus Abschnitt 3.3

verletzen.
Eine weitere Möglichkeit der Integration besteht in der Nutzung der offiziellen Plugin-Schnittstelle von Grafana. Die Nutzung einer zur Erweiterung der Funktionalität vorgesehenen Schnittstelle bringt in der Regel den wesentlichen Vorteile der Stabilität gegenüber potentieller Updates mit sich. Im vorliegenden Fall befindet sich die Schnittstelle jedoch noch im Alpha-Status, deshalb können Updates publiziert werden, welche Änderungen an der Schnittstelle mit sich bringen. Nichtsdestotrotz wurde sich im vorliegenden Fall für die Nutzung der Plugin-Schnittstelle entschieden, da die durch ein Update verursachten Änderungen an der Schnittstelle in der Regel im sogenannten Changelog dokumentiert werden und damit einfach aufzufinden sind.

Über die Plugin-Schnittstelle von Grafana lassen sich 4 verschiedene Arten von Plugins realisieren (vgl. Abschnitt 3.2.4). Deshalb soll nachfolgend eine für das Integrationsvorhaben geeignete Pluginart ausgewählt werden.
Im vorliegenden Fall fiel die Entscheidung auf das Panel-Plugin, da es eine Erweiterung Grafanas um zusätzliche Visualisierungen erlaubt [Lab19i].
Ein App-Plugin ließe sich hierzu ebenso verwenden, jedoch wurde im vorliegenden Fall die darin enthaltene Funktionalität zur Datenanbindung über das Data Source-Plugin nicht benötigt. Aus dem gleichen Grund wird auch kein Backend-Plugin in der Integrationsarchitektur verwendet.

Bevor mit der Implementierung des Plugins begonnen werden konnte, musste noch eine geeignete Programmiersprache ausgewählt werden.
Ein Grafana-Plugin kann theoretisch in jeder Programmiersprache geschrieben werden, welche sich zu JavaScript kompilieren lässt [Lab19g]. In diesem Fall wurde die Programmiersprache TypeScript in Verbindung mit dem JavaScript-Framework React ausgewählt.
Diese Entscheidung fußt einerseits auf der Tatsache, dass der Prototyp, welcher das Fundament der Integrationsarchitektur bilden soll, das JavaScript-Framework React und die Programmiersprache TypeScript verwendet. Damit bietet sich eine Nutzung von TypeScript in Verbindung mit React im Rahmen der Umsetzung der Integrationsarchitektur an. Zudem liegt die Vermutung nahe, dass sich die Integration zweier auf React-basierender Anwendungen einfacher gestalten sollte, als wenn es sich um zwei unterschiedliche Technologien handeln würde.
Andererseits sind die zur Plugin-Schnittstelle von Grafana gehörenden Komponenten und Bibliotheken in TypeScript und teilweise unter Verwendung von React implementiert worden, wodurch sich die Verwendung dieser Technologien in der Integrationsarchitektur anbietet.

Nachdem die vorbereitenden Tätigkeiten abgeschlossen wurden, konnte mit der Implementierung der Integrationsarchitektur begonnen werden. Die Basis hierfür bildete der Programmcode des evolutionären Prototypen. Auf dessen Grundgerüst die Integrationsarchitektur aufbaut. Dabei zeigt sich der Vorteil des modularen Aufbaus des Prototyps, welcher die einzelnen Funktionen und Bereiche der Anwendung in React-Komponenten und Services aufteilte. Diese Komponenten und die dazugehörigen Services können nun bei Bedarf einfach importiert werden. Somit kann eine Wiederverwendung bestehender Module sichergestellt werden, was wiederum die Wartbarkeit des zu entwickelnden Plugins erheblich erleichtert.

4.1 Architektur

Das nachfolgende Unterkapitel soll einen Überblick über den Aufbau, die Struktur und die einzelnen Komponenten der Integrationsarchitektur bieten. Außerdem sollen elementare Designentscheidungen, welche zur vorliegenden Architektur geführt haben, erörtert werden.

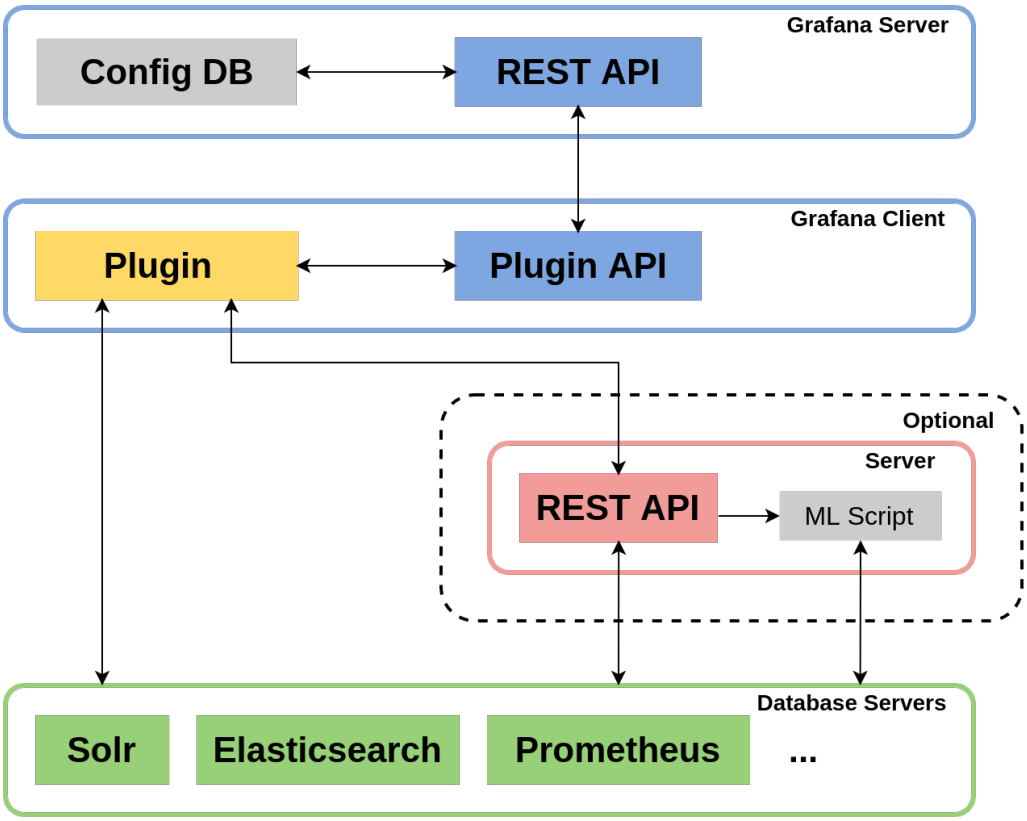

Abbildung 4.1: Eine übersichtliche Darstellung der einzelnen Komponenten der fertigen Integrationsarchitektur.

Abbildung 4.1 bietet einen Überblick über die Komponenten und Kommunikationskanäle der Integrationsarchitektur. Die Abbildung beschränkt sich auf die Darstellung

der für die Integration relevanten Aspekte. Nachfolgend sollen die einzelnen Elemente der Architektur und ihre jeweilige technologische Ausprägung erläutert werden.

Der Grafana-Server stellt eine REST-Schnittstelle bereit, welche unter anderem der Persistierung von Einstellungsdaten in einer Datenbank namens *Config DB* dient (vgl. Abbildung 4.1). Die REST-Schnittstelle auf dem Grafana-Server (vgl. Abbildung 4.1) wurde, analog zum Grafana-Server selbst, in der Programmiersprache Go entwickelt (siehe Abschnitt 3.2.1). Bei der Datenbank handelt es sich um das relationale Datenbanksystem SQLite[2]. Diese dient zur Persistierung der im Plugin vorgenommenen benutzerdefinierten Einstellungen und befindet sich analog zur REST-Schnittstelle ebenfalls auf dem Grafana-Server.
Damit die im Plugin vorgenommen Einstellungen persistiert werden können, müssen diese in einem Objekt gespeichert werden, welches wiederum bei der Plugin-Schnittstelle des Grafana-Clients registriert werden muss. Bei jeder Änderung der Konfigurationen werden diese über die Plugin-Schnittstelle im Grafana-Client an die REST-Schnittstelle auf dem Grafana-Server gesendet. Im Anschluss speichert die REST-Schnittstelle die Konfigurationsdaten in der dafür vorgesehenen Konfigurationsdatenbank (vgl. Config DB in Abbildung 4.1).
Beim Neustart der Anwendung werden initial die gespeicherten Einstellungen aus der Konfigurationsdatenbank geladen. Infolgedessen stehen dem Anwender die benutzerdefinierten Einstellungen in der fertigen Integrationsarchitektur über eine Browser-Session hinaus zur Verfügung.
Die Realisierung des Persistenzmechanismus für Einstellungsdaten ließe sich auch über die REST-Schnittstelle des Servers (vgl. Abbildung 4.1) bewerkstelligen. Dies hätte jedoch zufolge, dass die Verfügbarkeit der gespeicherten Einstellungen von der des Servers abhängig wäre. Die vorliegende Architektur (vgl. Abbildung 4.1), welche die Einstellungen in der bereits auf dem Grafana-Server vorhandenen Konfigurationsdatenbank speichert, umgeht diese Beschränkung und ist deshalb vorzuziehen.
Ein weiterer Vorteil der in Abbildung 4.1 gezeigten Architektur, ist die Nutzung bereits vorhandener Grafana-Komponenten in Form der Konfigurationsdatenbank und der für das Speichern und Laden zuständigen Programmlogik in der REST-Schnittstelle auf dem Grafana-Server. Hierdurch ergab sich nicht nur eine Reduktion des Aufwands bei der Entwicklung der Integrationsarchitektur, sondern auch eine Verringerung der Komplexität der Architektur durch weniger selbst entwickelte Komponenten.
Die oben genannten Argumente helfen bei der Entscheidung für die Speicherung der Einstellungsdaten in der bereits vorhandenen SQLite-Datenbank. Eine ausführliche Beschreibung der konkreten Implementierung des dargestellten Persistenzmechanismus folgt in Abschnitt 4.3.3.

[2]https://www.sqlite.org/index.html (besucht am 04. Jan. 2020)

Der *Server* in Abbildung 4.1 stellt ähnlich zum Grafana-Server ebenfalls eine REST-Schnittstelle zur Verfügung und konnte aus dem Prototyp (siehe Abschnitt 3.1.4) übernommen werden. Der Unterschied zur REST-Schnittstelle auf dem Grafana-Server manifestiert sich hierbei unter anderem in der verwendeten Programmiersprache, welche im vorliegenden Fall node.js ist.
Die REST-Schnittstelle auf dem Server (vgl. Abbildung 4.1) ermöglicht dem Plugin die Steuerung der Vorhersagefunktionalität der Integrationsarchitektur. Das grundlegende Zusammenspiel zwischen der REST-Schnittstelle und dem Skript (vgl. *ML Script* in Abbildung 4.1), welches den für die Vorhersage notwendigen Machine-Learning-Algorithmus enthält, hat sich im Vergleich zu der im Abschnitt 3.1.4 beschriebenen Art und Weise nicht geändert.
Ferner kann die REST-Schnittstelle des Servers bei Bedarf als Proxy im Rahmen der Datenanbindung genutzt werden. Zwar erlaubt der Grafana-Server über seine HTTP-Header CORS-Anfragen, jedoch kann es vorkommen, dass sich der Grafana-Server hinter einem Reverse-Proxy befindet, der die HTTP-Header des Grafana-Servers modifiziert, sodass sie keine CORS-Anfragen aus dem Grafana-Client heraus erlauben. Der indirekte Übertragungsweg dient der Umgehung des CORS-Sicherheitsmechanismus und wurde im Rahmen des Aufbaus der Integrationsarchitektur neu entwickelt. Hierbei besteht jedoch eine Einschränkung in Hinsicht auf die Bereitstellung des Servers, auf dem der Proxy läuft. Dieser muss über dieselbe Domain wie die des Grafana-Servers bereitgestellt werden, sodass die Kommunikation mit dem Proxy ohne die Auslösung des CORS-Sicherheitsmechanismus funktionieren kann. Das liegt an dem im Grafana-Client befindlichen Plugin, welches über den Grafana-Server ausgeliefert wird und damit nur HTTP-Anfragen an die Domain des Grafana-Server stellen darf.
Eine alternative zur Nutzung eines Proxies besteht in der Anpassung der HTTP-Header des jeweils genutzten Reverse-Proxy, welcher die HTTP-Header des Grafana-Servers, der unter anderem für die Auslieferung des Grafana-Clients zuständig ist, anpasst. Damit können explizit CORS-Anfragen für bestimmte Server erlaubt werden. Erfahrungsgemäß ist der hierfür notwendige administrative Zugriff auf die betreffenden Server, insbesondere in großen Organisationen, oft nur sehr restriktiv möglich, was die Anpassung der HTTP-Header in der Praxis erschwert. Deshalb wurde die oben beschriebene Proxy-Funktionalität für die Integrationsarchitektur entwickelt. Weiterhin ist die Integrationsarchitektur auch ohne den Server und die darauf befindlichen Funktionalitäten lauffähig, soweit die oben dargelegte Proxy-Funktionalität nicht benötigt wird. Diese Eigenschaft wurde aus dem Prototyp übernommen, da sie eine leichtere Bereitstellung für Anwender, welche nur die 3D-Visualisierung ohne die Vorhersage benötigen, ermöglicht.

Die Entscheidung, den Server des Prototyps anstatt den Grafana-Server zur Bereitstellung der Vorhersagefunktionalität zu nutzen, wurde durch mehrere Gründe

motiviert, welche im Folgenden erörtert werden sollen.
Ein Grund der Nutzung der REST-Schnittstelle auf dem Server, gegenüber der auf dem Grafana-Server, liegt in der unbeschränkten Erweiterbarkeit der REST-Schnittstelle auf dem Server. Diese Eigenschaft ist notwendig, um die notwendigen Endpunkte und Funktionen für die Vorhersage und den Proxy in die REST-Schnittstelle implementieren zu können. Die REST-Schnittstelle des Grafana-Server verfügt zwar auch über eine Schnittstelle zur Erweiterung, jedoch lassen sich hierüber nur die in einem Interface vordefinierten Methoden für Data-Source-Plugins implementieren. Dies verhindert die Erweiterung der REST-Schnittstelle Grafanas für Anwendungsfälle wie den Vorliegenden. Damit bliebe nur noch die Möglichkeit der direkten Modifikation des Programmcodes, welcher die REST-Schnittstelle auf dem Grafana-Server implementiert. Ein solches Vorgehen würde jedoch die Wartbarkeit und Stabilität der Integrationsarchitektur verringern. Außerdem würde ein Modifikation die Anforderung [R04] aus Abschnitt 3.3 verletzten.

