HAW HAMBURG

Bachelorarbeit

Abdul Basit Andar

CQRS und Schichtenarchitekturen – ein Vergleich

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

Abdul Basit Andar

CQRS und Schichtenarchitekturen – ein Vergleich

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Wirtschaftsinformatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 29. März 2023

Abdul Basit Andar

Thema der Arbeit

CQRS und Schichtenarchitekturen – ein Vergleich

Stichworte

CQRS, Schichtenarchitektur, Vergleich, Softwarearchitektur

Kurzzusammenfassung

Die vorliegende Bachelorarbeit setzt zwei verschiedene Ansätze von Systemarchitekturen in Vergleich. Hierbei werden zunächst alle relevanten sowie grundlegenden Informationen zusammengetragen. Daraufhin sollen beide Architekturen mit vorab festgesetzten Kriterien verglichen werden. Der Vergleich soll Aufschluss darüber geben, welche der Architekturen für welchen Anwendungsfall geeignet oder ungeeignet sind...

Abdul Basit Andar

Title of Thesis

CQRS and multitier architectures – a comparison

Keywords

CQRS, multitier architectures, comparison, softwarearchitecture

Abstract

This bachelor thesis compares two different approaches of system architectures. First, all relevant and basic information are collected. Thereupon, both architectures are to be compared with criteria fixed before. The comparison should provide information about which of the architectures is suitable or unsuitable for which use case...

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

»Design and programming are human activities; forget that and all is lost. «
- Bjarne Stroustrup

Es gibt zahlreiche Zitate zu Softwarearchitekturen, die viele verschiedene Blickwinkel aufzeigen. Das oben genannte Zitat beschreibt die Wichtigkeit des menschlichen Denkprozesses bei der Entwicklung von Software. Stroustrup geht so weit, dass er das absolute Scheitern als Konsequenz benennt. Eines der komplexesten Konstrukte, die Menschen erfunden haben sind Softwaresysteme. Aufgrund dieser Komplexität liegt die Schlussfolgerung nahe, dass viele Softwareprodukte scheitern, veralten oder aus Angst nicht angerührt werden. Doch auch das Gegenteil ist der Fall: viele Projekte sind erfolgreich, Erweiterungen und Fehlerbehebungen sind umsetzbar und all dies einhergehend mit einem realistischem Zeitaufwand. Viele Faktoren wirken sowohl beim Erfolg als auch beim Scheitern eines Softwaresystems mit. All dies wird meist in Dokumentationen festgehalten, welche definitiv vorhanden sein müssen, um eine gute Umsetzung der Architektur in Aussicht zu haben. [7] Eine Architektur ist nicht das Produkt selbst, sondern eher der Plan, der umgesetzt werden soll. Ein solcher Plan ist derweil notwendig, da Software über die Jahre zunehmend an Komplexität gewonnen hat. Zum gekonnten Umgang mit ebendieser Komplexität bedarf es wiederum einer Architektur. Eine Architektur entsteht aus Entscheidungen, die in einem Projektverlauf getroffen werden. Die wichtigsten Kriterien müssen abgewogen werden, um eine saubere Architektur zu definieren. Zu jeder guten Architektur findet sich eine zugehörige Dokumentation, die beschreibt, welche Mittel genutzt werden, um die Zielsetzung mit einem Softwaresystem zu erreichen. Zu verstehen, was Architektur ist und mit welchen Sichten auf Architekturen gearbeitet wird, entscheidet, wie das Endprodukt aussieht. Folglich sollte ebenfalls der Stil der Architektur richtig gewählt werden. [10] Da jede Problemstellung zudem individuell ist, sollte ferner jede Architektur individuell angepasst werden.

1.1 Problemstellung

Dem Titel der Arbeit ist dessen Gegenstand zu entnehmen: Die Gegenüberstellung von zwei Architekturen. Als Resultat wird erhofft, einen signifikanten Unterschied erkennen zu können. Hierbei wird eine im Unternehmen entwickelte Anwendung herangezogen, die aktuell das Schichtenmodel umsetzt. Diese Anwendung soll so umgestellt werden, dass CQRS als Architektur umgesetzt wird. Die Anwendung ist für die Verwaltung von Kunden, Artikeln und Bestellungen sowie für alle Prozesse, die dazugehören, geschrieben. Der Umfang und das Potential der Anwendung wurden unterschätzt, wodurch die Planung der Architektur im aktuell bekannten Architekturstil umgesetzt wurde. Um eine tatsächliche Veränderung feststellen zu können, soll anhand bestimmter Kriterien eine mögliche Verbesserung festgehalten werden. Dafür müssen Messungen durchgeführt und analysiert werden.

1.2 Zielsetzung

Mithilfe der Arbeit soll die Erkenntnis gezogen werden, ob eine der beiden Architekturen tatsächlich besser sein kann als die andere. Um diese Erkenntnis jedoch gewinnen zu können, ist es elementar, zunächst ein grundlegendes Verständnis für Softwarearchitektur aufzubauen. Ohne dieses Verständnis kann insoweit der Vergleich nicht nachvollzogen werden. Die Arbeit soll überdies die vorhandenen Möglichkeiten zur Umsetzung der Architektur darlegen. Demnach soll Aufschluss darüber gegeben werden, ob es lohnenswert ist, eine Anwendung in einen anderen Architekturstil zu überführen. Abschließend soll ein Vergleich gezogen werden, um ein Urteil treffen zu können. Damit dies jedoch möglich ist, muss die Architektur sauber überführt werden. Dies wird zugleich im Rahmen der Arbeit dokumentiert und erläutert.

1.3 Aufbau

Die Bachelorarbeit setzt sich aus vier Kapiteln zusammen. In Kapitel 1 wird die Arbeit allgemein vorgestellt und ein Überblick über den Aufbau und die Themen geboten. Daraufhin führt Kapitel 2 die wichtigsten Begriffe, Definitionen, Konzepte und Technologien ein, welche für diese Arbeit von Belang sind. Ferner werden die zu vergleichenden Architekturen eingeführt und ihre Charakteristiken sowie das herangezogene Kriterium zum

Ziehen eines Vergleichs benannt. Anschließend stellt Kapitel 3 die Anwendung vor, die in beiden Architekturstilen verglichen wird, die Problematik der aktuellen Architektur und wie diese mit einem anderen Architekturstil gelöst werden soll. Mithilfe des festgesetzten Kriteriums wird untersucht, ob die Problematik gelöst werden kann. Zuletzt werden in Kapitel 4 die Ergebnisse zusammengefasst, ein Ausblick darüber gegeben, wie diese Arbeit weitergeführt werden könnte und welche Rolle die Ergebnisse in zukünftigen Projekten spielen können.

2 Grundlagen der Softwarearchitektur

In diesem Kapitel sollen grundlegende Begrifflichkeiten, die für das Verstehen der vorliegenden Arbeit notwendig sind, aufgegriffen und erläutert werden. Jedoch sollen nicht nur Begriffsdefinitionen eingeführt sondern auch Konzepte herangezogen werden, die das Verstehen der vorliegenden Bachelorarbeit erleichtert. Folglich wird mit dem erläutern des Begriffs Softwarearchitektur begonnen und es wird abgeschlossen mit Konzeptionen, die ein Überschaubares Bild über das Spektrum der Arbeit bieten soll.

2.1 Was ist eine Architektur?

Für den Begriff Softwarearchitektur ist eine Reihe von Definitionen anzutreffen. Einige Definitionen beschränken sich auf die Struktur eines Softwaresystems.

»Softwarearchitektur ist die Struktur eines Software-Produkts. Diese umfasst Elemente, die extern wahrnehmbaren Eigenschaften der Elemente und die Beziehungen zwischen den Elementen. [7]«

Andere Definitionen beziehen sich wiederum auf die Entscheidungen, die getroffen wurden, um ein System zu entwerfen.

»Softwarearchitektur $= \sum$ *aller wichtigen Entscheidungen*

Wichtige Entscheidungen sind alle Entscheidungen die im Verlauf der weiteren Entwicklung nur schwer zu ändern sind. [7]«

Dadurch, dass die Architektur im Zusammenhang mit Software noch nicht allzu lange existiert, gibt es verschiedene Vorstellungen. Verschiedenste Definitionen geben widersprüchliche Vorstellungen darüber, was unter dem Begriff zu verstehen ist. [14] Auch wird Architektur als beschreibendes Mittel für die Strukturen eines Systems, dessen Bausteine und Schnittstellen genutzt, wobei das Zusammenspiel dieser betrachtet wird. [10] Zu erkennen ist, dass die verschiedenen Definitionen jeweils verschiedene Betrachtungsweisen haben. Der Begriff Softwarearchitektur setzt sich aus den Begriffen Software und Architektur zusammen. Der Begriff Software wird laut Oxford Languages wie folgt definiert:

»Alle nicht technisch-physikalischen Funktionsbestandteile eines Computers, insbesondere Computerprogramme.«

Der Begriff Architektur entstammt dem griechischen Wort: *arkhitekton*. Die Erläuterung beschreibt den Prozess sowie das Produkt der Planung, des Entwurfs und der Konstruktion von Gebäuden und anderen Dingen.