Die Plugin-Schnittstelle befindet sich im Grafana-Client (vgl. Abbildung 4.1) und ermöglicht die Einbindung von Plugins in Grafana. Im vorliegenden Fall handelt es sich bei dem Plugin um ein Panel-Plugin. Das Panel-Plugin ist eine klassische React-Anwendung, welche verschiedene Bibliotheken und Komponenten zur Erweiterung Grafanas nutzt. Die Basis des Plugins bildet der Prototyp aus Abschnitt 3.1. Zur Entwicklung wurden Bibliotheken und Schnittstellen von Grafana genutzt. Das Plugin muss in einen konfigurierbaren Pfad zusammen mit einer Datei JSON-Datei namens `plugin.json` platziert sein, damit Grafana die React-Anwendung als Plugin erkennt und korrekt einbinden kann.

Die Anbindung der Datenbankserver an das Plugin erfolgt wahlweise direkt aus dem Plugin heraus oder indirekt über die REST-Schnittstelle des Servers (vgl. Abbildung 4.1). Dabei wurde sich gegen eine Anbindung über die Datenbankschnittstelle von Grafana entschieden.
Ein Grund hierfür ist die Plugin-Schnittstelle von Grafana, welche React-Plugins keine Möglichkeit bietet, benutzerdefinierte Datenbankabfragen über die Datenbankschnittstelle zu senden. Die Anwesenheit einer solchen Funktion wäre jedoch von Nöten, damit das Plugin alle für die Visualisierung benötigten Daten von der jeweiligen Datenbank abfragen kann. Hierzu müssen viele verschiedene Datenbankfragen an die Datenbank gesendet werden. Darunter fallen unter anderem Datenbankabfragen, welche die Aggregation über bestimmte Zeiträume betreffen. Im Prototyp wurden solche Aggregierungen noch im Client durchgeführt, jedoch wurde sich im Rahmen des Aufbaus der Integrationsarchitektur dafür entschieden, so viele Berechnungen wie möglich auf die betreffende Datenbank auszulagern. Diese Entscheidung war nötig, um eine adäquate Performanz des Plugins sicherzustellen. Aufgrund der Tatsache, dass die jeweils anzubindenden Datenbanksysteme über verschiedene Funktionsumfänge verfügen, wird je nach Datenbanksystem ein Teil der Operationen

und Berechnungen weiterhin im Client durchgeführt.
Darüber hinaus konnten mit der Wahl der bestehenden Datenbankanbindung viele Funktionen aus dem Prototyp wiederverwendet werden. Dagegen hätte die Anbindung über die Datenbankschnittstelle Grafanas, sofern diese aus dem Plugin ansprechbar gewesen wäre, eine umfangreiche Anpassung der bestehenden Programmlogik für die Datenanbindung bedeutet.
Schließlich ist noch die verbesserte Portabilität der Integrationsarchitektur durch die nicht vorhandene Kopplung an die Datenbankschnittstelle Grafanas hervorzuheben. Aus diesem Grund lässt sich die Integrationsarchitektur deutlich einfacher für ähnlich gelagerte Anwendungsszenarien mit kleineren Anpassungen wiederverwenden. Damit wird auch die Anforderung [R01] aus Abschnitt 3.3, die eine solche Erweiterungsmöglichkeit für die fertige Integrationsarchitektur explizit fordert, umgesetzt.

4.2 Datenanbindung

In diesem Abschnitt soll die Umsetzung der Datenanbindung der Integrationsarchitektur thematisiert werden. Hierbei sollen die daran beteiligten Komponenten und deren Zusammenspiel untereinander erklärt werden. Außerdem sollen die Änderungen im Vergleich zur Datenanbindung des Prototyps (vgl. Abschnitt 3.1.3) verdeutlicht werden.

In einem ersten Schritt musste die Anforderung eines flexiblen Wechsels zwischen verschiedenen Kennzahlen in der fertigen Integrationsarchitektur implementiert werden (vgl. Anforderung [R06] in Abschnitt 3.3). Der Prototyp, welcher das Fundament der Integrationsarchitektur bildet, verfügte bis dato über keine solche Funktionalität. Bisher konnten mithilfe der 3D-Visualisierung des Prototyps nur Auswertungen nach einer Kennzahl vorgenommen werden. Dabei handelte es sich um die Anzahl der gemessenen Zugriffe auf einen Server zu einem bestimmten Zeitpunkt. Diese Kennzahl war in verschiedenen Modulen des Prototyps fest codiert worden. Dieser Umstand sollte nun im Rahmen der Entwicklung des Plugins für die Integrationsarchitektur geändert werden.
Hierzu wurde eine neue Zustandsvariable namens `selectedMeasure` eingeführt, die zur Laufzeit die aktuell ausgewählte Kennzahl zentral im Plugin speichert. Dabei wird die Variable gemeinsam mit den anderen verfügbaren Einstellungen des Plugins in einem Objekt gespeichert, welches durch die Plugin-Schnittstelle automatisch in der Konfigurationsdatenbank Grafanas (vgl. *Config DB* in Abbildung 4.1) persistiert wird. Der dynamische Wechsel zwischen den verfügbaren Kennzahlen kann vom Anwender über das Einstellungsmenü des Panel-Plugins vollzogen werden. Ein Wechsel der Kennzahl löst automatisch ein Neuladen der jeweils korrespondierenden Zeitrei-

hendaten vom Datenbankserver aus. Sobald die neuen Daten im Plugin verfügbar sind werden die 3D-Visualisierung und die dazugehörige 2D-Visualisierung mit den aktualisierten Daten der neu ausgewählten Kennzahl neugeladen. Die zur Auswahl stehenden Kennzahlen können sich von Organisation zu Organisation unterscheiden, deshalb können sie mittels eines Adapters vom Anwender festgelegt werden (siehe nachfolgender Abschnitt).
Die Realisierung der Möglichkeit, des dynamischen Wechsels zwischen den verschiedenen Kennzahlen über das Einstellungsmenü in der Benutzeroberfläche des Plugins wird in Abschnitt 4.3.3 thematisiert.

Die bisherige Architektur des Prototyps erlaubte keine Variabilität hinsichtlich der Struktur der von der jeweiligen Datenbank kommenden Rohdaten. Das Funktionieren des Prototyps war maßgeblich von der Struktur der Rohdaten abhängig. Dieser Umstand sollte im Rahmen des Aufbaus der Integrationsarchitektur geändert werden, sodass die Anforderung der Unterstützung verschieden strukturierter Rohdaten (vgl. Anforderung [R06] in Abschnitt 3.3) von der fertigen Integrationsarchitektur umgesetzt wird.

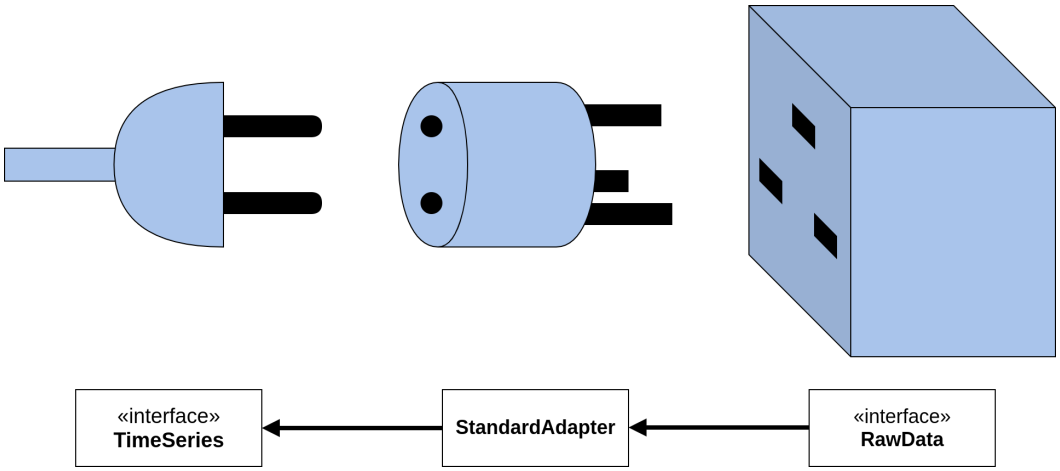

Abbildung 4.2: Die Abbildung illustriert die Funktionsweise des *Adapter Pattern* anhand einer Analogie zu Steckdosenadaptern. Weiterhin wird ein Bezug zu der vorliegenden Implementierung aufgezeigt. Die Pfeilrichtung zeigt die Richtung des Datenflusses an.

Hierzu sollte das sogenannte *Adapter Pattern* eingesetzt werden. Dabei handelt es sich um ein Entwurfsmuster für die Softwareentwicklung, welches die Zusammenarbeit zwischen zwei unterschiedlichen Interfaces mittels einer Konvertierung ermöglicht [Gam+95]. Abbildung 4.2 illustriert die Funktionsweise des Entwurfsmusters anhand einer Analogie zu einem Steckdosenadapter. Dabei repräsentiert das Interface `RawData` die Struktur der jeweiligen Rohdaten. Der `StandardAdapter` transformiert die Rohdaten in eine Datenstruktur, welche dem Interface `TimeSeries` (siehe Abbildung 3.7) entspricht. Diese Interface bildet die Grundlage der 3D-Visualisierung.
Der Prototyp verfügte zwar bereits über einen Adapter in Form des `SolrAdapter`, je-

doch unterstützte dieser nicht den dynamischen Wechsel zwischen unterschiedlichen Kennzahlen. Außerdem befand sich ein Teil der Zugriffslogik und der notwendigen Transformationslogik außerhalb des Adapters, über den Porgrammcode der Anwendung verteilt, was eine flexible Anbindung verschiedener Datenstrukturen an die Anwendung unmöglich machte. Deshalb wurde ein neuer Adapter mit dem Namen `StandardAdapter` entwickelt. Dieser beinhaltet die gesamte Transformationslogik und ermöglicht sowohl die Anbindung von Rohdaten unterschiedlicher Struktur, als auch den dynamischen Wechsel zwischen den verfügbaren Kennzahlen.
In diesem Zusammenhang wurde die bestehende Transformationslogik aus dem bestehenden `SolrAdapter` in den neuen `Standardadapter` übernommen und leicht angepasst. Zusätzlich wurde die Transformationslogik, welche sich bisher noch nicht innerhalb des `SolrAdapters` befand, in den `StandardAdapter` überführt. Außerdem wurde der direkte Zugriff auf Kennzahlen aus den Rohdaten entfernt und dynamisch gestaltet, sodass ein Wechsel zwischen den verfügbaren Kennzahlen einer Datenstruktur möglich ist. Ein Beispiel hierfür ist die Einführung der Variable `selectedMeasure`, welche die aktuell ausgewählte Kennzahl enthält und die dynamische Selektion einer Kennzahl erlaubt, für welche die Transformation im Adapter durchgeführt werden soll. Die dabei zur Auswahl stehenden Kennzahlen lassen sich nun ebenfalls in dem neuen Adapter definieren, sodass die gesamte Transformations- und Zugriffslogik im `StandarAdapter` enthalten ist. Damit lassen sich Rohdaten beliebiger Gestalt in die Integrationsarchitektur einbinden. Hierzu muss lediglich ein spezifischer Adapter für die anzubindende Datenstruktur implementiert werden. Dabei kann der `StandardAdapter`, beziehungsweise dessen Interface als Blaupause dienen.

Der Prototyp unterstützte bisher nur das Datenbankmanagementsystem (DBMS) Solr als Datenquelle. Dies sollte sich mit der fertigen Integrationsarchitektur ändern (vgl. Anforderung [R05] in Abschnitt 3.3). In einem ersten Schritt musste deshalb entschieden werden, ob für die Anbindung zusätzlicher Datenquellen neben Solr, jeweils ein spezielles Datasource-Plugin für Grafana entwickelt werden sollte, oder ob die bestehende Datenanbindung des Prototyps so umgestaltet werden sollte, dass die flexible Erweiterung um neue Datenquellen und der dynamische Wechsel zwischen den Datenquellen aus der Benutzeroberfläche heraus ermöglicht wird.
Im vorliegenden Fall wurde sich für die zweite Variante entschieden, da sich nur mit dieser Variante die Anforderung der Sicherstellung der Erweiterbarkeit der Integrationsarchitektur umsetzen lässt (vgl. Anforderung [R01] in Abschnitt 3.3). Die erste Variante würde zwar einen hohen Grad der Integration des Prototyps in Grafana erlauben, jedoch würde darunter die Erweiterbarkeit der Integrationsarchitektur für ähnliche Anwendungsfälle wie zum Beispiel die Integration in Kibana leiden. Ein weiterer Nachteil der ersten Variante ist die mangelnde Unterstützung von benutzerdefinierten Datenbankabfragen über die Plugin-Schnittstelle für React, somit wären viele Funktionen wie zum Beispiel die zeitintervallbasierte Selektion von Daten bei

dieser Variante nicht möglich.

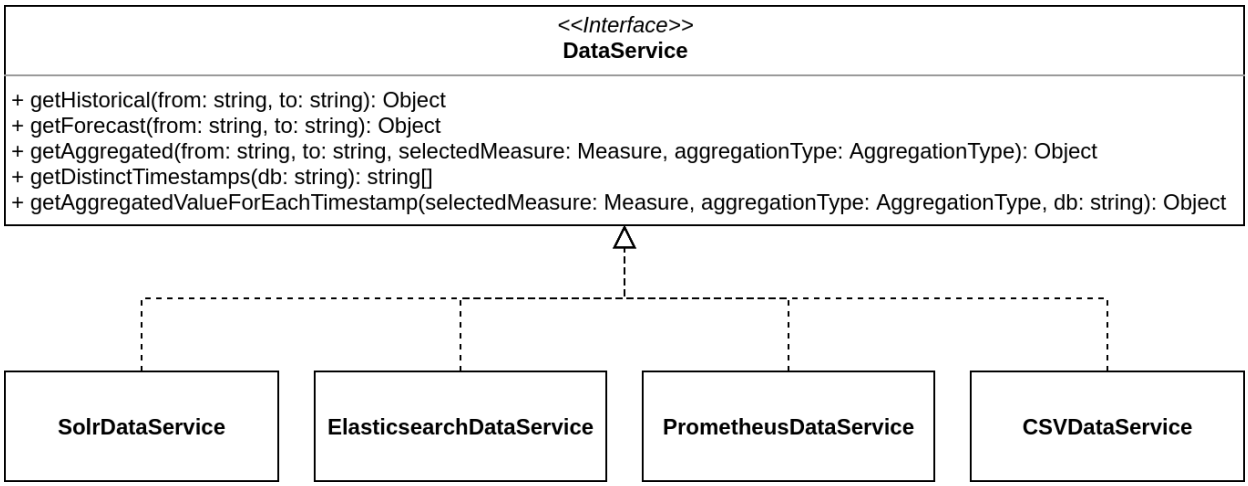

Abbildung 4.3: Das Interface `DataService` dient als gemeinsamer Kontrakt für die Services der verfügbaren Datenquellen.

Bisher war die Service-Klasse `DataService` für die Anbindung der Rohdaten aus Solr verantwortlich (vgl. Abbildung 3.4). Im Zuge der Umsetzung der neuen Anforderung sollte dem Anwender der dynamische Wechsel zwischen den verschiedenen Datenquellen ermöglicht werden.
Hierzu wurde für jede verfügbare Datenquelle ein eigener Service zur Datenanbindung entwickelt, da die einzelnen Datenquellen sich hinsichtlich des Funktionsumfangs der jeweiligen Abfragesprache erheblich unterscheiden können. Einige DBMS verfügen zum Beispiel über eine umfangreiche Anfragesprache, welche komplexe Transformationen und Aggregierungen vornehmen kann, währenddessen andere Systeme nur über vergleichsweise primitive Abfragemechanismen verfügen. Diesem Umstand wurde durch die Entwicklung eines neuen Interfaces mit dem Namen `DataService` (vgl. Abbildung 4.3) Rechnung getragen. Dieses Interface stellt sicher, dass jede Service-Klasse die Methoden des Interface implementiert, somit können die konkreten Implementierungen einer Methode an den Funktionsumfang der Abfragesprache der jeweiligen Datenquelle angepasst werden. Dieser Mechanismus führt zu einer Verbesserung der Performanz der Anwendung, da sich in einigen Fällen rechenintensive Operationen auf den jeweiligen Datenbankserver auslagern lassen, sodass die, in der Regel begrenzten, Ressourcen des Clients nicht unnötig belastet werden.
Ein Beispiel hierfür ist die Methode `getMaxValue` (vgl. Abbildung 4.3), die den maximalen Wert einer bestimmten Kennzahl in den betreffenden Zeitreihendaten bestimmt. Bei Solr könnte eine solche Bestimmung über die REST-Schnittstelle des Solr-Datenbankservers durchgeführt werden und das Ergebnis müsste nur noch in der sich im Client befindlichen Service-Methode `getMaxValue` entgegengenommen werden. Dagegen müsste die Methode bei einer unsortierten CSV-Datei die Untersuchung aller Elemente der betreffenden Zeitreihe im Client selbst vornehmen, da

in einem solchen Fall keine Auslagerung der Operation auf einen Datenbankserver erfolgen kann.
Im Rahmen der Umsetzung wurde der bisherige Service `DataService` des Prototyps in `SolrDataService` umbenannt (vgl. Abbildung 4.3). Im Zuge der Realisierung des dynamischen Wechsels zwischen den einzelnen verfügbaren Services, wurde der Name der bisher für die Datenanbindung Solrs zuständigen Klasse `DataService` umgewidmet. Den Namen trägt nun das Interface `DataService` (vgl. Abbildung 4.3). Der Aufruf der verschiedenen Services erfolgt über die Zustandsvariable `dataSource`, die als Datentyp das Interface `DataService` nutzt. Die Zustandsvariable `dataSource` ist analog zu anderen Einstellungsoptionen des Plugins, Teil eines Objekts, welches automatisch von Grafana in der Konfigurationsdatenbank persistiert wird. Diese Variable enthält zur Laufzeit der Anwendung eine Instanz des Service, des Datenbanksystems, welches der Anwender über das Einstellungsmenü (siehe Abschnitt 4.3.3) ausgewählt hat. Dadurch wird eine Entkopplung des Moduls, welches den Service, beziehungsweise dessen Schnittstelle nutzt und der konkreten Implementierung des Interface `DataService` erreicht. Infolgedessen kann im ganzen Programmcode des Plugins auf die Datenanbindung über das Interface `DataService` zurückgegriffen werden, anstatt wie bisher die jeweils konkrete Implementierung eines Service für eine Datenquelle aufrufen zu müssen. Ein Vorteil dieser Implementierung ist die Erweiterbarkeit um neue Datenquellen, denn diese müssen nur einmal in der Auswahllogik des Einstellungsmenüs zur Verfügung gestellt werden, anstatt wie bisher an allen Stellen im Programm, an denen ein Aufruf des Service für die Datenanbindung stattgefunden hat.
Das oben beschriebene Vorgehen entspricht dem sogenannten *Strategy Pattern*. Dabei handelt es sich um ein Entwurfsmuster in der Softwareentwicklung, welches den Wechsel zwischen den verschiedenen Implementierungen eines Algorithmus erlaubt [Gam+95]. Im vorliegenden Fall ermöglicht es den dynamischen Wechsel zwischen den verschiedenen Implementierungen des Interface `DataSource`.

Eine weitere Anforderung an die Integrationsarchitektur ist die Möglichkeit zur Aggregation von Zeitreihendaten über eine bestimmte Zeitspanne (vgl. Anforderung [R06] in Abschnitt 3.3). Diese Zeitspanne wird über die 2D-Visualisierung vom Anwender selektiert. Anhand der Zeitspanne, der Kennzahl und dem Aggregationstyp werden dann aus den vorhandenen Daten aggregierte Werte berechnet. Die dafür notwendige Programmlogik wurde in das oben beschriebene Interface `DataService` in Form der Methode `getAggregated` eingefügt (vgl. Abbildung 4.3). Nachfolgend soll die Implementierung dieser Funktionalität für die Datenquelle Solr beschrieben werden.
Die bisherige Selektion eines Zeitraumes mithilfe der 2D-Visualisierung des Prototyps verursachte keine Aggregation, stattdessen wurden durch die 3D-Visualisierung die Daten für den Zeitpunkt der unteren Intervallgrenze des ausgewählten Zeitintervalls dargestellt. Dieses Verhalten sollte so angepasst werden, dass bei der Auswahl

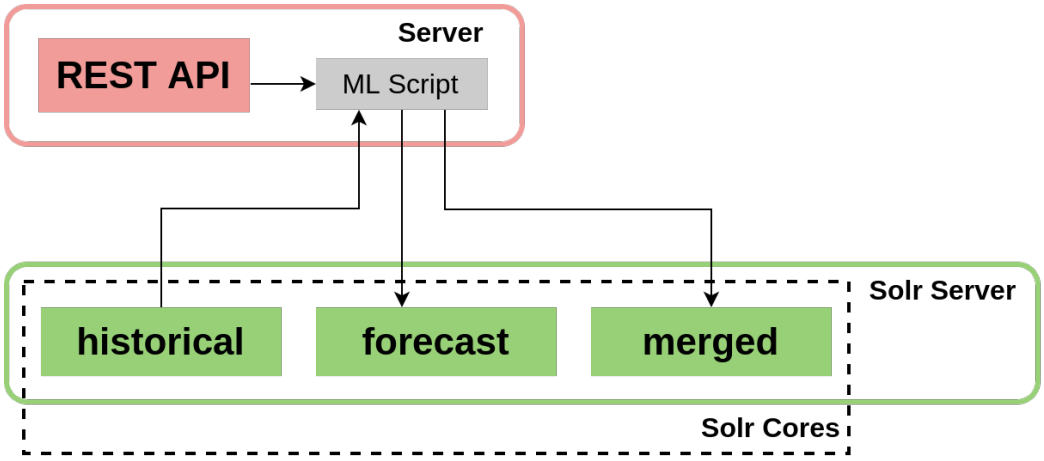

Abbildung 4.4: Die Abbildung zeigt die Integration des neuen Solr-Core `merged` in die bestehende Architektur. Er vereinigt die Datensätze aus der Vergangenheit mit den Vorhergesagten.

eines Zeitintervalls automatisch eine Aggregation über die in diesem Zeitintervall enthaltenen Werte durchgeführt wird. Bei der Implementierung wurde klar, dass das Zeitintervall, sofern die Vorhersagefunktion aktiviert wurde, sowohl in der Vergangenheit als auch in der Zukunft liegen kann, somit wären bei einer Aggregation sowohl die Zeitreihendaten des Solr-Core `historical`, als auch die des Solr-Core `forecast` betroffen. Deshalb wurde sich dazu entschieden, im Rahmen der Berechnung der Prognose durch den Vohersagealgorithmus (vgl. `ML Script` in Abbildung 4.1) neben dem Core `forecast` einen Core namens `merged` zu erstellen und darin die verschmolzenen Zeitreihendaten aus den beiden Solr-Cores `historical` und `forecast` zu speichern (vgl. Abbildung 4.4).
Dadurch konnten rechenintensive Operationen, wie die Verschmelzung der historischen und prognostizierten Zeitreihendaten und die Berechnung der aggregierten Werte auf dem Datenbankserver, anstatt im Client durchgeführt werden. Ansonsten hätten die im Zeitintervall liegenden Zeitreihendaten zuerst von den jeweiligen Solr-Cores in den Client geladen werden müssen. Woraufhin dort die aggregierten Werte berechnet werden würden. Aufgrund der eingeschränkten Leistungsfähigkeit der meisten Clients, hätte dies jedoch inbesondere bei großen Datenmengen zu erheblichen Einbußen bei der Performanz der Anwendung geführt, deshalb wurden im Rahmen der oben beschriebenen Lösung die Operationen auf den Solr-Server ausgelagert. Das Interface `DataService` (vgl. Abbildung 4.3) erlaubt es, unterschiedliche Implementierungen über die gleiche Schnittstelle, zur Verfügung zu stellen. Infolgedessen kann für jedes DBMS eine spezfische Implementierung der Aggregationsfunktionalität erfolgen, somit kann analog zu den anderen Methoden des Interface, je nach Funktionsumfang der Abfragesprache des jeweiligen DBMS, sich für eine Implementierung der Aggregationsfunktionalität im Client oder auf dem Datenbankserver entschieden werden.

Im Rahmen der Implementierung wurden zusätzlich zu den bereits beschriebenen Maßnahmen, Veränderungen an der für die 2D-Visualisierung (vgl. Abbildung 3.3) zuständigen Komponente vorgenommen, um die Anforderung einer verbesserten Performanz gegenüber des Prototyps gerecht zu werden. Dies umfasst die Auslagerung der Funktion, welche für jeden Zeitstempel, der in den Client geladenen Zeitreihendaten, den Mittelwert, das Minimum und das Maximum über alle Werte einer Kennzahl, welche über den selben Zeitstempel verfügen, berechnen. Dabei gilt es zu beachten, dass diese Funktion sich von der oben beschriebenen Methode `getAggregated` des Interface `DataService` insofern unterscheidet, dass bei der Methode `getAggregated` die aggregierten Werte für jede einzelne Instanz berechnet werden müssen, während bei der vorliegenden Funktion eine Aggregation über alle Instanzen pro Zeitstempel und Kennzahl durchgeführt wird.
Die Funktion sollte aus der Komponente der 2D-Visualisierung in die Services für die Datenquellen ausgelagert werden. Dadurch kann, sofern es die jeweilige Anfragesprache des betreffenden DBMS erlaubt, eine Auslagerung dieses rechenintensiven Vorgangs auf den jeweiligen Datenbankserver durchgeführt werden. Hierzu wurde erneut das Interface `DataService` (vgl. Abbildung 4.3) um die hierfür notwendige Methode erweitert (vgl. `getAggreagtedValueForEachTimestamp` in Abbildung 4.3), somit können die einzelnen Services spezifische Implementierungen, entsprechend den Fähigkeiten des jeweils verwendeten Datenbanksystems, beinhalten.

Eine weitere Optimierung des vorliegenden Plugins gegenüber dem Prototyp, ist die Selektion der Daten, die der Service vom Datenbankserver holt. Der Service `DataService`, welcher für die Datenanbindung von Solr im Prototyp verantwortlich war, unterstützte keine zeitintervallbasierte Selektion der Daten, welche vom betreffenden Datenbankserer in den Client geladen werden. Damit wurden bei jedem Aufruf der Methode `getLogData` alle verfügbaren Daten vom Datenbankserver geladen. Dies führte insbesondere bei größeren Datenmengen zu erheblichen Einschränkungen bei der Benutzbarkeit der Anwendung, da die Daten nachdem sie vom Datenbankserver geladen wurden, im Hauptspeicher des jeweiligen Clients abgelegt wurden. Abhilfe für dieses Problem schaffte die Einführung einer zeitintervallbasierten Selektion der Daten über die Benutzeroberfläche Grafanas (vgl. Abbildung 4.7). Dabei werden die beiden selektierten Intervallgrenzen über die Plugin-Schnittstelle den einzelnen React-Komponenten des Plugins zur Verfügung gestellt. Die Intervallgrenzen sind Teil des gleichen Objekts wie die anderen Einstellungsoptionen, sodass eine Persistierung der vorgenommenen Einstellungen ermöglicht wird. Die beiden selektierten Intervallgrenzen werden den beiden Methoden `getHistorical` und `getForecast` der jeweils zuständigen Service-Klasse beim Aufruf als Argumente übergeben. Der Service führt daraufhin eine Anfrage an den betreffenden Datenbankserver durch, welche ausschließlich die Daten für das ausgewählte Zeitintervall vom Server lädt.