In der IT wird der Begriff Architektur oftmals in folgenden Bereichen genutzt:[11]

- Gesamtorganisation von Softwaresystemen
- Gesamtorganisation von Kombinationen aus Hardware und Software
- Gesamtorganisation von der Informationstechnologie innerhalb von Organisationen (Unternehmens-IT-Architektur)
- Beliebige technische Dokumentation

Der Fokus der vorliegenden Arbeit soll jedoch auf den ersten zwei Punkten liegen. Wenn all die genannten Informationen in Betracht gezogen werden, so scheint die folgende Definition für den Begriff Softwarearchitektur angemessen:

»Softwarearchitektur: die grundlegende Organisation eines Systems, die in seinen Komponenten, ihren Beziehungen zueinander und zur Umgebung sowie den Grundsätzen für seinen Entwurf und seine Entwicklung zum Ausdruck kommt.«
– Institute of Electrical and Electronics Engineers

Aus dieser und den vorher genannten Definitionen wird ersichtlich, wie wichtig und einflussreich eine Softwarearchitektur ist. Dabei handelt es sich hierbei um weitaus mehr als nur Software, nämlich um jegliche Komponenten, die in Zusammenhang stehen, auf welche Art und Weise diese im Zusammenhang miteinander stehen und welche Grundsätze für den Entwurf genutzt wurden.

2.2 Dokumentation einer Architektur

»Unter Dokumentation versteht man die Nutzbarmachung von Informationen zur weiteren Verwendung. Ziel der Dokumentation ist es, schriftlich oder auf andere Weise dauerhaft niedergelegte Informationen gezielt auffindbar zu machen.«
Wikipedia

Damit eine Architektur die Chance hat, wie vorgesehen umgesetzt zu werden, bedarf es einer Architekturdokumentation. Es genügt nicht, wenn diese lediglich existent ist, vielmehr muss diese zugleich explizit kommuniziert werden. Insoweit ist es nicht hilfreich, wenn der Architekt seine Gedanken nicht äußert. Dies ist mit dem Fall vergleichbar, dass eine Architektur nur niedergeschrieben ist und nie genutzt wird. Die Bedeutsamkeit einer Architektur kommt ferner darin zum Ausdruck, dass bestimmte Erkenntnisse erst zur Erstellung der Dokumentation gezogen werden, die daraufhin für eine Architektur förderlich sind. Als Schlüsselrolle für eine Dokumentation gilt die Kommunikation. Ein Architekt kommuniziert mithilfe mehrerer stilistischer Mittel, wie z. B. Präsentationen, Reviews und Schulungen. Auf unmittelbare oder mittelbare Weise resultiert dies in einer Dokumentation. Die Zielsetzungen dabei sind:[14]

- Entscheidungen vollständig und unmissverständlich festhalten
- Architektur auf die verschiedenen Zielgruppen abstimmen und kommunizieren

Für eine Architektur werden Richtlinien festgehalten, damit Qualität, Korrektheit der Umsetzung und Kommunikation gewährleistet sind. Die Richtlinien beschränken sich auf zwei Bereiche:[14]

- Wie Architektur-Mittel zu verwenden sind.
- Wie Architektur zu dokumentieren ist.

Mittel und Richtlinien sind etwa Namenskonventionen, festgelegte Programmiersprache und Verwendung von UML-Diagrammen. [14]

Richtlinien einer Dokumentation

Auch für die Erstellung einer Dokumentation werden Richtlinien festgelegt. [14] Hierbei werden verschiedene Aspekte abgedeckt, welche für die Dokumentation der Architektur vonnöten sind, wie z.B. der Inhalt, die Struktur und die Organisation einer Dokumentation. Wie eine Dokumentation aktuell gehalten wird, die Regeln für einen Schreibstil und viele weitere Punkte werden festgelegt. Bei Bedarf ist es dem Architekten zudem möglich, weitere Kriterien hinzuzufügen. [14]

2.3 Eigenschaften einer guten Architektur

Architekturen können theoretisch jegliche Form annehmen. Meist ist eine Architektur durch viele Umstände beeinflusst. Dementsprechend ist es wichtig, einige Punkte zu benennen, die Architekturen charakterisieren. Architektur macht die Komplexität überschaubar und handhabbar, indem sie die wesentlichen Aspekte eines Systems zeigt. Gleichzeitig wird auf Details bewusst nicht zu sehr geachtet, um in relativ kurzer Zeit einen Überblick über ein System zu erlangen. [14] Eine Architektur sollte mithin auch ‚sauber' sein. Folgende Kriterien muss eine Architektur erfüllen, um als ‚sauber' zu gelten: [10]

- relevant
- effizient pflegbar
- sparsam
- verständlich und nachvollziehbar
- korrekt und aktuell
- prüfbar
- akzeptiert bei Lesern

Relevant

Eine Architekturdokumentation soll nur die relevanten Informationen eines Systems beinhalten, welche für die Architektur eines Systems relevant sind. [10] Welche Informationen jedoch als relevant gelten, unterliegt der Entscheidung des Architekten. [10]

- Systemspezifische Strukturen und Entscheidungen
- Strukturen bzw. Entscheidungen, welche für die Implementation und spätere Änderungen bedeutsam sind
- Unerwartete oder unkonventionelle Lösungsansätze
- Wiederholt auftretende Lösungsmuster, Aspekte oder technische Konzepte

Was jedoch eine Architekturdokumentation ist und wie diese aussieht, wird im späteren Verlauf der Arbeit genau erläutert.

Effizient pflegbar

Der Pflegeaufwand einer Dokumentation skaliert mit dem Umfang einer Dokumentation. Folglich ist es um so wichtiger, dass eine Dokumentation verständlich, genau und aktuell ist. Es gibt mehrere Faktoren, welche Pflegbarkeit beeinflussen:[10]

- Umfang (Menge): Dokumentationen, die kürzer sind, sind entsprechend simpler zu pflegen.
- Redundanz: Wiederholungen vermeiden Vor- und Rücksprünge, erhöhen jedoch den Aufwand zur Pflege.
- Standardisierte Strukturen
- Eingesetzte Werkzeuge: Eine Werkzeugkette, die in den Arbeitsprozess integriert ist, kann den Pflegeaufwand verringern.

Als Ansatz sollte generell eine standardisierte Gliederungsstruktur eingehalten werden, was allen beteiligten Lesern das Zurechtfinden innerhalb der Dokumentation erleichtert. Überdies ist es ratsam, die Menge an Details zu reduzieren, um eine Überschaubarkeit zu bewahren. Weniger ist oftmals mehr.[10]

Sparsam

»So viel wie nötig, so wenig wie möglich. [10]«

Das oben genannte Zitat soll die Herangehensweise bei der Beschreibung von Softwarearchitekturen darstellen. Grundsätzlich beginnt die Beschreibung mit einem Überblick und sollte nur nach Bedarf Details aufzeigen. Es ist zu überlegen, für wen wie viele Informationen in der Architekturdokumentation enthalten sein müssen. Je kompakter die Architekturdokumentation ausfällt, desto besser. Hierbei soll nicht das Fehlverständnis aufkommen, dass Informationen ausgelassen werden können, im Gegenteil: es sollten alle nötigen Informationen enthalten sein.

Verständlich und nachvollziehbar

Neben der Erstellung von Systemen und deren Tests werden ebenfalls der Betrieb und die Weiterentwicklung positiv beeinflusst, wenn eine Dokumentation verständlich und nachvollziehbar ist. Dieser Einfluss wirkt sich auf die Kosten und Risiken aus. Das schnelle und zuverlässige Finden soll angestrebt werden. Es hilft den Lesern, praktisch mit der Dokumentation zu arbeiten und den größten Nutzen daraus zu ziehen. Verständlichkeit und Nachvollziehbarkeit sollten am besten von Lesern beurteilt werden, welche potentielle Verbesserungsvorschläge liefern können. [10]

Korrekt und aktuell

Alle Informationen müssen korrekt sein, da alle Beteiligten, die sich der Dokumentation bedienen, sich auf diese verlassen. Mit der Entwicklung des Systems muss zugleich die Dokumentation synchron aktualisiert werden. Ist dies nicht der Fall, können Änderungen am System nicht nachvollzogen werden und die Dokumentation verliert ihre Daseinsberechtigung. Korrektheit beschränkt sich hierbei auf die Vollständigkeit der Informationen gemäß dem Informationsbedarf und zielt nicht darauf ab, möglichst viel zu beschreiben. [10]

Prüfbar

Risiken und Probleme sollten möglichst schnell identifiziert werden. Um dies zu erreichen, sollte eine Dokumentation die Prüfbarkeit der gewählten Lösung hinsichtlich der gestellten Anforderung unterstützen. [10] Es müssen Prüfkriterien und Bewertungsszenarien in einer Dokumentation festgehalten werden, um eine Prüfbarkeit zu gewährleisten. Hierbei helfen ebenso die verschiedenen Architektursichten. [10] Was es mit Architektursichten auf sich hat, wird im späteren Verlauf der Arbeit erläutert.