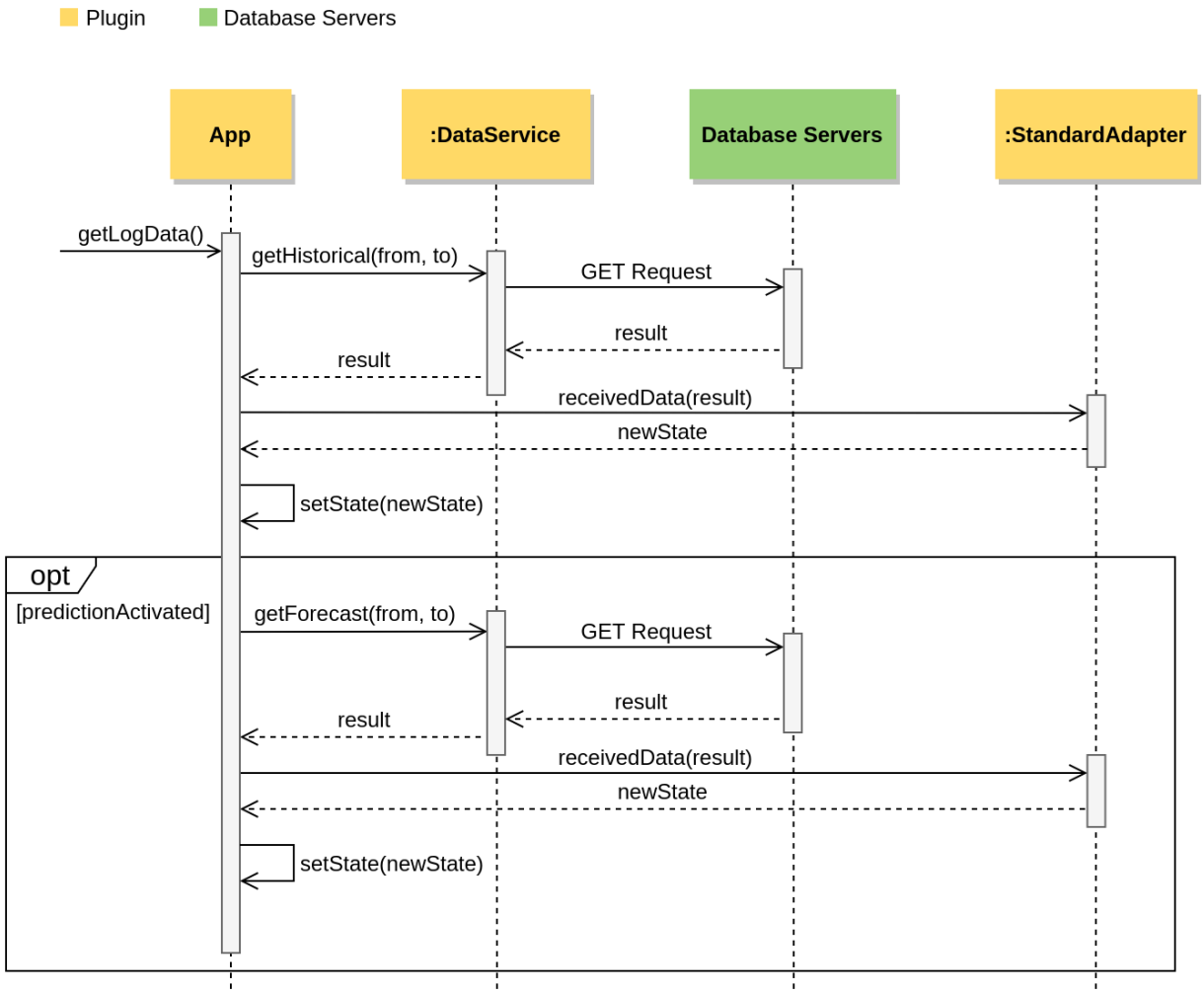

Abbildung 4.5: Das Sequenzdiagramm visualisiert den Prozess der Datenanbindung, welcher entweder durch die Initialisierung der Anwendung oder durch das Klicken der Schaltfläche für die manuelle Aktualisierung angestoßen wird.

Ein Unterschied zur Datenanbindung des Prototyps (vgl. Abbildung 3.4), stellt die Einführung der Zustandsvariable `predictionActivated` (vgl. Abbildung 4.5) dar. Dabei handelt es sich um eine boolesche Variable, welche den Wert `true` zugewiesen bekommt, wenn der Anwender die Vorhersage aktiviert. Das Aktivieren und Deaktivieren der Vorhersage geschieht über das Einstellungsmenü des Plugins (siehe Abschnitt 4.3.3). Die Variable erlaubt das optionale Laden der mithilfe des `ML Scripts` (vgl. Abbildung 4.1) vorhergesagten Werte, indem in der Methode `getLogData` (vgl. Abbildung 4.5) geprüft wird ob die Variable `predictionActivated` den Wert `true` enthält. Damit wird die Prognose nur geladen, sofern diese vom Anwender aktiviert wurde, was eine beschleunigte Ausführung der Methode `getLogData` bei deaktivierter Vorhersage zur Folge hat, weil die Daten der Vorhersage nicht in den Client geladen und dort mithilfe der Methode `receivedData` vom `StandardAdapter` transformiert werden müssen. Außerdem wird hierdurch der Hauptspeicher des

Clients entlastet, da dieser keine nicht benötigten Daten bereitstellen muss, was wiederum die Performanz der gesamten Anwendung erhöht.

Der in Abbildung 4.5 beschriebene Prozess lädt die Daten, welche bei einer Selektion eines Zeitpunkts über die 2D-Visualisierung (vgl. Abbildung 3.3), benötigt werden. Die Methode `getLogData` wird nicht bei der Auswahl eines neuen Zeitpunkts in der 2D-Visualisierung aufgerufen, sondern nur bei der Initialisierung und der manuellen Aktualisierung. Dies liegt daran, dass die Daten bereits nach der Initialisierung der Anwendung in der Zustandsvariable `timeSeries` der Wurzelkomponente `App` des Plugins und damit im Hauptspeicher vorhanden sind.
Sollte jedoch ein Zeitintervall anstatt eines Zeitpunkts über die 2D-Visualisierung selektiert werden, so wird automatisch die Methode `getAggregatedLogData` aufgerufen, welche ähnlich zu der in Abbildung 4.5 beschriebenen Sequenz abläuft. Dabei wird die Methode `getAggregated` der Instanz der Klasse `DataService` augerufen. Diese berechnet dann die aggregierten Werte, beziehungsweise lässt sie, sofern es die jeweilige Datenbankabfragesprache zulässt, auf dem korrespondierenden Datenbankserver berechnen. Nachdem die Berechnung auf dem Datenbankserver erfolgt ist, werden die Werte an die ursprünglich aufrufende Methode `getAggregatedLogdata` zurückgesendet, wo sie in Empfang genommen werden und in einer Zustandsvariable der Wurzelkomponente `App` gespeichert werden. Die Zustandsvariable ist gleichzeitg ein Property der für die 3D-Visualisierung zuständigen Komponente und verursacht bei jeder Änderung ein automatisches Neuladen mit den aggregierten Daten des, über die 2D-Visualisierung ausgewählten Zeitintervalls. Die Methode `getAggregatedLogData` wird im Gegensatz zur `getLogData` bei jeder erneuten Auswahl eines Zeitintervalls über die 2D-Visualisierung durchgeführt. Dies liegt daran, dass die Aggregation für jedes Zeitintervall individuell berechnet werden muss.

4.3 Benutzeroberfläche

Der nachfolgende Abschnitt wird sich mit der Integration der Benutzeroberflächenelemente des Prototyps in die Benutzeroberfläche des Plugins and Grafanas befassen. Im Folgenden sollen die allgemeinen Aspekte der vorliegenden Integration der Benutzeroberfläche diskutiert werden. Daraufhin wird den einzelnen von der Integration betroffenen Bereichen der Benutzeroberfläche Grafanas, jeweils ein eigener Abschnitt gewidmet.

Wie bereits bekannt, wurden die Benutzeroberflächenelemente des Prototyps mithilfe von React-Komponenten entwickelt (vgl. Abschnitt 3.1.2). Die meisten dieser Komponenten verfügen über eine dedizierte CSS-Datei, in welcher das Erscheinungsbild des betreffenden Teils der Benutzeroberfläche gestaltet wird. Zur Erleichterung

wurde hierbei vom CSS-Framework Bootstrap[3] Gebrauch gemacht.
Bei der Integration der Komponenten in die Benutzeroberfläche Grafanas stellte sich heraus, dass einige CSS-Klassen der React-Komponenten aus dem Prototyp mit den CSS-Klassen Grafanas interferierten, was zu unschönen Grafikfehlern und Inkonsistenzen in der Oberflächengestaltung des Plugins führte. Es stellte sich heraus, dass die von Grafana verwendeten CSS-Klassen teilweise eine Überlappung hinsichtlich der Namensgebung mit den von Bootstrap verwendeten CSS-Klassennamen aufweisen, was nicht weiter überrascht, da beide sehr generische Begriffe als Bezeichnung für die einzelnen CSS-Klassen verwenden. Dies führte dazu, dass in einigen Teilen der Benutzeroberfläche Grafanas die CSS-Klassen der Komponenten des Prototyps anstatt der CSS-Klassen Grafanas vom Browser zur Darstellung genutzt wurden. Dieser Effekt liegt in der Vorgehensweise von Browsern beim Rendervorgang einer Webseite begründet. Dabei wird den CSS-Klassen, die später im HTML-Dokument definiert wurden, eine vorrangige Behandlung gegenüber früher definierten CSS-Klassen eingeräumt [W3C19b]. Dies führte im vorliegenden Fall dazu, dass die CSS-Klassen der Komponenten des Prototyps, die CSS-Klassen von Grafana, im Falle eines gleichen Klassenbezeichners, überdeckten. Leider ließ sich im vorliegenden Fall die Reihenfolge der CSS-Dateien im HTML-Dokument nicht ohne Weiteres ändern, da die Plugin-Schnittstelle zur Laufzeit automatisch die CSS-Dateien der Plugins nach den CSS-Dateien Grafanas im HTML-Dokument platziert.
Das oben beschriebene Problem ließe sich durch das Hinzufügen von Präfixes zu den CSS-Klassennamen der React-Komponenten des Prototyps lösen, jedoch wurde diese Option im Lichte einer Alternative verworfen. Diese bestand darin, anstatt der CSS-Klassen von Bootstrap, die CSS-Klassen von Grafana für die Gestaltung der einzelnen Komponenten des Prototyps zu nutzen. Dieser Ansatz hat den entscheidenden Vorteil, dass dadurch eine konsistente und einheitliche Benutzeroberfläche geschaffen wird, welche die Voraussetzung für die Erfüllung der Anforderung [R02] aus Abschnitt 3.3 ist.
Bei der Entwicklung stellte sich die Programmbibliothek `@grafana/ui` als sehr hilfreich heraus. Dabei handelt es sich um eine Sammlung von generischen Benutzeroberflächenkomponenten wie zum Beispiel Schaltflächen zur Auswahl von Einstellungen. Anhand dieser konnten die verfügbaren CSS-Klassen getestet und auf ihre visuellen Eigenschaften hin analysiert werden. Dies war notwendig da zum Zeitpunkt der Implementierung der vorliegenden Integrationsarchitektur keine Dokumentation für die Bibliothek und die verfügbaren CSS-Klassen vorhanden war.

[3] https://getbootstrap.com/ (besucht am 21. Jan. 2020)

4.3.1 Panel

Die 3D-Visualisierung und die dazugehörige 2D-Visualisierung stellen die Kernfunktionalität des Prototyps dar und wurden dort zentral in der Anwendung platziert (vgl. Abbildung 3.2). Dies konnte auch im vorliegenden Plugin erreicht werden. Hierzu wurden die Visualisierungen direkt im Panel des Plugins platziert (vgl. Abbildung 4.6), währenddessen die dazugehörigen Einstellungsmöglichkeiten separat in das extra dafür vorgesehenen Menü ausgelagert wurden (vgl. Abbildung 4.9). Sowohl die 3D-Visualisierung, als auch die 2D-Visualisierung, befinden sich im Prototyp jeweils in einer eigenen Komponente. Dieser Umstand erleichterte die vorliegende Integration erheblich, da die Komponenten einfach in die Komponentenhierarchie des Plugins eingefügt werden konnten, sodass sie im Panel-Bereich des Plugins platziert werden konnten.

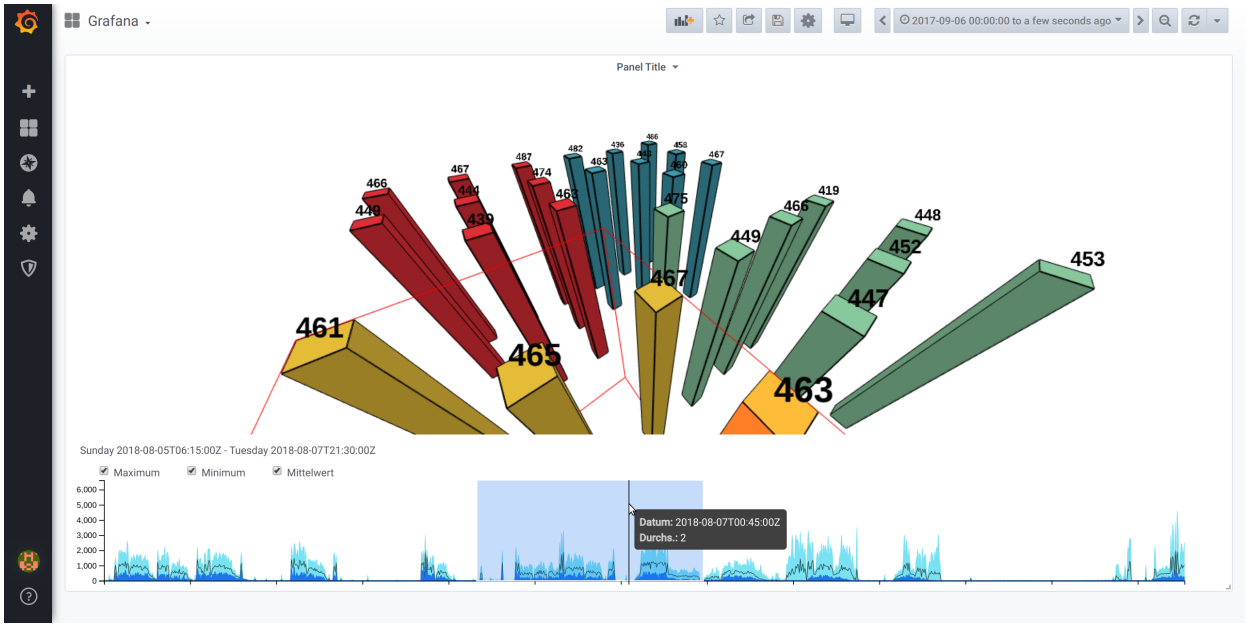

Abbildung 4.6: Die Abbildung zeigt die 3D- und 2D-Visualisierung gemeinsam in einem Panel in Grafana.

Die Benutzeroberfläche des Prototyps ist aufgrund der begrenzten Entwicklungszeit nur teilweise responsiv gestaltet worden. Das heißt, dass manche Komponenten und Schaltflächen sich nicht an die individuellen Bildschirmgrößen der Clients anpassen. Dieser Nachteil wurde bis dato nicht deutlich, da die Laptops, der an der Entwicklung des Prototyps beteiligten Personen, alle eine ähnliche Auflösung hatten. Eine responsive Gestaltung der Benutzeroberfläche von Web-Anwendungen steigert die Benutzerfreundlichkeit dieser [Wei+14], deshalb wurden wo möglich Anpassungen an den bestehenden Komponenten vorgenommen, um eine weitestgehend responsive Benutzeroberflächengestaltung im fertigen Plugin zu erzielen.
Hierzu zählt unter anderem die 3D-Visualisierung aus dem Prototyp, welche mit festen Pixelhöhen und -breiten implementiert worden ist. Die three.js-Schnittstelle

erwartet die Pixelhöhe und -breite der Visualisierung als Argumente. Deshalb wurden die festen Werte durch die Höhe und Breite des Containers, der die 3D-Visualisierung enthält, ersetzt. Damit nutzt die 3D-Visualisierung bei der Initialisierung automatisch die maximale Höhe und Breite im Panel, sodass der vorhandene Platz im Panel optimal ausgenutzt wird.
Der Anwender kann jedoch nach der Initialisierung die Größe des Browserfensters anpassen, mit welchem die Anwendung aufgerufen wird. In diesem Fall funktioniert die oben genannte Lösung nicht mehr, da die Dimensionen, welche zur Initialisierung der Anwendung optimal waren, es nun nicht mehr sind. Deshalb wurde die Methode `updateCanvasDimensions` implementiert. Diese bestimmt die optimalen Dimensionen über die Werte des Containers und wird im Browserfenster als Ereignisbehandlungsroutine für das Browser-Ereignis `resize` registriert, sodass sie bei einer Anpassung der Fenstergröße des Browsers automatisch aufgerufen wird.
Eine weitere Möglichkeit zur Änderung der Größe der Visualisierung besteht in der manuellen Anpassung der Größe des Panels in Grafana. Es stellte sich heraus, dass in diesem Fall die oben beschriebene Lösung nicht mehr funktioniert. Das liegt daran, dass die Anpassung des Panels keine Auslösung des Ereignisses `resize` zur Folge hat. Jedoch erfolgt bei der Anpassung der Größe des Panels ein Aufruf der für für das Rendering der 3D-Visualisierung zuständigen Methode in der React-Komponente. Deshalb wurde in dieser Methode ein Aufruf der bereits weiter oben erwähnten Methode `updateCanvasDimensions` hinzugefügt, sodass bei der Anpassung der Panelgröße, die nicht das Ereignis `resize` auslöst, eine dynamische Anpassung der Größe der 3D-Visualisierung erfolgen kann.

4.3.2 Zeitauswahl

Im Abschnitt 4.2 wurde die Klasse `DataService` vorgestellt, welche die Anbindung verschiedener Datenbanksysteme in der Anwendung ermöglicht. Dieser Service ermöglicht die Selektion der vom Server kommenden Daten mithilfe von Zeitintervallen. Im nun folgenden Kapitel soll die Auswahl dieses Zeitintervalls über die Benutzeroberfläche des Plugins realisiert werden.

Zuerst ist festzustellen, dass die Zeitauswahl in der Kopfleiste des Prototyps (vgl. obere Leiste in Abbildung 3.2) und die Zeitauswahl Grafanas (vgl. Abbildung 4.7) dem gleichen Zweck, nämlich der Festlegung eines Zeitintervalls für die Auswahl der vom Datenbankserver kommenden Daten, dienen. Deshalb bietet sich eine Konsolidierung der beiden Komponenten im Rahmen der Entwicklung des Grafana-Plugins an.
Im vorliegenden Fall soll die Zeitauswahl des Prototyps wegfallen und stattdessen die native Zeitauswahl Grafanas für das Plugin genutzt werden. Dieses Vorgehen hat den Vorteil, dass die Benutzeroberfläche keine redundanten Funktionen enthält.

Außerdem müssen die Anwender keine neue Benutzeroberfläche erlernen. Zusätzlich wird hiermit eine einheitliche und konsistente Gestaltung der Benutzeroberfläche erreicht.

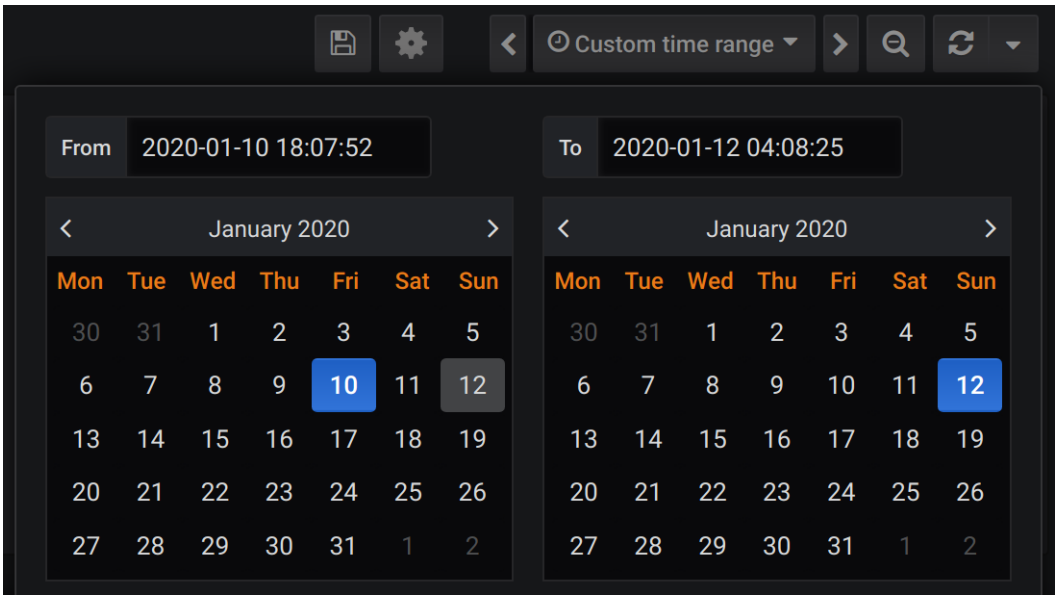

Abbildung 4.7: Das Menü für die globale Zeitauswahl in Grafana.

Die zuständige Methode des oben erwähnten Services erwartet beim Aufruf die untere Grenze und die obere Grenze des Zeitintervalls als Argumente. Diese Argumente müssen als Zeitstempel im ISO-Format vorliegen, damit die Methode korrekt funktioniert und die Daten auf dem Datenbankserver entsprechend selektiert werden können. Sollten die Zeitstempel aus der Grafana-Komponente nicht in diesem Format vorliegen, muss eine entsprechende Transformation der Zeitstempel vorgenommen werden.
Deshalb muss ein Weg gefunden werden, wie die beiden Intervallgrenzen aus der bestehenden Grafana-Komponente extrahiert, transformiert und in das Plugin überführt werden können.

In einem ersten Schritt wurde die Plugin-Schnittstelle Grafanas auf Möglichkeiten zur Extraktion der gesuchten Zeitstempel untersucht. Bei der Untersuchung stellte sich heraus, dass die Plugin-Schnittstelle viele Einstellungsdaten dynamisch über sogenannte *Properties*, welche in Form von Interfaces definiert sind, zur Verfügung stellt. Dabei handelt es sich um Variablen, die zur Laufzeit einer React-Komponente übergeben werden und dort zur weiteren Verwendung zur Verfügung stehen. Im vorliegenden Fall wurde das Interface `PanelProps` der Plugin-Schnittstelle genutzt. Dieses Interface wird von der Wurzelkomponente des Plugins (vgl. `App` in Abbildung 4.8 implementiert. Die darin definierten Variablen und Methoden umfassen unter anderem die Variable `timeRange`, die das derzeit ausgewählte Intervall enthält (vgl. Abbildung 4.8). Damit mussten nur noch die beiden Intervallgrenzen aus

dem Zeitintervall extrahiert, und lokal als Zustandsvariable in der Komponente `App` gespeichert werden, sodass diese dem Service zur Datenanbindung zur Verfügung stehen.

4.3.3 Einstellungen

Nachdem in Abschnitt 4.1 die grobe Funktionsweise des Persistenzmechanismus für benutzerdefinierte Einstellungen und dessen Einbettung in die Gesamtarchitektur beschrieben worden sind, soll im Folgenden eine detaillierte Beschreibung des Vorgangs folgen. Dabei soll unter anderem auch die Umsetzung des Einstellungsmenüs im Grafana-Plugin geschildert werden.

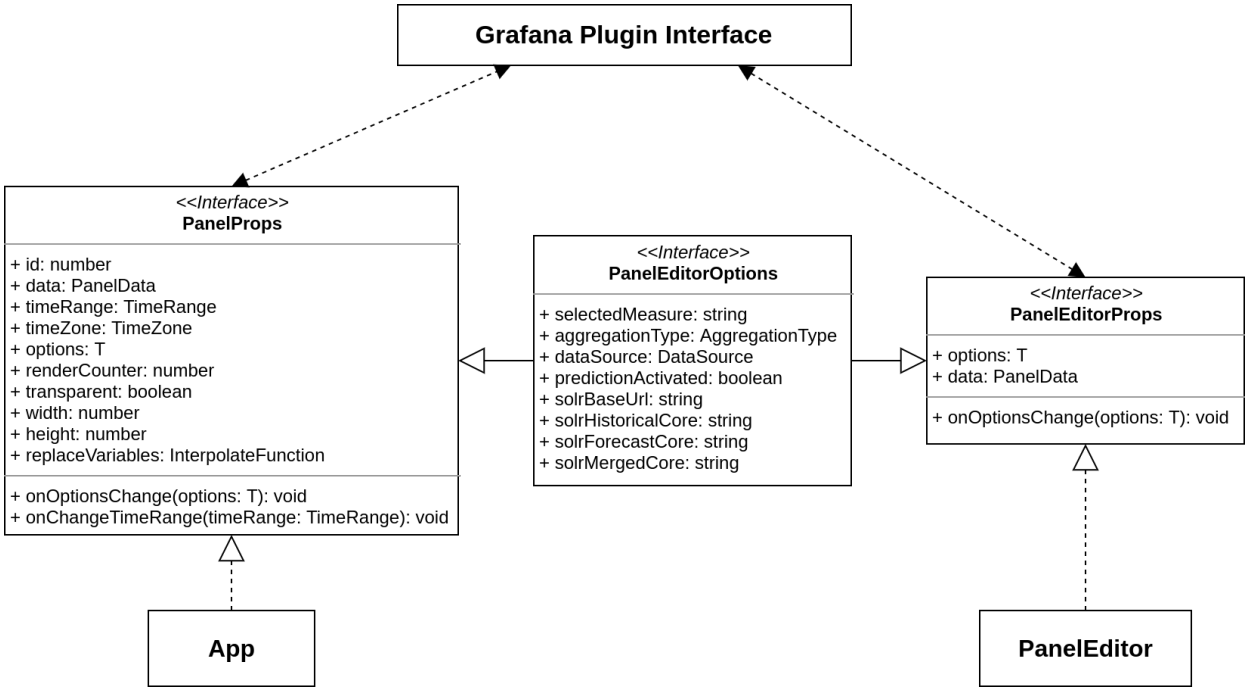

Abbildung 4.8: Die Abbildung veranschaulicht die Beziehungen zwischen den Interfaces, welche von den beiden React-Komponenten `App` und `PanelEditor` implementiert werden müssen, damit die Persistierung der benutzerdefinierten Einstellungen (`PanelEditorOptions`) über die Plugin-Schnittstelle Grafanas reibungslos funktioniert.

Grafana stellt über die Plugin-Schnittstelle für Panel-Plugins eine Methode namens `setEditor` zur Verfügung, die dazu dient eine Einstellungseite zum Plugin hinzuzufügen. Von dieser Funktionalität wurde auch im vorliegenden Fall Gebrauch gemacht, um das Einstellungsmenü aus dem Prototyp in das Plugin zu überführen.
Die Basis des Einstellungsmenüs im Plugin bildet die React-Komponente `PanelEditor`, welche das Interface `PanelEditorProps` implementiert (vgl. Abbildung 4.8). Die Komponente `PanelEditor` existiert parallel zur Wurzelkomponente `App`, die den Rest der Programmlogik des Plugins enthält. Damit die in der Komponente `PanelEditor` vorgenommenen Einstellungen persistiert werden können, müssen diese der Plugin-

Schnittstelle bekannt gemacht werden. Dies lässt sich durch die Erweiterung des Interface `PanelEditorProps` realisieren. Im vorliegenden Fall geschah dies durch das eigens angelegte Interface `EditorPanel` (vgl. Abbildung 4.8). Hierin wurden die Variablen, welche durch das Einstellungsmenü festgelegt werden können, definiert. Damit die Einstellungswerte auch in der Wurzelkomponente `App` zur Verfügung stehen, musste das dort verwendete Interface `PanelProps` ebenfalls durch das Interface `PanelEditorProps` erweitert werden (vgl. Abbildung 4.8).
Die Methode `onOptionsChange` ist sowohl Teil des Interface `PanelProps`, als auch des Interface `PanelEditorProps` und ermöglicht die dynamische Änderung der Einstellungen durch das Einstellungsmenü. Diese Methode musste in die Anwendungslogik des bisherigen Einstellungsmenüs integriert werden, damit bei jedem Aufruf der Methode, die React-Komponenten, welche auf die Einstellungen zugreifen, aktualisiert werden können. Ein Beispiel hierfür ist die Aktivierung des Vorhersagemodus, der die Visualisierungen um die prognostizierten Daten erweitert.
Zur Laufzeit werden die Interfaces durch konkrete Objekte, die unter anderem den Austausch der Einstellungen zwischen der Konfigurationsdatenbank und dem Plugin erlauben, ersetzt. Bei jeder Änderung der Einstellungen über die Methode `onOptionsChange` wird ein Objekt mit den aktualisierten Einstellungen über die Plugin-Schnittstelle im Grafana-Client an die REST-Schnittstelle auf dem Grafana-Server gesendet (vgl. Abbildung 4.1). Im Anschluss speichert die REST-Schnittstelle des Grafana-Servers die Datei in der dafür vorgesehenen Konfigurationsdatenbank (vgl. Config DB in Abbildung 4.1).