Akzeptiert bei Lesern

Nur die Leser können bewerten, ob eine Dokumentation qualitativ und nützlich ist. Die Leser sind in die verschiedensten Rollen aufgeteilt, was zugleich den Grund für die unterschiedlichen Erwartungshaltungen darstellt. Demnach ist es wichtig, die Leser zu befragen, um die subjektiven Meinungen einzuholen. Durch die Sammlung an Antworten ist es daraufhin möglich, in zukünftigen Iterationen auf die Bedürfnisse der Leser einzugehen. Es sollte für die Leser geschrieben werden, nicht für einen selbst. [10]

2.3.1 Bestandteile einer Architektur

Systeme bestehen in ihrer statischen Struktur aus einer Menge von Bausteinen, welche in Beziehungen zueinander stehen. Beim Entwerfen eines Systems müssen diese Bausteine und ihre Beziehungen geplant werden. Der Begriff „Baustein“ soll hierbei als abstrahierendes Mittel dienen, um zwischen der Vielzahl an Begriffen aus Programmiersprachen und Designmethoden zu unterscheiden. Bausteine erstrecken sich von kleinen bis hin zu mittleren und großen Bausteinen. Beispiele für kleine Bausteine sind Funktionen und Unterprogramme. Bei mittelgroßen Bausteinen handelt es sich wiederum um Klassen und Module. Große Bausteine könnten Subsysteme, Libraries bis hin zu Schichten sein. Alle Bausteine werden als Bestandteil des Systems betrachtet, haben einen Namen und weisen idealtypische Eigenschaften einer Blackbox auf. [10]

2.4 Sichten auf Softwarearchitekturen

»Eine Sicht stellt ein vollständiges System aus dem Blickwinkel einer Menge von zusammenhängenden Interessen heraus dar – Institute of Electrical and Electronics Engineers«

So wie unterschiedliche Gebäude in der Architektur unterschiedliche Baupläne haben, so sind auch Softwaresysteme verschieden aufgebaut. Beim Kauf und Bau von Häusern wird meist der Grundriss des Hauses betrachtet. Hingegen werden bei bestimmten anderen Installationen Pläne für Heizungs- und Wasserleitungen herangezogen. Dies kann analog auf Softwarearchitekturen übertragen werden: Sind es mithin bei Bauten die Pläne, so sind es bei den Softwarearchitekturen die Sichten. Die Idee von verschiedenen Sichten stammt von Phillipe Kruchten. Mithilfe der verschiedenen Sichten sollen diverse Blickwinkel auf ein System gewährleistet werden. Jede Sicht soll bestimmte Aspekte in den Vordergrund rücken. Vorgaben darüber, welche Sichten zuerst erstellt werden sollen, gibt es nicht. [10]

Diese Sichten sollen dabei helfen, bestimmte Aspekte einer Architektur in den Vordergrund zu stellen. Es wird zwischen drei Sichten unterschieden:[10]

- Bausteinsicht
- Verteilungssicht
- Laufzeitsicht

Die Sichten sollten in Verbindung miteinander betrachtet werden. Wie die Sichten miteinander zusammenhängen, ist der Abbildung 2.1 zu entnehmen. Oftmals wird auch *Logical view* oder auch Systemkontext als eine Sicht benannt. [4]

2.4.1 Systemkontext

Als etwas übergeordnete Sicht zeigt der Systemkontext, wie das System in einer Umgebung eingegliedert ist. Es wird von oberhalb betrachtet, um zu erkennen, welche Abgrenzungen zu anliegenden Systemen herrschen. Dadurch werden Probleme bezüglich Schnittstellen und Infrastrukturen identifiziert. Der Systemkontext wird häufig in der Kommunikation mit Stakeholdern verwendet, die keinen technischen Hintergrund haben.

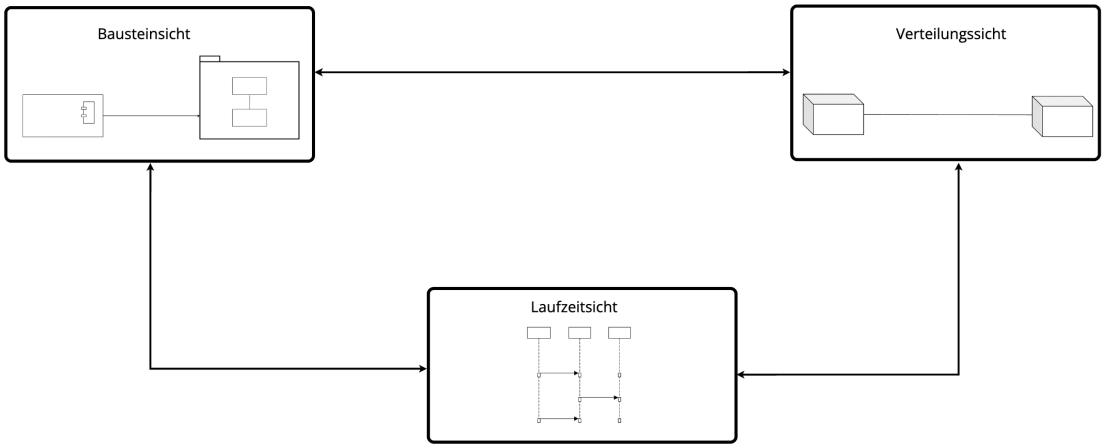

Abbildung 2.1: Sichten in Verbindung mit einander [10]

[4] Das System wird auf einer Ebene abstrahiert, die das Verstehen der Systemarchitektur vereinfacht. Ein gutes Beispiel dafür ist das UML Komponenten-Diagramm. Andere Diagramme sind selbstverständlich ebenfalls in der Lage dies umzusetzen.[4] Eine Visualisierung eines Beispiels ist in Abbildung 2.2 zu sehen.

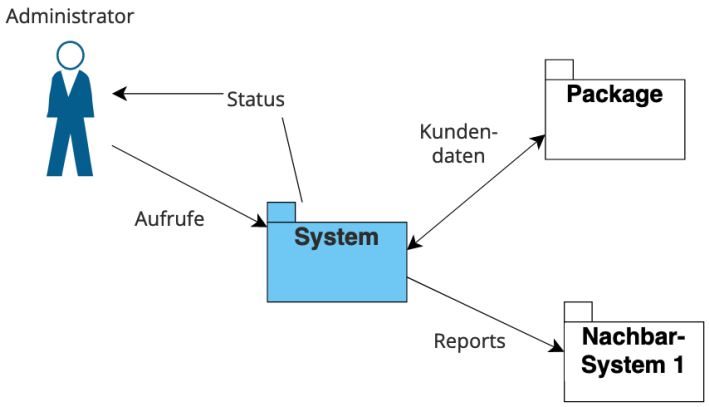

Abbildung 2.2: Visualisierung der Kontextabgrenzung [10]

2.4.2 Bausteinsicht

Die Bausteinsicht wird als Grundrissplan der Softwarearchitektur erachtet, da sie die statische Struktur des Systems beschreibt. Der Aufbau aus Softwarebausteinen und ihren Beziehungen wird hierbei beschrieben. Jeder dieser Bausteine ist durch Quellcode umgesetzt. Als Beispiel für solche Bausteine sind Klassen aus Programmiersprachen, Packages, Datenbanken oder Shell-Skripte zu nennen. Die Bausteinsicht beschreibt, wie die Konstruktion aus sogenannten Bausteinen das gesamte System ergibt. Zugleich werden die einzelnen fachlichen und technischen Eigenschaften eines jeden Bausteins betrachtet. [10]

Für die Darstellung wird gerne ein Top-down-Ansatz verfolgt und diese in Verfeinerungsstufen abstrahiert. In diesen Stufen werden die Bausteine in kleinere Bausteine zerlegt, bis alle relevanten Details aufgeführt sind. Hierbei werden stilistische Mittel aus UML genutzt, wie z. B. Pakete, Symbole für Klassen als auch Komponenten. [4]

2.4.3 Verteilungssicht

Mit der Verteilungssicht wird die Distribution von Systembestandteilen in Hardware- und Softwareumgebungen verdeutlicht. Hierbei wird gezeigt, welche Teile des Systems sich auf welchen Rechnern, Standorten oder Umgebungen befinden. [10] Die Sicht kann einer Landkarte gleichgesetzt werden, in der alle Komponente und Protokolle der Hardware aufgezeigt werden. [4] Sollte ein System nur auf einem Endgerät installiert werden, kann auf diese Sicht verzichtet werden. Bei komplexeren Systemen mit mehr Endgeräten klärt sie jedoch die standortbezogenen Fragen, wie z. B. auf welche Hardware eine Software oder an welchem Standort ausgeführt werden soll. [10]

2.4.4 Laufzeitsicht

Die Laufzeitsicht beschreibt, wie der Name bereits ausdrückt, das Verhalten der Bausteine im System zur Laufzeit. Genauer ausgedrückt, beschreibt die Laufzeitsicht das Verhalten der Bausteine untereinander und zu angrenzenden Systemen. Ferner werden wichtige Funktionalitäten und beispielhafte Abläufe gezeigt. Darüber hinaus wird ersichtlich, wie Bausteine des Systems wesentliche Funktionen erledigen. [10]

2.5 Auswirkung mangelhafter Umsetzung von Softwarearchitekturen

Wirtschaftliche Einbußen sind ein klarer Hinweis dafür, dass Anstrengungen in eine Architektur erbracht werden sollten. Der Einfluss einer Architektur auf die Entwicklung ist derweil enorm. Vernachlässigungen und eine inkonsistente Umsetzung einer Architektur münden unweigerlich in negativen Einflüssen auf ein System. *Architecture Erosion* und *Technical Debt* sind Begriffe, ddie es hierbei zu erwähnen gilt, da sie dafür verantwortlich sind. Beide Begriffe müssen präzise definiert werden, damit die Folgen nachvollziehbar sind und Vorgehensweisen zur Vorbeugung besser verstanden werden. [9] Ein System, welches den Einflüssen der genannten Phänomene unterliegt, wird den Zustand erreichen, in dem es kaum noch wartbar ist. Ein solches System wird auch *Fossil System* genannt, System genannt, ein System, welches kaum wartbar ist und dessen Erweiterung wirtschaftlich nicht rentabel ist. Ab diesem Zeitpunkt steht ein Unternehmen am Scheideweg. Entweder werden ganze Teile des Systems ausgetauscht oder ein komplett neues System muss angeschafft werden. Damit dies verhindert wird und ein System insgesamt erhalten bleibt, darf zu keinem Zeitpunkt *Architecture Erosion* oder *Technical Debt* Platz in der Entwicklung finden. [9]

2.5.1 Architecture Erosion – Architektonische Erosion

»Violations of a system's architecture that leads to significant problems in the system and contributes to its increasing brittleness. - IGI Global«

Als Architektonische Erosion werden Verstöße gegen die Architektur eines Systems bezeichnet, die erhebliche Probleme in einem System hervorrufen und zu dessen Anfälligkeit beitragen. Es ist das Management der Architektur, welches zur Folge haben kann, dass ein System im Prozess der Architektonischen Erosion zum Zerfall kommt. Ein Beispiel dafür ist ein Team, in dem jedes Individuum an einem anderen Baustein einer Architektur arbeitet. Jedes dieser Individuen setzt jedoch seine eigenen Vorstellungen der gesamten Architektur um. Eine solche unstrukturierte Arbeitsweise wird zwangsläufig dazu führen, dass die Wertigkeit eines Systems abnimmt und der finanzielle Aufwand steigt. [9] Dennoch ist es nicht die absichtliche Achtlosigkeit von Architektur und Management, die im Wertverlust des Systems resultiert. Häufig sind es die unbewussten Entscheidungen und Umstände, die im Entwicklungsprozess dazu führen.