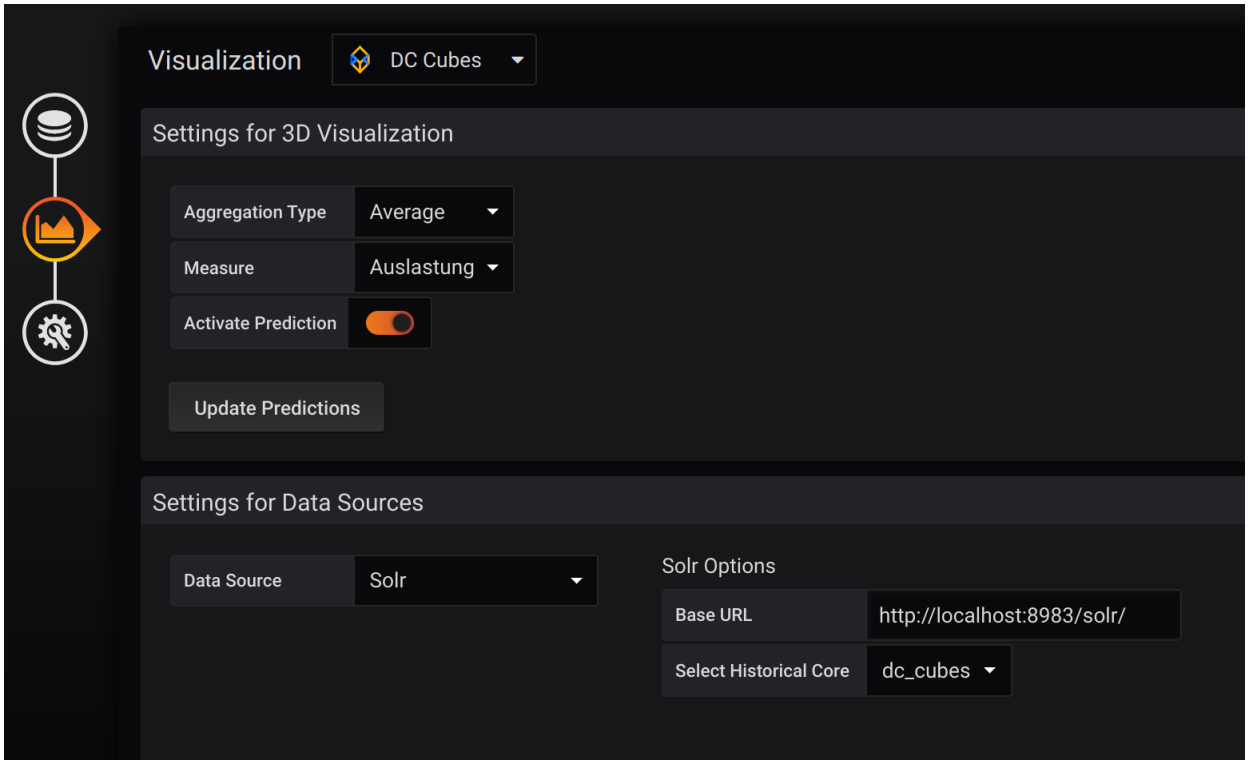

Abbildung 4.9: Das Einstellungsmenü des Panel-Plugins.

Analog zur restlichen Benutzeroberfläche sollte auch das Einstellungsmenü des Plugins mithilfe der Komponenten aus der Bibliothek `@grafana/ui` umgesetzt werden, um eine gestalterisch konsistente Benutzeroberfläche im fertigen Plugin zu erzielen. Deshalb wurden die bereits vorhandenen Komponenten aus dem Prototyp diesbezüglich angepasst und mit den nötigen CSS-Klassen versehen. Das Ergebnis dieser gestalterischen Anpassung der Benutzeroberfläche lässt sich in Abbildung 4.9 betrachten.
Das Einstellungsmenü beinhaltet sowohl die Einstellungen, welche die Visualisierung betreffen, als auch Einstellungsmöglichkeiten für alle unterstützten Datenbanksysteme.
Die Einstellungen für die Visualisierungen wurden aus der Komponente der 3D-Visualisierung extrahiert und im Einstellungsmenü platziert (vgl. Abbildung 4.9), damit sich alle Einstellungen des Plugins an einem Ort befinden und das Panel ausschließlich für die Visualisierung der vorhandenen Daten zuständig ist. Des Weiteren wurden die Einstellungsmöglichkeiten, welche die Vorhersagefunktionalität betreffen, in das Einstellungsmenü eingefügt (vgl. Abbildung 4.9).

4.3.4 Datenaktualisierung

Die manuelle Aktualisierungsfunktion aus dem Prototyp soll den Anwendern auch im fertigen Plugin zur Verfügung stehen. Dabei ist festzustellen, dass Grafana analog zur Zeitauswahl, bereits über ein Benutzeroberflächenelement zur manuellen Aktualisierung der Daten verfügt. Das Element ist in Abbildung 4.7 in der oberen rechten Ecke zu sehen. Im Folgenden soll die Integration jenes Elements in die hiesige Integrationsarchitektur beschrieben werden.

Aufgrund der Tatsache, dass die Integrationsarchitektur nicht die Datenanbindung über Grafana nutzt, sondern eine eigens entwickelte, konnte die bestehende Aktualisierungschaltfläche der Benutzeroberfläche Grafanas nicht ohne Weiteres genutzt werden. Deshalb musste ein Weg gefunden werden, die Aktualisierungsfunktion Grafanas mit dem für die Datenanbindung zuständigen Programmcode im Plugin zu verknüpfen.
Eine Untersuchung der Plugin-Schnittstelle auf potentielle Möglichkeiten zur Anbindung von Grafanas Aktualisierungsfunktion zeigte, dass die Bibliothek `@grafana/data` ein Interface für ein Event namens `refresh` bereitstellt. Dabei handelt es sich um ein Event, welches mithilfe der Klasse `EventEmitter` aus der Standardbibliothek von Angular erzeugt wird. Dieses Event wird auch intern von der Aktualisierungskomponente Grafanas benutzt. Diese Komponente ist leider, wie viele andere Teile der Benutzeroberfläche von Grafana, noch nicht zu React migriert worden [Öde18] und kann deshalb auch nicht in der Plugin-Schnittstelle für React angebunden werden. Dies liegt daran, dass Events, welche mithilfe der Klasse `EventEmitter` in Angular

erzeugt wurden, nur in der jeweiligen Angular-Anwendung zur Verfügung stehen und nicht ohne Weiteres in einer parallel laufenden React-Anwendung verwendet werden können.
Die Bereitstellung eines ähnlichen Mechanismus in Grafanas Plugin-Schnittstelle für React steht noch aus. Deshalb musste eine Alternative für die Integration des Aktualisierungsmechanismus gefunden werden.

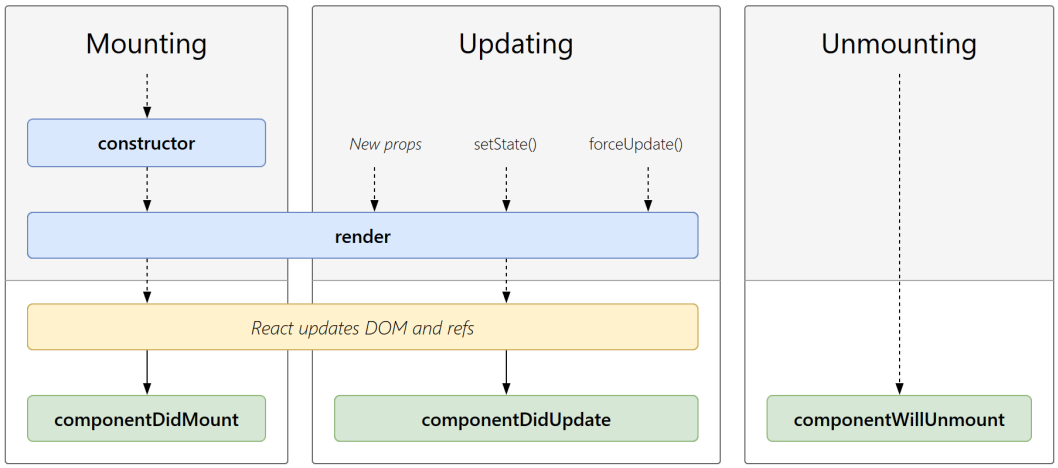

Abbildung 4.10: Eine übersichtliche Darstellung aller verfügbaren Lifecycle-Methoden in React [Maj19].

Im Verlauf der Suche nach einer Lösung stellte sich heraus, dass sich bei jedem Klick auf die Aktualisierungsschaltfläche ein Property, welches der Wurzelkomponente `App` des Plugin durch das Interface `PanelProps` (vgl. Abbildung 4.8) übergeben wird, ändert. Im vorliegenden Fall handelte es sich dabei um das Property `requestID`, welches eine Zahl speichert, die bei jedem Klick auf die Aktualisierungsschaltfläche von Grafana um eins erhöht. Eine Änderung an den Properties, in diesem Fall das Addieren der Zahl 1 auf die in der Variable `requestID` gespeicherte Zahl, verursacht einen Aufruf der sogenannten Lifecycle-Methode `componentDidUpdate` (vgl. Abbildung 4.10) in dem Plugin. Damit musste der Wert der Variable nur noch als Variable lokal in der Komponente `App` gespeichert werden und bei jedem Aufruf der Methode `componentDidUpdate` eine Prüfung stattfinden, ob sich in den neuen Properties die Variable `requestID` gegenüber der lokal gespeicherten Version der Variable verändert hat. Sofern dies der Fall ist, wird die Methode `getLogData` aufgerufen, was eine Aktualisierung der lokalen Datenbasis zur Folge hat.
Die oben beschriebene Lösung ermöglicht die Nutzung der nativen Schaltfläche Grafanas zur Aktualisierung der lokal geladenen Datenbasis in der Integrationsarchitektur und leistet damit einen Beitrag zur Erfüllung der Anforderung [R02] aus Abschnitt 3.3, da sie eine durchgängig konsistente Benutzeroberfläche ermöglicht.

Eine Alternative zu der oben genannten Vorgehensweise wurde ebenfalls in Betracht gezogen. Dabei hätte die Angular-Klasse `EventEmitter`, welche für die Auslösung des Events `refresh` verantwortlich ist, so modifiziert werden sollen, dass parallel zur Auslösung des Events durch den `EventEmitter` in Angular, ein klassisches JavaScript-Event erzeugt wird und über das globale Objekt `window` der Browser-Schnittstelle, auch im React-Programmcode des Plugins zur Verfügung steht. Dieses Vorgehen hätte jedoch eine Modifikation des Grafana-Programmcodes erfordert und wurde damit als unvereinbar mit der Anforderung [R04] aus Abschnitt 3.3 eingestuft und letztendlich verworfen.

4.4 Server

Die Notwendigkeit eines zusätzlichen Servers mit einer REST-Schnittstelle neben dem bestehenden Grafana-Server in der Integrationsarchitektur, ergibt sich aus der nicht vorhandenen Möglichkeit, die REST-Schnittstelle des Grafana-Servers ohne Modifikation zu erweitern.
Die Bereitstellung des Vorhersagealgorithmus erfordert jedoch einen Server, da der Vorhersagealgorithmus nicht als JavaScript-Code für den Browser vorliegt und damit nicht mit dem restlichen JavaScript-Programmcode des Plugins über den Browser ausgeliefert werden kann. Theoretisch ließe sich der für die Vorhersage zuständige Programmcode (vgl. Abbildung 4.1) zwar von der Programmiersprache Python in JavaScript konvertieren und damit im restlichen Programmcode des Plugins einfügen, jedoch hätte dies einen erheblichen Mehraufwand bei der Entwicklung der Integrationsarchitektur zur Folge gehabt. Außerdem hätte sich die Verlagerung der Vorhersagefunktionalität in den Client, negativ auf die Performanz der gesamten Anwendung ausgewirkt, da der Vorhersagealgorithmus mit allen verfügbaren Vergangenheitsdaten arbeitet und die meist leistungsschwachen Clients damit überfordert wären.
Ein weiterer Grund für die Verwendung einer erweiterbaren REST-Schnittstelle in der Integrationsarchitektur ist die Notwendigkeit einer Möglichkeit zur Umgehung des CORS-Sicherheitsmechanismus, über welchen alle modernen Browser verfügen. Der Grafana-Server verfügt zwar auch über eine solche Funktionalität, jedoch ist diese auf die Verwendung der Datenbankschnittstelle Grafanas ausgelegt und ist somit nicht mit der im vorliegenden Fall selbstentwickelten Datenanbindung kompatibel.
Schließlich ist noch zu erwähnen, dass die Verwendung einer von Grafana unabhängigen REST-Schnittstelle eine einfachere Portierbarkeit der Integrationsarchitektur auf ähnlich gelagerte Anwendungsfälle wie zum Beispiel die Integration der vorliegenden Visualisierung in das ITOA-System Kibana, ermöglicht. Damit wird auch der Anforderung [R01] aus Abschnitt 3.3 Rechnung getragen.

Die bisherige REST-Schnittstelle aus dem Prototyp konnte in die Integrationsarchitektur übernommen werden, jedoch musste die Proxy-Funktionalität noch hinzugefügt werden. Hierfür wurde im Plugin eine neue Funktion eingefügt, welche bei jedem Aufruf einer Methode des für die Datenanbindung zuständigen Service dynamisch prüft, ob der optionale Server mit der REST-Schnittstelle (vgl. Abbildung 4.1) verfügbar ist. Ist dies der Fall, so werden alle ausgehenden HTTP-Anfragen des Service über die REST-Schnittstelle geleitet, welche als Proxy fungiert und die Anfragen an die jeweilige Zieldatenbank weiterleitet.
An der Vorhersagefunktionalität auf dem Server (vgl. *ML Script* in Abbildung 4.1) mussten keine Änderungen gegenüber der Version aus dem Prototyp vorgenommen werden. Die Umsetzung der Einstellungsmöglichkeiten für die Vorhersagefunktionalität in der Benutzeroberfläche des Plugins ist bereits im Rahmen der Implementierung des Einstellungsmenüs in Abschnitt 4.3.3 erfolgt (vgl. Abbildung 4.9).

4.5 Grafana-Server

Die Anforderung [R03] aus Abschnitt 3.3 beschreibt die Notwendigkeit eines Persistenzmechanismus für benutzerdefinierte Einstellungen in der fertigen Integrationsarchitektur. Diese sollte sich in der fertigen Integrationsarchitektur wiederfinden.

Hierzu wurde die Konfigurationsdatenbank *Config DB* auf dem Grafana-Server genutzt (vgl. Abbildung 4.1). Die REST-Schnittstelle des Grafana-Server ermöglicht dem Grafana-Client und damit dem Plugin, den Zugriff auf die dort gespeicherten Einstellungsdaten.
Neben SQLite lassen sich auch die beiden DBMS PostgreSQL[4] und MySQL[5] als Konfigurationsdatenbank auf dem Grafana-Server festlegen [Lab19c]. Im vorliegenden Fall wurde sich aus Gründen der Einfachheit jedoch für SQLite entschieden, da es sich dabei im Gegensatz zu den anderen beiden Systemen, um ein Datenbanksystem handelt, welches die Daten in einer einfachen sqlite-Datei anstatt eines Datenbankserver mit einer Datenbank speichert [SQL19]. Bei den anderen Datenbanksystemen wäre hingegen eine aufwendige Installation und Konfiguration auf dem Zielsystem nötig, damit die Datenbankserver an Grafana angebunden werden können, was jedoch einen unverhältnismäßigen Aufwand für die Einrichtung eines Plugins darstellen würde und deshalb verworfen wurde.

Neben der oben beschriebenen Lösung wurde auch eine weitere Variante zur Speicherung der Konfigurationsdaten im Laufe der Implementierung evaluiert. Diese soll nachfolgend dargelegt werden.

[4] https://www.postgresql.org/ (besucht am 12. Feb. 2020)
[5] https://www.mysql.com/ (besucht am 12. Feb. 2020)

Der Persistenzmechanismus wäre auch über eine eigene Konfigurationsdatenbank auf dem Server mit seiner erweiterbaren REST-Schnittstelle realisierbar gewesen, jedoch wäre dann zusätzlicher Entwicklungsaufwand nötig gewesen, um die Programmlogik für das Speichern und Laden der Einstellungen sowohl auf dem Server, als auch im Plugin umzusetzen. Ein weiter Nachteil einer solchen Lösung, läge in der fragmentierten Speicherung der Einstellungsdaten, denn neben dem vorliegenden Plugin, nutzt auch der restliche Teil des Grafana-Clients die Schnittstelle zur Speicherung verschiedenster Einstellungsdaten. Ein Vorteil dieses Vorgehens wäre dagegen die leichtere Portierbarkeit der Integrationsarchitektur auf andere ITOA-Systeme neben Grafana, denn der beschriebene Persistenzmechanismus würde unabhängig von den Schnittstellen Grafanas funktionieren.
Nichtsdestotrotz wurde sich nach einer eingehenden Prüfung und Abwägung oben genannter Vor- und Nachteile für erstere Lösungsalternative entschieden.

4.6 Bereitstellung

Der folgende Abschnitt wird kurz auf einzelne Aspekte der Bereitstellung des Plugins eingehen.

Die Module und Komponenten des Plugins der Integrationsarchitektur befinden sich alle im selben Ordner. Dieser liegt wiederum in einem Unterordner des Verzeichnisses des Prototyps. Somit kann das Plugin einfach die bereits bestehenden Services des Prototyps importieren und wiederverwenden. Damit entfällt ein aufwendiges Kopieren der benötigten Dateien, was wiederum zu Inkonsistenzen führen könnte, da die Dateien nach einer Anpassung immer manuell in das Verzeichnis des Plugins kopiert werden müssten.
Die Dateien des Servers, welcher die REST-Schnittstelle mit dem Proxy und der Vorhersagefunktionalität bereitstellt, befinden sich ebenfalls im selben Verzeichnis und können sowohl vom Prototyp als auch vom Plugin gleichermaßen verwendet werden.

Ein eigener Build-Prozess wandelt die TypeScript-Artefakte des Plugins in JavaScript-Dateien um. Im Rahmen dieses Prozesses werden die einzelnen Artefakte zusammengefasst und in einem eigenen Ordner platziert. Dieser Ordner muss daraufhin in das Plugin-Verzeichnis Grafanas kopiert werden, sodass Grafana das neue Plugin erkennt, woraufhin der Anwender das Plugin über die Benutzeroberfläche in Grafana installieren kann.
Der Server (vgl. Abbildung 4.1) kann analog zum Grafana-Server als Autostart-Prozess im jeweiligen Betriebssystem registriert werden, was die Administration des Systems erleichtern sollte.

5 Ergebnisse

Nachfolgend sollen die Ergebnisse des Aufbaus der Integrationsarchitektur diskutiert werden. Dabei sollen in einem ersten Schritt die Erkenntnisse und Herausforderungen der Implementierung der Integrationsarchitektur dargelegt werden. Daraufhin wird eine kritische Überprüfung der Integrationsarchitektur hinsichtlich der Umsetzung der in Abschnitt 3.3 definierten Anforderungen folgen.

Im Folgenden sollen die wesentlichen Ergebnisse und Erkenntnisse dieser Arbeit beschrieben werden.
Die vorliegende Arbeit zeigt, dass es möglich ist eine flexible Integrationsarchitektur für Grafana zu entwickeln, die es erlaubt beliebige React-Anwendungen in Form eines Plugins in Grafana einzubinden.
Weiterhin ist die Integrationsarchitektur durch ihre Modularität und den Verzicht auf eine zu starke Kopplung an Grafana, so flexibel gestaltet worden, dass eine Weiterverwendung der vorliegenden Architektur für ähnlich gelagerter Integrationsvorhaben wie zum Beispiel die Integration des Prototyps in das ITOA-System Kibana mit einigen kleineren Änderungen möglich ist.
Die modulare Gestaltung der Integrationsarchitektur erleichtert außerdem die Erweiterung der Architektur um zusätzliche Funktionalitäten. Dies konnte unter anderem durch eine hohe Kohäsion der Komponenten und einer geringen Kopplung zwischen den Komponenten der Integrationsarchitektur erreicht werden.
Außerdem ermöglicht die Datenanbindung der vorliegenden Integrationsarchitektur, die flexible Anbindung unterschiedlicher Datenquellen mittels eines auf die jeweilige Datenquelle zugeschnittenen Services, solange dieser Service das Interface `DataSource` (vgl. Abbildung 4.3) implementiert.
Des Weiteren ermöglicht die fertige Integrationsarchitektur die Anbindung von Datenstrukturen beliebiger Gestalt über einen austauschbaren Adapter, welcher die Rohdaten so transformieren kann, dass sie dem für die 3D-Visualisierung notwendigen Interface `TimeSeries` (vgl. Abbildung 3.7) entsprechen.
Abschließend ist als weiteres Ergebnis dieser Arbeit die verbesserte Benutzerzufriedenheit bezüglich der Performanz des vorliegenden Systems zu nennen. Diese konnte im Rahmen einer Evaluation festgestellt werden, bei welcher die Teilnehmer dem fertigen System deutlich kürzere Antwortzeiten gegenüber dem ursprünglichen Prototypen bescheinigten.

Zu Beginn dieser Arbeit wurde in Abschnitt 1.2 das Ziel eines Aufbau einer Integrationsarchitektur ausgegeben. Diese sollte dazu in der Lage sein, den Prototyp (vgl. Abschnitt 3.1) in das ITOA-System Grafana zu integrieren. Gleichzeitig sollte eine für den Anwender zufriedenstellende Performanz und die Erweiterbarkeit der fertigen Architektur sichergestellt werden. Die oben beschriebenen Ergebnisse zeigen, dass alle Ziele dieser Arbeit mit der fertigen Integrationsarchitektur erreicht werden konnten.

Während der Entwicklung des Grafana-Plugin zeigten sich an vielen Stellen die Probleme einer sich noch im Alpha-Status befindlichen Plugin-Schnittstelle.
Hierbei stellte sich insbesondere das Fehlen einer vollständigen und aktuellen Dokumentation als große Herausforderung dar. Viele Dokumentationsartikel und Blog-Artikel erwiesen sich als veraltet, oder zu oberflächlich in Bezug auf den Inhalt. Insbesondere komplexere Anwendungsfälle werden von der derzeit verfügbaren Dokumentation nicht abgedeckt.
Ein weiteres Problem sind die unterschiedlichen Plugin-Schnittstellen für React und Angular in Grafana. Viele Funktionalitäten, welche in der Plugin-Schnittstelle für Angular zur Verfügung stehen, fehlen oder sind bisher nur teilweise in der Plugin-Schnittstelle für React umgesetzt worden.
Die oben genannten Herausforderungen sorgten während der Entwicklung teilweise für erhebliche Verzögerungen, da teilweise manuell der Programmcode Grafanas und die Bibliotheken zur Erweiterung Grafanas nach potentiell verwertbaren Funktionen durchsucht und analysiert werden musste. Außerdem machten Inkonsistenzen zwischen der Schnittstellenbeschreibung in den Dokumentationsartikeln und der tatsächlichen Implementierung der Schnittstellen in den Bibliotheken, eine effiziente Arbeitsweise schwer.

Nachdem in Abschnitt 3.3 die Anforderungen an die Integrationsarchitektur definiert wurden, soll im Folgenden eine Überprüfung des fertigen Systems hinsichtlich der Erfüllung dieser Anforderungen stattfinden.

[R01] Erweiterbarkeit
Die Nutzung der Plugin-Schnittstelle Grafanas und der weitgehende Verzicht auf direkte Abhängigkeiten zu Grafana sorgen für eine lose Kopplung zwischen den Komponenten der Integrationsarchitektur und Grafana selbst. Hierdurch wird nicht nur eine funktionale Erweiterbarkeit der vorliegenden Architektur erreicht, sondern auch eine potentielle Portierung der Architektur auf ähnlich gelagerte Anwendungsfälle, wie zum Beispiel die Integration im ITOA-System Kibana ermöglicht.

[R02] Einheitliche Benutzeroberfläche
Das Entfernen des CSS-Frameworks Bootstrap und die Verwendung von Komponenten aus der Grafana-Bibliothek `@grafana/ui` und Grafanas CSS-Klassen stattdessen,

sorgt für eine einheitliche und konsistente Benutzeroberfläche im Grafana-Plugin (siehe Abbildung 4.9 und 4.6).

[R03] Einführung eines Persistenzmechanismus
Die fertige Integrationsarchitektur verfügt über einen Persistenzmechanismus, der in der Lage ist, alle Einstellungsdaten des Plugins permanent zu speichern. Dabei wurde die bestehende Konfigurationsdatenbank auf dem Grafana-Server und die Anbindung über die Plugin-Schnittstelle einer aufwendigen Eigenentwicklung vorgezogen (siehe Abschnitt 4.3.3).

[R04] Stabilität gegenüber Updates
Der konsequente Verzicht auf Modifikationen im Programmcode Grafanas und die gleichzeitige Nutzung der Schnittstellen zur Erweiterung Grafanas machen die vorliegende Integrationsarchitektur äußerst stabil gegenüber zukünftigen Updates und Upgrades von Seiten Grafanas.

[R05] Flexible Datenanbindung
Währenddessen im Prototyp nur das DBMS Solr als Datenquelle zur Verfügung stand, so ist es in der fertigen Integrationsarchitektur möglich beliebige Datenbanksysteme anzubinden. Dies wird durch die neu entwickelte Datenanbindung ermöglicht (siehe Abschnitt 4.2).

[R06] Dynamischer Wechsel von Kennzahlen und Aggregationstypen
Das Plugin ermöglicht den dynamischen Wechsel zwischen den verfügbaren Kennzahlen und die dynamische Auswahl aus einer Liste verschiedener Aggregationstypen aus der Benutzoberfläche heraus. Die Kennzahlen lassen sich mittels eines Adapters festlegen (siehe Abschnitt 4.2).

[R07] Funktionsumfang
Alle Funktionen des Prototyps finden sich auch im fertigen Plugin wieder. Zusätzlich wurden Funktionalitäten wie zum Beispiel die dynamische Auswahl zwischen den verfügbaren Kennzahlen hinzugefügt.

[R08] Performanz
Die Performanz des fertigen Systems hat sich gegenüber dem Prototyp spürbar verbessert. Dies liegt unter anderem an der Möglichkeit zur Selektion der vom Server kommenden Zeitreihendaten durch die Benutzeroberfläche Grafanas (vgl. Abbildung 4.7). Der Prototyp verfügte über keine solche Funktionalität und lud deshalb immer alle verfügbaren Daten vom Solr-Server, was insbesondere bei großen Datenmengen zu spürbaren Verzögerungen beim Start der Anwendung geführt hat.
Ein weiterer Grund für die Performanzverbesserung ist die Auslagerung von Berechnungen auf die Datenbankserver, soweit dies das jeweilige DBMS unterstützt.

Im vorliegenden Fall wurde die Berechnung von Aggregierungen, wie Minimum, Maximum und Mittelwert auf den Solr-Server ausgelagert.

[R09] Bereitstellung
Die fertige Integrationsarchitektur verwendet viele Komponenten und Module aus dem Prototypen wieder, oder erweitert diese. Hierdurch konnte die geforderte Wiederverwendung bestehender Komponenten aus dem Prototypen erreicht werden.
Außerdem verfügt die Integrationsarchitektur und die darin enthaltenen Komponenten über eine eigene Build-Pipeline, welche die zur Bereitstellung notwendigen Schritte, automatisch durchführt.
Um die Bereitstellung des fertigen Systems für Anwender zu erleichtern, wurde eine kurze Anleitung verfasst, welche die einzelnen zur Installation des Plugins notwendigen Schritte verständlich erklärt.

Die vorausgehende Evaluation der geforderten Anforderungen hinsichtlich ihrer jeweiligen Umsetzung in der fertigen Integrationsarchitektur zeigt, dass alle Anforderungen hinreichend erfüllt wurden.