Gleichzeitig ist jedoch zu erwähnen, dass Architektonische Erosion nicht von Personen, die am System arbeiten, verursacht wird. Dies geschieht gänzlich von allein. Im zeitlichen Verlauf ändern sich Umgebung und Umgang des Systems. Funktionen veralten, das Verhalten von Nutzern wird sich verändern, Funktionalitäten werden ausgetauscht oder neuere Techniken sind bereits auf dem Markt. Diese Aspekte führen dazu, dass einst gut modellierte Architektur zwar weiterhin funktionstüchtig ist, jedoch weit abgestuft wird. Es bedarf der Remodellierung oder des Austauschs der Architektur, um den Nutzen des Systems zu wahren. *Architecture Erosion* wird durch äußere Umstände ausgelöst. [9] Entwicklerteams müssen dauerhaft gegen diese Probleme, ankämpfen damit das Warten dieser Systeme nicht ermüdend wird. Jedoch sind auch hierbei die Kosten zu berücksichtigen. [7]

2.5.2 Technical Debt - Technische Kosten

Technische Kosten sind das Resultat der Abweichungen einer Architektur. Präziser ausgedrückt ist damit gemeint, dass andere Lösungen eingebaut werden als diejenigen, welche in einer Architekturdokumentation festgehalten wurden. Dabei wird ein Schaden angerichtet, der im Nachhinein durch großen Aufwand behoben werden muss. Die Schadensbemessung erweist sich dabei als mühsam, die technische Kosten einer Ansammlung von abstrakten Fällen abdeckt. Diese Schuld entsteht immer dann, wenn Abkürzungen genommen werden, wenn ein System entwickelt wird. Bis diese Schuld nicht bereinigt wird, werden immer mehr Erschwernisse bei der Weiterentwicklung eines Systems entstehen. Das Aufschieben von Aufgaben aufgrund von vermeintlichem Zeitmangel stellt hierbei ein typisches Beispiel dar, sei es auch, wenn nur ein Teil der gesamten Forderung umgesetzt wird. Beide genannten Szenarien führen zur Fehleranfälligkeit, Komplexität, Erschwernis in der Wartbarkeit und Entwicklung des Systems. Im Zuge dessen entstehen erhebliche Kosten. [7]

2.6 Architektur-Stile

Architekturstile werden nach Shaw und Garlan als ein Muster der strukturellen Organisation einer Familie von Systemen definiert. Ein Architekturstil spiegelt die grundlegende

Struktur von Softwaresystemen und ihren Eigenschaften wider. Demnach wird sich ebenfalls einem Architekturstil bedient, um Architekturen zu kategorisieren. Demzufolge kann ein Stil dafür genutzt werden, um Varianten und Konsequenzen einer

fundamentalen Architektur zu verstehen. In dieser Arbeit werden lediglich die Stile benannt, welche für die praktische Erarbeitung notwendig sind. [14]

2.7 Schichtenarchitektur

Einer der am meisten genutzten Architekturstile ist die Schichtenarchitektur. Ausschließlich die Schichtenarchitektur definiert Schichten als Mittel für eine Architektur. Schichten sind ein zusammenfassendes Element. Bausteine eines Softwaresystems werden als größere Einheiten zusammengefasst. Folglich sind Schichten zugleich Bausteine. Sie werden horizontal und hierarchisch sortiert. Die festgelegte Regel ist hierbei, dass weiter unten liegende Schichten nicht auf höher liegende zugreifen dürfen. Als erstes Softwaresystem, welches Schichten beinhaltete, galt das THE-System von Dijkstra. In den aktuellen Schichtenarchitekturen existieren zwei Dimensionen: die technische und die fachliche Schichtung. [7]

2.7.1 Technische Schichtung

Bei der technischen Schichtung werden Bausteine nach ihren technischen Funktonen zugeordnet und hierarchisiert. Dadurch entsteht ein System, welches den technischen Aufbau eines Systems widerspiegelt. [7] In Abbildung 2.3 ist zu sehen, wie eine solche Schichtung ungefähr zu verstehen ist.

2.7.2 Fachliche Schichtung

Bei Systemen, die größere Dimensionen annehmen, reicht eine technische Schichtung nicht aus. Orientierungslosigkeit macht sich breit bei Systemen, deren Grenzen für die technische Sichtung überschritten wurden. Orientierungslosigkeit macht sich breit bei Systemen, deren Grenzen für die technische Sichtung überschritten haben. Ab einer gewissen Größe des Systems ist es für Architekten und Entwickler selbstverständlich, die fachliche Schichtung als Mittel heranzuziehen. Zusätzlich zur horizontalen Hierarchie der

Abbildung 2.3: Visualisierung der Technischen Schichtung [7]

technischen Sichtung wird die fachliche Sichtung in vertikale Module gespalten. Diese liegen quer über den Modulen der technischen Schicht. In jedem fachlichen Modul ist ein Teil der technischen Schicht. Ein System, das die fachliche Schichtung implementiert, wird langlebig sein. Wie die fachliche Schichtung praktisch aussieht, ist in der Abbildung2.4 zu entnehmen. [7]

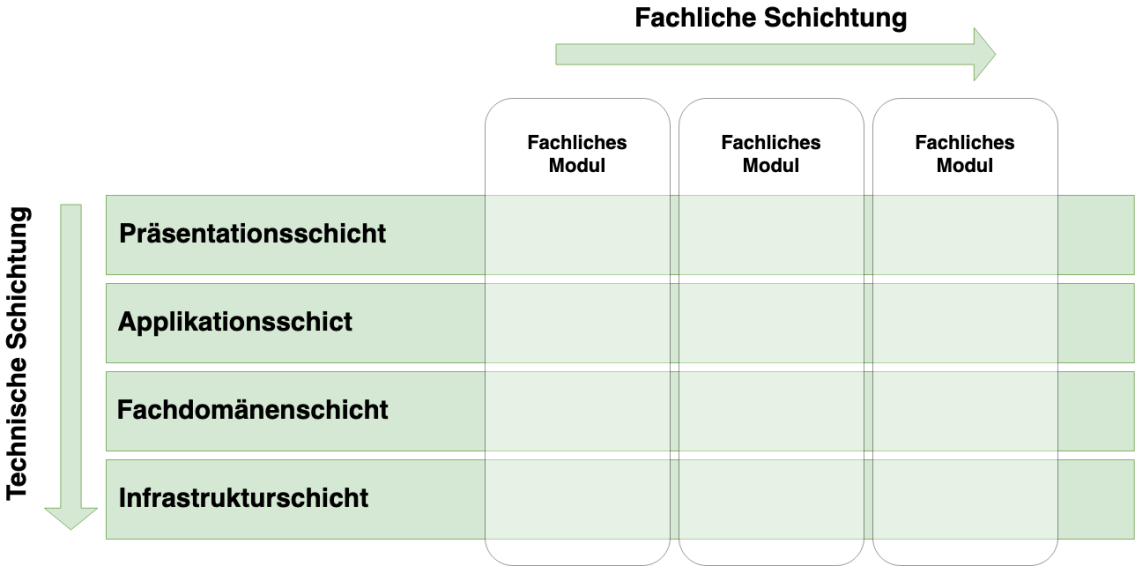

Abbildung 2.4: Visualisierung der Fachlichen Schichtung [7]

2.8 CQRS

Ein weiterer Stil für eine Systemarchitektur ist CQRS. CQRS steht für »Command and Query Responsibility Segregation« bzw. für die Trennung von Befehlen und Abfragen. Dieser Architekturstil zielt darauf ab, dass Operationen, welche den Zustand von Daten ändern, von den lesenden Operationen separiert werden. Lesende und schreibende Modelle werden voneinander abgegrenzt. *»Command Query Separation «* (CQS) wurde erstmals von Bertrand Meyer in seinem Buch *»Objektorientierte Softwareentwicklung «* erwähnt. Der Grundgedanke ist hierbei, dass alle Systemvorgänge absolut voneinander getrennt sind: [2]

- Abfragen: Geben als Rückmeldung ein Ergebnis wobei der Zustand des Systems unverändert bleibt.
- Befehle: Verändern den Zustand eines Systems.

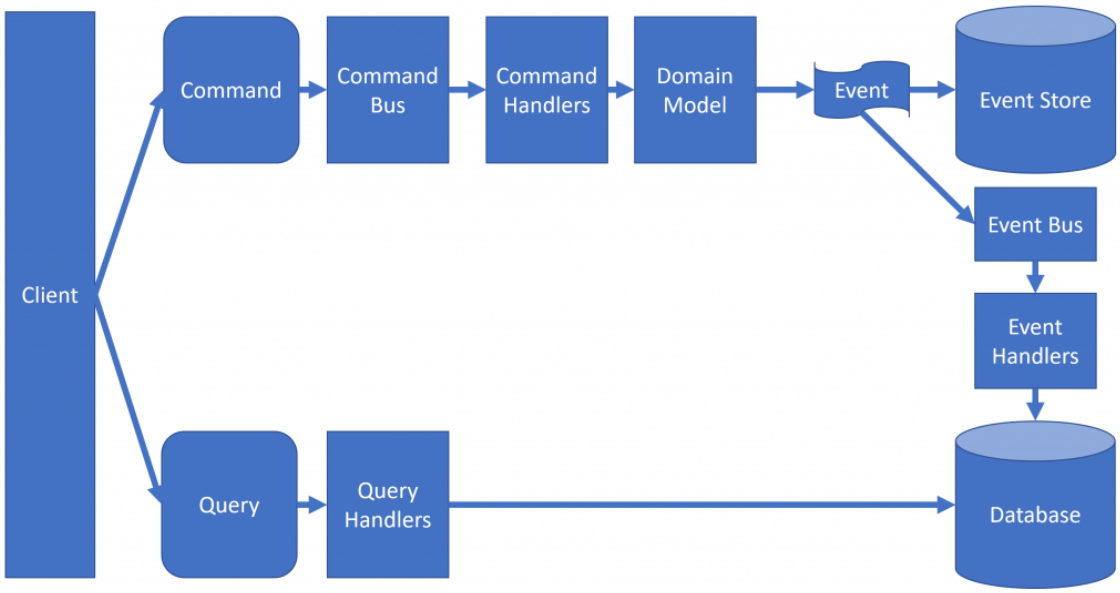

Abbildung 2.5: Komplexe Architektur, die CQRS beschreiben soll [2]

»Jede Methode sollte entweder ein Befehl sein, der eine Aktion ausführt, oder eine Abfrage, die Daten an den Aufrufer zurückgibt, aber nicht beides. «
– Bertrand Meyer

CQRS geht einen Schritt weiter als CQS, indem es Befehle und Abfragen in verschiedene Objekte trennt. Es wird oftmals angenommen, dass CQRS sehr komplex und schwierig ist. Ein Grund dafür könnte sein, dass Diagramme wie solche in Abbildung 2.5 es komplex erscheinen lassen. [2] Wenn über CQRS gesprochen wird, dann werden zumeist andere Stile und Technologien mit benannt. Gleichzeitig wird CQRS auch ein *Ermöglicher* oder *Enabler* genannt. Grund dafür ist die erleichterte Verwendung von anderen Mustern die oft gut zusammenpassen. Dementsprechend CQRS und Event Sourcing häufig gemeinsam eingesetzt. [2] Es ist jedoch kein Event Sourcing und setzt das Event Sourcing nicht voraus. CQRS eignet sich gut und ermöglicht, Event Sourcing anzuwenden.[2]

2.8.1 Anwendungsfall

Ist von einer bestimmten Architektur die Rede, sollte ferner der richtige Anwendungsfall bekannt sein bzw. wann eine Architektur ausgewählt werden sollte. Wie jedes Muster ist auch CQRS an einigen Stellen nützlich, während es an anderen nicht sinnvoll ist. Viele Systeme passen in ein CRUD-Modell und sollten mithin in diesem Stil durchgeführt werden. CQRS ist ein erheblicher Gedankensprung und sollte folglich nur genutzt werden, wenn der Nutzen den Sprung wert ist. Es existieren zwar erfolgreiche Anwendungen von CQRS, welche in den meisten Fällen jedoch nicht so gut waren. In diesen Fällen wurde CQRS als ein signifikanter Grund dafür benannt, dass ein System in schwerwiegende Schwierigkeiten geriet. Mithilfe von CQRS lassen sich möglicherweise einige komplexe Einsatzbereiche leichter bewältigen. Jedoch muss betont werden, dass eine solche Eignung für CQRS eher die Minderheit darstellt. In der Regel gibt es genügend Überschneidungen zwischen Befehls- und Abfrageseite, so dass die gemeinsame Nutzung eines Modells sich einfacher gestaltet. [5]

Die Verwendung von CQRS in einem Bereich, der nicht dafür geeignet ist, erhöht die Komplexität und damit die Produktivität und das Risiko. Ein anderer besonderer Vorteil liegt in der Handhabung von Hochleistungsanwendungen. Die Last von Lese- und Schreibvorgängen ist getrennt und kann unabhängig voneinander skaliert werden. Wenn eine Anwendung eine große Diskrepanz zwischen Lese- und Schreibvorgängen aufweist, ist dies sehr praktisch. Auch ohne dies können unterschiedliche Optimierungsstrategien auf beiden Seiten angewandt werden. CQRS verwendet für alle Abfragen ein eigenes Modell. Trotz dieser Vorteile ist die Verwendung von CQRS mit Vorsicht zu genießen. Viele Informationssysteme passen gut zu einem Konzept einer Informationsbasis, die auf

gleiche Weise aktualisiert wird, wie sie gelesen wird. Das Hinzufügen von CQRS an ein solches System kann eine erhebliche Komplexität zur Folge haben. [5]

2.8.2 DDD – Domain-Driven Design

Wenn über die praktische Umsetzung von CQRS gesprochen wird, so fällt oft der Begriff Domain-Driven Design, ein Ansatz zur Softwareentwicklung, der sich auf die Modellierung von Geschäftsszenarien konzentriert. [3] Er wurde von Eric Evans in seinem Buch „Domain-Driven Design: Tackling Complexity in the Heart of Software“ (2003) eingeführt und hat sich seither als wichtiger Ansatz für die Entwicklung komplexer Softwaresysteme etabliert. DDD konzentriert sich darauf, die Sprache und die Modelle des Geschäftssystems zu verstehen und diese in die Softwarearchitektur und -entwicklung zu übertragen. [3] Durch den Einsatz von DDD kann ein besseres Verständnis des Geschäftssystems erreicht werden, was wiederum zu einer besseren Abbildung des Geschäftssystems in der Software führt. [5] Ein wichtiger Bestandteil von DDD ist die Verwendung von Ubiquitous Language, einer gemeinsamen Sprache zwischen Geschäftsexperten und Entwicklern, die sicherstellt, dass das Geschäftssystem und die Software konsistent modelliert werden [3]. Weitere wichtige Konzepte von DDD sind Bounded Context, Aggregate und Entity.[12] DDD wird oft in Verbindung mit anderen Technologien wie Microservices und Command Query Responsibility Segregation (CQRS) verwendet. [5]

2.9 Bemessungsgrundlage

2.9.1 Performance als Qualitätsmerkmal

Dieser Abschnitt dient der Erläuterung des Begriffs Performance. Im Englischen wird der Ausdruck Performance laut dem Cambridge Dictionary vielfach ausgelegt. Für die vorliegende Arbeit ist die Bedeutung Leistung von Interesse und somit einschlägig. Im Deutschen kann es als Performanz übersetzt werden. Dennoch wird im weiteren Verlauf das englische Wort *Performance* genutzt.

»Die Performance eines Softwaresystems ist dessen messbare Leistungsfähigkeit und beinhaltet dessen Zeit- und Verbrauchsverhalten«[8]

Das Zeitverhalten beschreibt die Antwort- und Verarbeitungszeit, wobei die Antwortzeit als die Zeit zwischen Eingang der Ereignisse und die Reaktion eines Systems beschrieben wird. [8] Verdeutlicht an einem bildlichen Beispiel, ist es die Zeit zwischen dem Öffnen bzw. Laden einer Datei bis hin zu dem Zeitpunkt, an dem diese Datei dargestellt wird.

Die Menge der benötigten Ressourcen für eine Ausführung ist das Verbrauchsverhalten. Verbrauchs- und Zeitverhalten sind in Abhängigkeit voneinander zu betrachten, denn je mehr Ressourcen benötigt werden als eigentlich zur Verfügung stehen, desto negativer wirkt sich dies auf das Zeitverhalten aus. Resümierend ist Performance ein Qualitätsmerkmal für Softwaresysteme. [8]

Da im Rahmen dieser Arbeit zwei Architekturstile miteinander verglichen werden, die bei der gleichen Anwendung umgesetzt worden sind, wird hier die Performance zugleich als das wichtigste Qualitätsmerkmal herangezogen. Denn es soll ersichtlich werden, welche der beiden Umsetzungen für einen bestimmten Anwendungsfall geeigneter ist.

2.9.2 Apache JMeter

Die Frage, die sich nach letztem Abschnitt stellt, lautet: Wie soll Performance praktisch gemessen werden? JMeter von Apache bietet sich diesbezüglich als ein hilfreiches Tool an. Mit diesem Tool ist es möglich, zahlreiche Performance-Tests durchzuführen. Hierbei wird das Verhalten auf Antwortzeiten geprüft. Dies bedeutet, dass geprüft werden kann, wie lange es dauert, bis eine Anwendung auf eine Anfrage antwortet. Zum besseren Verständnis dessen stehen grafische Repräsentationen zur Verfügung, die Messdaten einfach widerspiegeln. Das besondere hierbei ist ferner, dass JMeter die Möglichkeit von angepassten Testplänen bietet. Ein weiteres interessantes Merkmal ist, dass JMeter sich nicht ausschließlich auf HTTP und HTTPS beschränkt, sondern auch die Möglichkeit für REST bietet. [13]

Dies kommt dem Testen der Performance hinsichtlich des Vergleiches der Arbeit zugute. Denn die Anwendung bietet, wie bereits erwähnt, REST-Schnittstellen an. Diese können zur Durchführung der Tests genutzt werden, um ein möglichst präzises und realitätsnahes Ergebnis zu erzielen. Wie ein solches Ergebnis aussieht, wird im weiteren Verlauf der Arbeit visualisiert.