6 Zusammenfassung und Ausblick

Das vorliegende Kapitel soll einen inhaltlichen Überblick über diese Bachelorarbeit bieten. Des Weiteren soll dem Leser im Rahmen eines Ausblicks ein Eindruck über Potentiale und Erweiterungsmöglichkeiten der fertigen Integrationsarchitektur vermittelt werden.

6.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde eine Integrationsarchitektur, die die Integration einer React-Anwendung in das ITOA-System Grafana ermöglicht, entwickelt.

Nachdem in Kapitel 1 dem Leser ein kurzer Überblick über die Ziele und die Motivation der Arbeit gegeben wurde, konnte im darauffolgenden Kapitel 2 eine Vermittlung der, für das Verständnis der folgenden Kapitel notwendigen, theoretischen Grundlagen erfolgen.

Im Anschluss an Kapitel 2 folgte in Kapitel 3 eine Analyse der beiden an der Integration beteiligten Anwendungen. Hierbei wurde sowohl für den Prototyp als auch Grafana die jeweilige Architektur, Benutzeroberfläche und Datenanbindung auf integrationsrelevante Aspekte hin untersucht. Weiterhin wurde Grafana auf potentielle Schnittstellen, welche zur Anbindung des Prototyps dienen könnten, hin analysiert. Außerdem wurden die Schwächen des Prototyps herausgearbeitet, um sie im Rahmen der Umsetzung der Integrationsarchitektur soweit wie möglich eliminieren zu können. Das Kapitel wurde durch eine abschließende Anforderungsanalyse für die Integrationsarchitektur komplementiert. Dabei wurden unter Berücksichtigung der Ergebnisse der vorausgehenden Analyse die Anforderungen an die zu implementierende Integrationsarchitektur systematisch erarbeitet.

Das Kapitel 4.6 begann mit einer Erläuterung von wichtigen Entscheidungen, welche zu Beginn der Umsetzung der Integrationsarchitektur getroffen wurden. Dies umfasste unter anderem die Entscheidung auf welcher Ebene der Applikationsintegration (vgl. Abbildung 2.3) die Integration der beiden Anwendungen erfolgen sollte. Daraufhin wurde dem Leser ein Überblick über die fertige Architektur und die darin enthaltenen Komponenten geboten. Dabei erfolgte auch eine Dokumentation

zentraler und wichtiger Architekturentscheidungen. Im restlichen Teil des Kapitels wurde den einzelnen Komponenten der Architektur jeweils ein eigener Abschnitt gewidmet, in welchem die Implementierung der Komponenten beschrieben wird.

Die Ergebnisse dieser Arbeit wurden in Kapitel 5 dokumentiert. Hierbei wurden auch die Herausforderungen, welche während der Umsetzung der Integrationsarchitektur auftraten, dargelegt. Daraufhin folgte eine Überprüfung der in Abschnitt 3.3 definierten Anforderungen an die fertige Integrationsarchitektur. Dabei zeigte sich, dass die fertige Integrationsarchitektur im Wesentlichen alle geforderten Anforderungen erfüllt. Weiterhin konnte festgestellt werden, dass die zu Beginn dieser Arbeit definierten Ziele (vgl. Abschnitt 1.2) vollumfänglich erreicht wurden.

6.2 Ausblick

Im Zuge der Umsetzung der vorliegenden Integrationsarchitektur entwickelten sich neue Anregungen und potentielle zukünftige Erweiterungsmöglichkeiten, welche leider im Rahmen dieser Arbeit nicht bearbeitet werden konnten. Diese sollen nachfolgend kurz vorgestellt und erläutert werden.

Eine Erweiterungsmöglichkeit besteht in der Portierung der vorliegenden Integrationsarchitektur auf das ITOA-System Kibana. Kibana verfügt analog zu Grafana ebenfalls über eine Schnittstelle für Plugins. Diese Schnittstelle unterstützt unter anderem Anwendungen, welche mithilfe von React entwickelt worden sind. Damit könnten die React-Anwendung des Plugins und die restlichen Komponenten der Integrationsarchitektur bis auf kleinere Anpassungen wiederverwendet werden.

Ein weiterer potentieller Anwendungsfall für die 3D-Visualisierung neben der Visualisierung von Kennzahlen, welche die IT-Infrastruktur betreffen, könnte die Visualisierung von Kennzahlen der Anwendungen sein, welche auf den einzelnen Instanzen laufen. Ein Beispiel hierfür ist die Visualisierung der durchschnittlich zu erwartenden Wartezeiten in Warteschlangensystemen wie RabbitMQ[1]. Weiterhin könnte die integrierte Vorhersagefunktionalität der Integrationsarchitektur bei der Vorhersage von zu erwartenden Wartezeiten behilflich sein.

Mit der neuen Integrationsarchitektur ist es möglich mithilfe eines Interface beliebige Datenquellen zur Architektur hinzufügen. Hierzu muss jeweils nur ein Service erstellt werden, der das Interface `DataSourceService` (vgl. Abbildung 4.3) implementiert. Derzeit verfügt das entwickelte Grafana-Plugin lediglich über einen Service, der die Anbindung an das DBMS Solr ermöglicht. Dieser Umstand ließe sich im Zuge der

[1] https://www.rabbitmq.com/ (besucht am 15. Feb. 2020)

Anbindung weiterer Datenquellen und durch die Entwicklung korrespondierender Services ändern.

Literatur

[Bin15] Jeremy Bingham. *Where and Why We Use Go*. 2015. URL: https://grafana.com/blog/2015/08/21/where-and-why-we-use-go/ (besucht am 3. Nov. 2019) (zitiert auf Seite 20).

[Chi+19] Alina Chircu, Eldar Sultanow, David Baum, Christian Koch und Matthias Seßler. „Visualization and Machine Learning for Data Center Management“. In: *INFORMATIK 2019: 50 Jahre Gesellschaft für Informatik – Informatik für Gesellschaft (Workshop-Beiträge)*. Hrsg. von Claude Draude, Martin Lange und Bernhard Sick. Bonn: Gesellschaft für Informatik e.V., 2019, S. 23–35 (zitiert auf den Seiten 1, 2, 9, 21, 27).

[Cli05] Andrew Clifford. *Minimal integration 7: point-to-point, hub, or bus?* 2005. URL: https://it.toolbox.com/blogs/andrewclifford/minimal-integration-7-point-to-point-hub-or-bus-111405 (besucht am 19. Okt. 2019) (zitiert auf Seite 6).

[Cor19] Mozilla Corporation. *Cross-Origin Resource Sharing (CORS)*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS (besucht am 15. Okt. 2019) (zitiert auf Seite 22).

[Dan+07] Florian Daniel, Jin Yu, Boualem Benatallah et al. „Understanding UI Integration: A Survey of Problems, Technologies, and Opportunities“. In: *IEEE Internet Computing* 11.3 (Mai 2007), S. 59–66 (zitiert auf Seite 8).

[Dir18] J. Dirksen. *Learn Three.js - Third Edition*. Packt Publishing, 2018, S. 1 (zitiert auf Seite 12).

[DJ19] Martin Drasar und Tomas Jirsik. „IT Operations Analytics: Root Cause Analysis via Complex Event Processing“. In: *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. Apr. 2019, S. 741–742 (zitiert auf den Seiten 1, 5).

[Ell14] Stephen Elliot. *DevOps and the Cost of Downtime: Fortune 1000 Best Practice Metrics Quantified*. International Data Corporation (IDC), 2014 (zitiert auf Seite 1).

[FP14] Michael Fauscette und Randy Perry. *Simplifying IT to Drive Better Business Outcomes and Improved ROI: Introducing the IT Complexity Index*. International Data Corporation (IDC), 2014 (zitiert auf Seite 2).

[Gam+95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995, S. 139, 315 (zitiert auf den Seiten 35, 38).

[Gle05] Rodney Gleghorn. „Enterprise application integration: a manager's perspective". In: *IT Professional* 7.6 (Nov. 2005), S. 17–23 (zitiert auf Seite 6).

[Gro17] Peter Groucutt. *BA outage demonstrates the importance of understanding the true cost of IT downtime*. 2017. URL: https://www.cbronline.com/enterprise-it/ba-outage-understanding-true-cost-downtime/ (besucht am 5. Okt. 2019) (zitiert auf Seite 1).

[He+16] Huihong He, HaibinZhai, Qian Liu und Yong Wang. „Using Object-Oriented Big Data Analytics to Reveal Server Performance Dead Zone". In: *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. Bd. 1. Juni 2016, S. 619–624 (zitiert auf Seite 5).

[Hol19] Peter Holmberg. *Writing React Plugins*. 2019. URL: https://grafana.com/blog/2019/03/26/writing-react-plugins/ (besucht am 13. Okt. 2019) (zitiert auf Seite 21).

[Inc19] Facebook Inc. *Thinking in React*. 2019. URL: https://reactjs.org/docs/thinking-in-react.html (besucht am 17. Dez. 2019) (zitiert auf Seite 11).

[ISO11] ISO. *ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. Standard. Geneva, CH: International Organization for Standardization, 2011 (zitiert auf Seite 11).

[Lab19a] Grafana Labs. *Backend Plugins*. 2019. URL: https://grafana.com/docs/grafana/latest/plugins/developing/backend-plugins-guide/ (besucht am 15. Nov. 2019) (zitiert auf Seite 23).

[Lab19b] Grafana Labs. *Basic Concepts*. 2019. URL: https://grafana.com/docs/guides/basic_concepts/ (besucht am 2. Nov. 2019) (zitiert auf den Seiten 19, 21).

[Lab19c] Grafana Labs. *Configuration*. 2019. URL: https://grafana.com/docs/grafana/latest/installation/configuration/ (besucht am 28. Jan. 2020) (zitiert auf Seite 52).

[Lab19d] Grafana Labs. *Dashboard JSON*. 2019. URL: https://grafana.com/docs/grafana/latest/reference/dashboard/ (besucht am 6. Feb. 2020) (zitiert auf Seite 22).

[Lab19e] Grafana Labs. *Data Sources*. 2019. URL: https://grafana.com/docs/features/datasources/ (besucht am 11. Okt. 2019) (zitiert auf den Seiten 20, 21).

[Lab19f] Grafana Labs. *Data Sources*. 2019. URL: https://grafana.com/docs/grafana/latest/plugins/developing/datasources/ (besucht am 15. Nov. 2019) (zitiert auf Seite 23).

[Lab19g] Grafana Labs. *Developer Guide*. 2019. URL: https://grafana.com/docs/grafana/latest/plugins/developing/development/ (besucht am 13. Dez. 2019) (zitiert auf den Seiten 22, 23, 29).

[Lab19h] Grafana Labs. *Getting Started*. 2019. URL: https://grafana.com/docs/grafana/latest/guides/getting_started/ (besucht am 12. Dez. 2019) (zitiert auf Seite 20).

[Lab19i] Grafana Labs. *Grafana Features*. 2019. URL: https://grafana.com/grafana/ (besucht am 10. Okt. 2019) (zitiert auf den Seiten 23, 29).

[Lee16] Daniel Lee. *Timing is Everything. Writing the Clock Panel Plugin for Grafana 3.0*. 2016. URL: https://grafana.com/blog/2016/04/08/timing-is-everything.-writing-the-clock-panel-plugin-for-grafana-3.0/ (besucht am 20. Nov. 2019) (zitiert auf Seite 23).

[Lui14] James V. Luisi. „Part III - Information Systems“. In: *Pragmatic Enterprise Architecture*. Hrsg. von James V. Luisi. Boston: Morgan Kaufmann, 2014, S. 160–161 (zitiert auf den Seiten 5–7).

[Mah07] Zaigham Mahmood. „Service Oriented Architecture: Potential Benefits and Challenges“. In: *Proceedings of the 11th WSEAS International Conference on Computers*. ICCOMP'07. Agios Nikolaos, Crete Island, Greece: World Scientific, Engineering Academy und Society (WSEAS), 2007, S. 497–501 (zitiert auf Seite 7).

[Maj19] Wojciech Maj. *React Lifecycle Methods diagram*. 2019. URL: http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/ (besucht am 15. Jan. 2020) (zitiert auf Seite 50).

[Mar03] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003 (zitiert auf Seite 11).

[Miy+19] Takaya Miyazawa, Hiroaki Harai, Yusuke Yokota und Yasushi Naruse. „Sparse Regression Model to Predict a Server Load for Dynamic Adjustments of Server Resources“. In: *2019 22nd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. Feb. 2019, S. 249–256 (zitiert auf Seite 1).

[Öde18] Torkel Ödegaard. *Frontend: Code structure, architecture and plans for the future*. 2018. URL: https://github.com/grafana/grafana/wiki/Frontend:-Code-structure,-architecture-and-plans-for-the-future (besucht am 25. Okt. 2019) (zitiert auf den Seiten 20, 21, 49).

[PEM03] Frauke Paetsch, Armin Eberlein und Frank Maurer. „Requirements engineering and agile software development“. In: *WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003.* Juni 2003, S. 308–313 (zitiert auf Seite 23).

[Sch13] Susanne Schölzel. *Anwendung der Stadt-Metapher zur Visualisierung von Organisationsstrukturen am Beispiel der BTU Cottbus*. 2013, S. 42 (zitiert auf Seite 1).

[SQL19] SQLite. *About SQLite*. 2019. URL: https://www.sqlite.org/about.html (besucht am 11. Feb. 2020) (zitiert auf Seite 52).

[Tro+04] David Trowbridge, Ulrich Roxburgh, Gregor Hohpe, Dragos Manolescu und E.G. Nadhan. *Integration Patterns*. Microsoft Corporation, 2004, S. 241 (zitiert auf Seite 6).

[W3C19a] World Wide Web Consortium (W3C). *Inline frames: the IFRAME element*. 2019. URL: https://www.w3.org/TR/html4/present/frames.html#h-16.5 (besucht am 18. Dez. 2019) (zitiert auf Seite 27).

[W3C19b] World Wide Web Consortium (W3C). *Introduction to style sheets*. 2019. URL: https://www.w3.org/TR/html401/present/styles.html#h-14.1 (besucht am 22. Dez. 2019) (zitiert auf Seite 43).

[Wei+14] Wei Jiang, Meng Zhang, Bin Zhou, Yujian Jiang und Yingwei Zhang. „Responsive web design mode and application“. In: *2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA)*. Sep. 2014, S. 1303–1306 (zitiert auf Seite 44).