2.9.3 Lambdas

Damit die Performance nicht nur getestet wird, werden in diesem Abschnitt Lambdas als ein Konzept eingeführt, welches zur Verbesserung zur Performance beitragen soll. Der Datenverarbeitungsservice bietet die Möglichkeit, Code auszuführen, ohne selbstständig Server bereitzustellen bzw. zu verwalten. Der Code wird mithilfe von Lambdas auf einer Recheninfrastruktur ausgeführt, welche Server und Betriebssysteme wartet und Kapazitäten nicht nur bereitstellt, sondern auch automatisch skaliert. Funktionen werden nur dann ausgeführt, wenn diese wirklich gebraucht werden. Dadurch ist es möglich, z. B. einige wenige tägliche oder tausende Anfragen in Sekunden zu verarbeiten. Aus Sicht der Kosten ist der Vorteil gegeben, dass nur ein ausgeführter Code bezahlt werden muss. Die Verantwortlichkeit hinsichtlich der Ressourcen übernimmt Lambda in Gänze. So ist es möglich, dass Programmierer sich ausschließlich um die Korrektheit des Codes bemühen müssen. Infolgedessen ist es möglich, dass die Performance gesteigert wird und dies automatisiert erfolgt, sofern Lambdas richtig eingesetzt werden. [1]

3 Optimierung einer Schichtenarchitektur mit CQRS

Bevor diese Arbeit auf den direkten Vergleich beider Architekturstile eingeht, muss zunächst die genutzte Anwendung erläutert werden. Diese ist eine bestehende Anwendung mit einem Schichtenmodell und soll in CQRS überführt werden. Der Name der Anwendung wird nicht benannt, da hierfür keine Relevanz vorherrscht. Das Konzept der Anwendung hingegen wird erläutert, da dies die Wahl des Architekturstiles beeinflusst. Es werden zum Vergleich festgesetzte Kriterien herangezogen, die vorher erklärt werden.

3.1 Die Anwendung

Für jedes Problem, dem ein Mensch begegnet, sucht er eine Lösung. In der heutigen Zeit, in welcher der technologische Fortschritt so weit ist wie nie zuvor, gibt es eine Reihe technischer Möglichkeiten, die zuvor genannten Probleme zu lösen. Die Anwendung, welche für diese Arbeit herangezogen wird, soll eines der menschlichen Alltagsprobleme lösen. Bei der Idee handelt es sich um folgende: Jeder Haushalt muss für den eigenen Erhalt einkaufen. Um dies zu erleichtern, soll eine Anwendung den Einkauf digitalisieren. Der Anwender nutzt die Anwendung, um aus dem Bestand eines regionalen Supermarktes seinen digitalen Einkaufswagen zu füllen und diesen kostenpflichtig zu bestellen. Auf der anderen Seite der Anwendung empfangen Supermarkt und ein Lieferfahrer die entsandte Bestellung. Diese wird vorbereitet und durch einen Lieferfahrer an den Anwender geliefert. Dem Kunden ist es möglich, sich seinen Supermarkt selbst auszusuchen und den aktuellen Warenbestand einzusehen. Dem Supermarkt ist es dadurch möglich, mehr Ware mit geringem Aufwand zu verkaufen.

3.2 Aktueller Zustand der Architektur (Schichtenmodel)

Wie zuvor erwähnt, sind Architekturen vielfältig. Infolgedessen kann es der Fall sein, dass die Überschaubarkeit abhandenkommt, sodass es umso wichtiger ist, Architekturen explizit zu beschreiben.

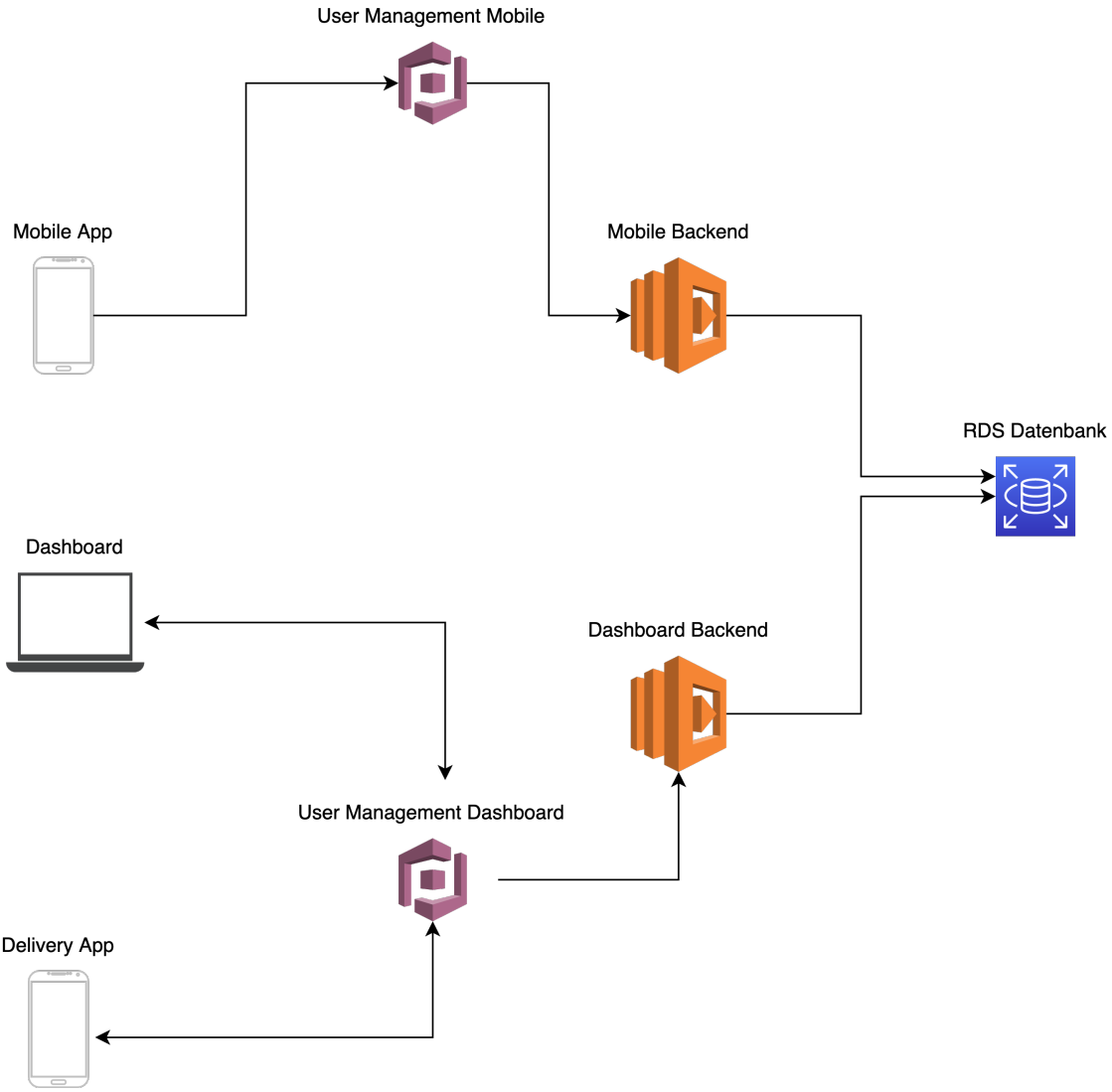

Abbildung 3.1: Architektur der Anwendung

In Abbildung 3.1 ist der aktuelle Zustand der Architektur illustriert. Zu sehen sind verschiedene Objekte, die irgendwie miteinander in Verbindung stehen. Folgend werden alle

Objekte und ihre Verbindungen beschrieben. Die Beschreibung soll helfen, ein besseres Gefühl für die Architektur zu entwickeln und die Unterschiede für den späteren Verlauf klar dazustellen. Die Arbeit fokussiert sich auf das Mobile und Dashboard Backend.

3.2.1 Backend

Sowohl Mobile als auch Dashboard Backend sind Server, welche jegliche Anfragen der Anwendungen abarbeiten. Beide Backend Server sind gleich aufgebaut. Die jeweiligen Backend-Anwendungen bestehen aus einem Monolithen und teilen sich nicht in Microservices auf. Diese werden jeweils auf einem Server ausgeführt.

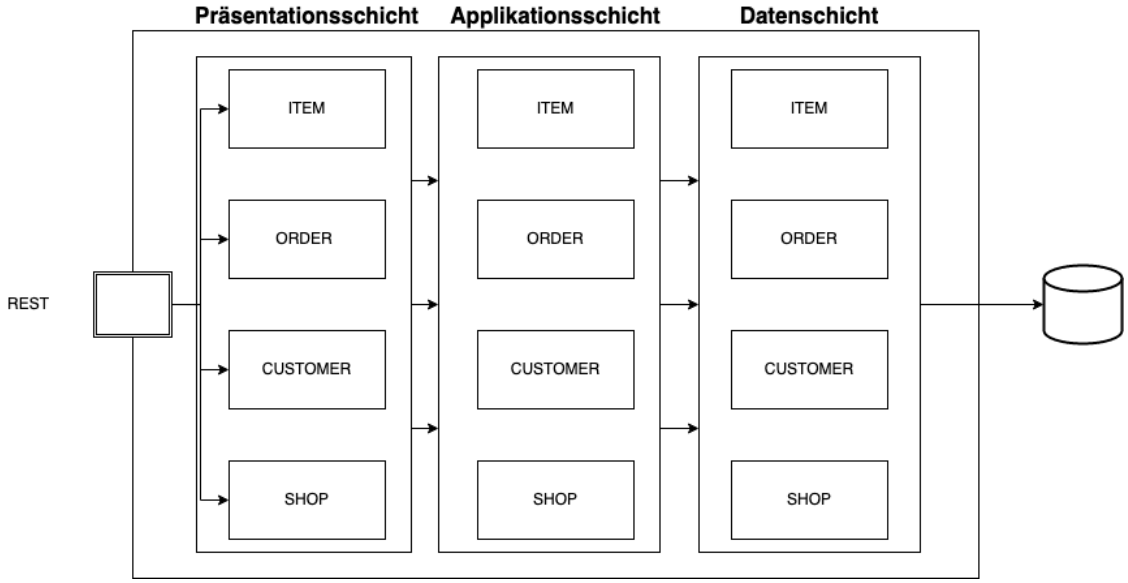

Abbildung 3.2: Abbildung des Mobile Backend

Der Aufbau dieser Server ist in Abbildung 3.2 ersichtlich. Die erste Schicht in der vorhandenen Schichtenarchitektur ist die Präsentationsschicht. Diese bietet nach außen hin eine REST-Schnittstelle, über die mit dem System kommuniziert werden kann. Es folgt die Business-Schicht, welche die fachliche Logik des Systems enthält und über die Datenschicht mit der Datenbank kommuniziert. Die Datenschicht stellt die bidirektionale Kommunikation zur Datenbank sicher. Zusätzlich enthält die Datenschicht die Anwendungsdomäne. Die Abhängigkeiten der jeweiligen Schichten verlaufen von der Präsentationsschicht zur Anwendungsschicht hin zur Datenschicht. Abhängigkeiten in gegensätzlicher Richtung existieren nicht.

Ein detaillierter Blick in die Architektur offenbart, dass die Operationen, sowohl lesende als auch schreibende, auf einem Server ausgeführt werden. Eine zu hohe Last von lesenden Anfragen würde die Performance des Servers gänzlich beeinträchtigen. Dies gilt es zu ändern. Doch bevor dies geschieht, soll geprüft werden, wie performant bzw. leistungsfähig die aktuelle Architektur ist.

3.2.2 Probleme beim aktuellen Zustand

Damit die angestrebte Veränderung besser verstanden wird, muss der aktuelle Zustand besser nachvollzogen werden. Aus diesem Grund wird in diesem Abschnitt das Problem mit der aktuellen Architektur näher betrachtet. Es geht nicht primär darum, was der Code im Sinne der Anwendungslogik tut, sondern darum, wie sich die Anwendung in der aktuellen Architektur verhält. Demnach wird der Programmiercode vorliegend nicht aufgegriffen.

Die Anwendung bietet viele Endpunkte an, welche alle auf gleichem Wege ihre Daten beziehen bzw. diese verändern. Die meisten Endpunkte bieten lesende Operationen an. Das Problem, welches daraus resultieren kann, ist, dass die Performance im Falle vieler lesender Anfragen beeinträchtigt werden könnte. Die schreibenden Operationen, die angeboten werden, sind wichtige werthaltige Operationen des Onlineshops, die darunter leiden, wenn die Performance sinkt. Die Häufigkeit der Nutzung hängt oftmals mit der aktuellen Saison und Uhrzeit zusammen. Dies bedeutet, dass es beispielsweise im Sommer zu weniger Anfragen kommt als im Winter oder auch dass mittags mehr Anfragen eintreffen als am späten Abend. Dies mündet wiederum in hohen Kosten, wenn keine Skalierbarkeit möglich ist. Die schreibenden Operationen, welche den Umsatz einbringen, sollen vor dem Verlust der Performance geschützt werden. Die lesenden Operationen sollen nach Bedarf skaliert werden, um dem entgegenzuwirken.

3.2.3 Aktuelle Messung - JMeter

Der aktuelle Zustand soll auf die Performance geprüft werden. Hierbei wird, wie zuvor erwähnt, JMeter genutzt, da die Messdaten nicht nur genauestens gemessen werden, sondern zugleich als Grafik abgebildet werden. Zur Überschaubarkeit der Ergebnisse wird nur der am häufigsten genutzte Endpunkt geprüft. Infolgedessen würden zu viele Daten der Übersichtlichkeit der Arbeit entgegenwirken. JMeter schickt Anfragen mithilfe der REST-Schnittstelle und misst verschiedene Werte. Diese sind:

- **No of Samples**
- **Latest Sample**
- **Average**
- **Deviation**
- **Throughput**

Um die Performance besser zu veranschaulichen, wird jedoch die Ausgabe der Testungen vereinfacht dargestellt. Dafür wird ein bestimmtes Plug-in genutzt, welches JMeter selbst anbietet, um die Darstellung zu vereinfachen. Die Performance gilt es, im aktuellen Architekturstil zu prüfen, um festzustellen, ob CQRS in der Umsetzung mehr oder weniger Performance erbringt.

Zunächst muss betont werden, dass der getestete Service eine der CRUD-Operationen abdecken soll. Das Akronym CRUD steht für:

- **C**reate: Daten anlegen
- **R**ead: Daten lesen
- **U**pdate: Daten aktualisieren
- **D**elete: Daten löschen

Da das Akronym nur eine allgemeine Erläuterung ist, muss entsprechend die CRUD-Operation in REST ermittelt werden. Anhand der Feststellung dieser wird sichergestellt, dass tatsächlich die richtige CRUD-Operation abgedeckt wird. REST bietet die gleichen Funktionen, welche lediglich anders benannt werden:

- POST
- GET
- PUT/PATCH
- DELETE

Die Messung, die in JMeter durchgeführt wird, muss hierbei kurz erläutert werden. In JMeter werden die URL und die Benennung des REST-Endpunktes übergeben. In dieser Arbeit wird lediglich ein Endpunkt geprüft. *»/getItems «* ist der am meist genutzte Endpunkt, den es zu testen gilt. Der Test ist wie folgt aufgebaut:

- Number of Threads (users): Anzahl der Anfragen, die verschickt werden
- Ramp-up period (seconds): Sekundenzahl in der die Anfragen verschickt werden sollen
- Loop Count: Anzahl der Iterationen des Versuchs

In diesem Fall sind es 1.500 Anfragen, die in einer Sekunde verschickt werden sollen. Dies geschieht nur einmal, da jede Iteration nur eine Wiederholung wäre. Mit dem von JMeter angebotenen *»Response Time Graph«*werden die Daten nicht nur erfasst, sondern auch unmittelbar in einen Graphen umgewandelt, aus dem die Antwortzeiten jeder Anfrage herauszulesen sind. In Abbildung 3.3 ist das Ergebnis der ersten Messung zu sehen. Zu Beginn des Tests sind die Antwortzeiten noch relativ gering, während sie im weiteren Verlauf immer weiter zunehmen. Die Höhe der Antwortzeiten steigt stark und ist mit den Antwortzeiten am Anfang nicht mehr zu vergleichen.

Die Antwortzeiten lagen anfangs bei ca. 200 Millisekunden (in folgenden ms). Mit jeder weiteren Anfrage an den Server steigt die Antwortzeit. Der Höhepunkt bei diesem Test liegt bei ca. 2.600 ms. Hierbei ist jedoch die Tendenz steigend. Dies bedeutet, dass mit jeder weiteren Anfrage die Antwortzeiten weiter zu nehmen.

3.3 Überführung in CQRS durch Verwendung von Lambdas

Um das aktuelle Projekt in CQRS zu übertragen, müssen mithin alle sondierenden und verändernden Methoden getrennt werden. Doch um dies praktisch an diesem Beispiel umzusetzen, bedarf es einiger Änderungen. Wie zuvor genannt, müssen praktisch alle Endpunkte, welche die Anwendung anbietet, korrekt aufgeteilt werden und im Sinne der Architektur sein. Jedoch wird in dieser Arbeit zunächst nur der am meist genutzte Endpunkt überprüft, um mögliche Unterschiede in der Performance zu messen. Das resultierende Ergebnis sollte zugleich auf die anderen Endpunkte übertragbar sein. Der

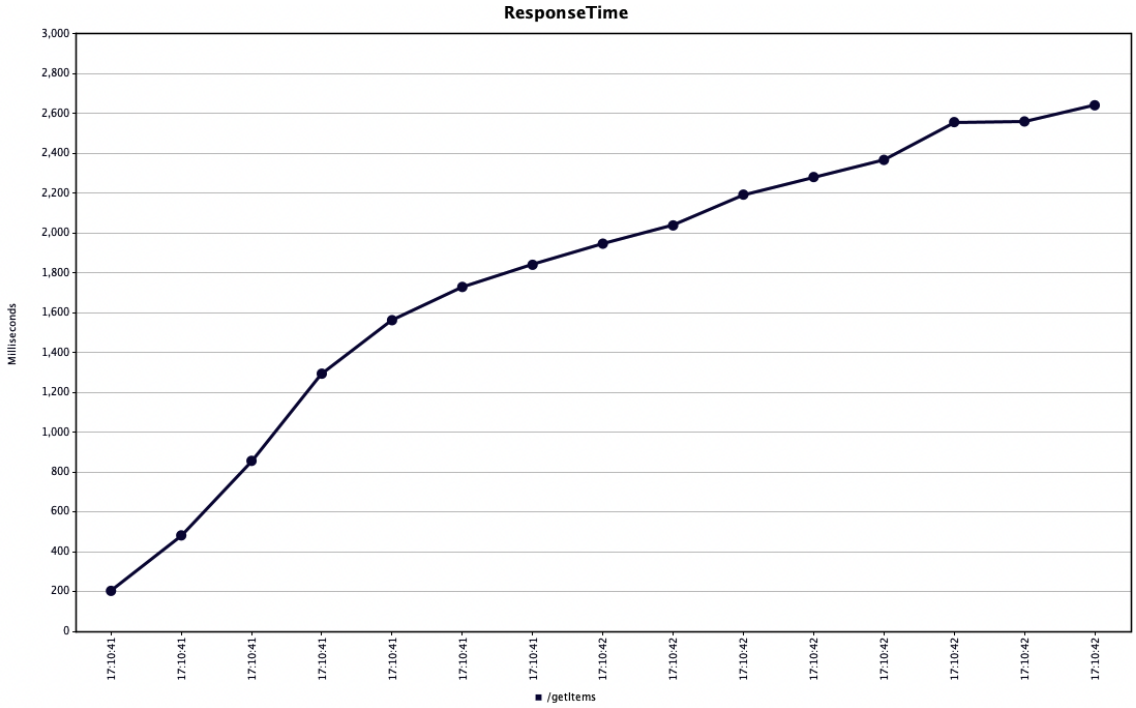

Abbildung 3.3: Messdaten des Endpunktes /getItems in Schichtenarchitektur

Endpunkt *»/getItems «* ist der am meist genutzte Endpunkt und wird derjenige sein, anhand dessen eine Veränderung an der Performance gemessen werden soll. In Abbildung 3.1 ist der aktuelle Zustand der Architektur zu sehen. In der Theorie soll die Architektur in den Zustand, wie er in Abbildung 3.4 erkennbar ist, überführt werden. Hierbei bleiben die drei Schichten Präsentations-, Applikations- und Datenschicht die gleichen.

In Abbildung 3.4 ist zu sehen, dass jede Operation mithilfe der Lambdas voneinander getrennt werden soll. Da jede Operation aufgrund der Lambdas die Möglichkeit hat, den Server, auf den die Operation sich bezieht, automatisiert zu skalieren, würde hiermit CQRS umgesetzt sein. So sollte in der Performance eine Veränderung zu sehen sein.

3.3.1 Technische Umsetzung der Architektur mit Lambdas

Damit ein Handle auf einer Lambda Funktion gehostet und ausgeführt werden kann, muss die nötige Infrastruktur geschaffen werden. Dafür sind mehrere Optionen gegeben das Vorhaben umzusetzen. Zum einen kann die jeweilige Code-Passage des Handlers händisch auf dem Lambda Service hochgeladen werden, damit anschließend die Ausführung

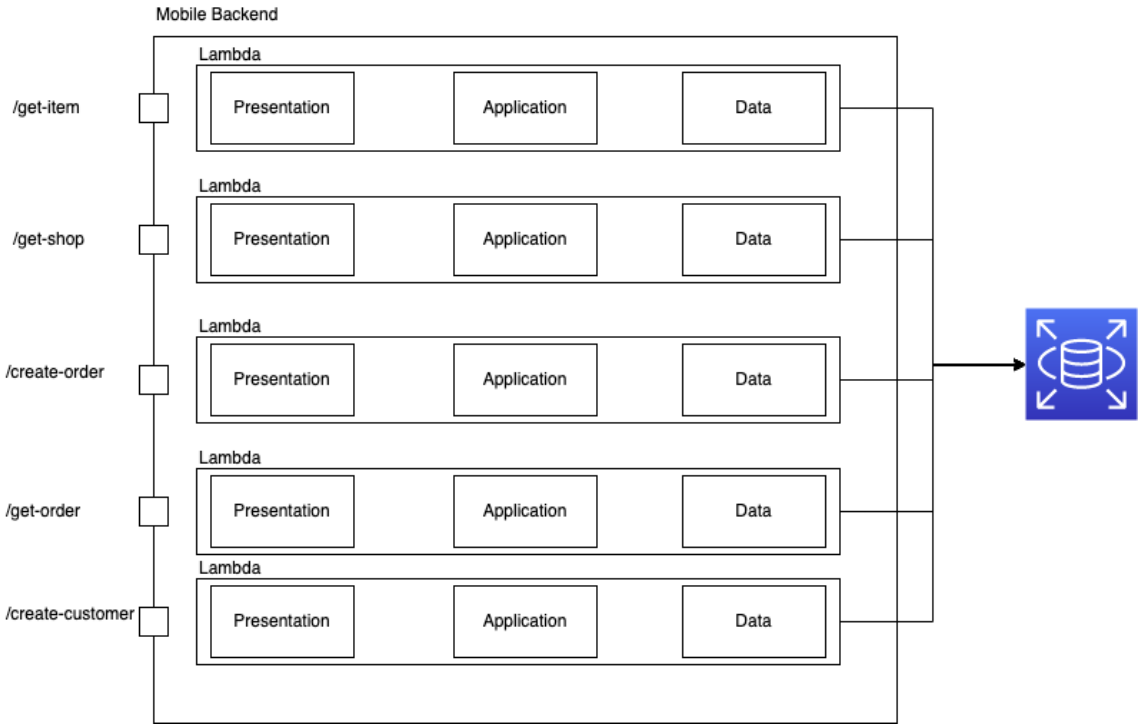

Abbildung 3.4: Abbildung des Mobile Backend nach Optimierung

ermögicht wird und somit alle Vorzüge der Lambda Technologie genutzt werden können. Das manuelle Hochladen der Handler auf die Lambda Infrastruktur ist in der Praxis ineffizient, weswegen auf Hilfs-Frameworks wie das Serverless Framework zugegriffen wird. Durch die Verwendung des Serverless Frameworks wird das automatisierte Hochladen einer neuen Version des Handlers gewährleistet.[6] Auch im Falle der beschriebenen Architektur wurde das Serverless Framework zum Deployen von neuen Handler Versionen genutzt. Hierfür muss in einem vom Serverless Framework vorgegebenen Manifest die Spezifikationen des Handlers definiert werden. Im aufgeführten Beispiel in Abbildung 3.5 ist zu ersichtlich, dass es sich hierbei um einen HTTP Handler handelt, welcher auf die HTTP Methode *GET* horcht. Unter dem Pfad */getItems* ist der Handler aufrufbar und akzeptiert zwei Query String Parameter.

Dadurch kann der Handler durch tätigen eines Befehls in der Kommandozeile auf die jeweilige Lambda Infrastruktur deployed werden. Vorteile die sich dadurch ergeben, sind dass die Handler zum Teil gemeinsamen Code teilen, welche im aufgeführten Fall die Domäne ist. Abbildung 3.6 zeigt ein Beispiel dieser Domäne, welche von verschiedenen Lambda Funktionen genutzt werden.

```
63
64    item-list:
65      handler: src/functions/item/getItemsForShopAndCategory/handler.main
66      events:
67        - http:
68            path: /getItems
69            method: get
70            cors:
71              origin: '*'
72            request:
73              parameters:
74                querystrings:
75                  shopId: true
76                  categoryId: true
77
```

Abbildung 3.5: Spezifikation an dem Handler

Ein Beispiel einer solchen Lambda Funktion ist in Abbildung3.7 zu sehen. Die Besonderheit dabei ist, dass die Handler autonom voneinander ausgeführt und skaliert werden, die Domäne jedoch die selbe bleibt.

Dadurch wird die Redundanz im Code reduziert. Dabei enthält dieses Modul die fachliche Logik der Anwendung. Handler kommunizieren über die Datenschicht mit der Datenbank. Diese lassen sich in zwei Submodule unterteilen. Zum einen sind es schreibende Operationen, welche Einträge in die Datenbank schreiben. Zum anderen sind es lesende Operationen, welche Daten aus der Datenbank geliefert bekommen. Hierfür wird ein Mapping von Daten, aus der Datenbank, in die Domäne durchgeführt. Dies ist in Abbildung 3.8 zu entnehmen.

Für die Kommunikation mit der Datenbank werden in einem dafür zuständigem Modul SQL-Queries definiert, welche gegen die Datenbank ausgeführt werden. Dies geht aus Abbildung 3.9 hervor. BLABLABLA

3.3.2 Messung mit umgesetzter Architektur

Da der aktuelle Endpunkt *»/getItems «*, wie es in einer CQRS-Architektur der Fall sein soll, ausgelagert ist, besteht die Möglichkeit, eine Messung vorzunehmen und die Ergebnisse der vorherigen Tests miteinander zu vergleichen. Bei diesem Test müssen die

```
export class Item {
  constructor(partial: Partial<Item>) {
    Object.assign(this, partial);
  }
  itemId: string;
  categoryId: number;
  salesUnit: number;
  maximumOrderQuantity: number;
  weight: number;
  brand: string;
  volumeSize: number;
  name: string;
  description: string;
  originCountry: string;
  language: number;
  imgUrl1: string;
  imgUrl2: string;
  minimumOrderQuantity: number;
  nonFood: boolean;
  freshFood: boolean;
  coolingRequired: boolean;
  isHalalCertified: boolean;
  eanNumber: string;
  isUnpublished: boolean;
  shopId: string;
}
```

Abbildung 3.6: Beispiel einer Domäne

Parameter identisch mit dem vorherigen Test sein. Durch die Schaffung gleicher Grundvoraussetzungen soll gewährleistet werden, dass Ergebnisse nicht aufgrund von unbekannten Faktoren abweichen und somit den Vergleich nicht ermöglichen. Erneut werden die gleichen Parameter genutzt:

- Number of Threads: 1500
- Ramp-up period: 1
- Loop Count: 1

Mit diesen Voreinstellungen ergibt sich Abbildung 3.10, aus der deutlich zu entnehmen ist, dass der Verlauf der Antwortzeiten sich von dem vorherigen Test deutlich unterscheidet.

```
import { formatJSONResponse } from "../../../libs/apiGateway";
import { middyfy } from "../../../libs/lambda";
import schema from "./schema";
import { getItemsForShopAndCategory } from "../Controller/getItemsForShopAndCategoryCtr";
import { APIGatewayProxyEvent, APIGatewayProxyResult } from "aws-lambda";

const getItems = async (
  event: APIGatewayProxyEvent,
): Promise<APIGatewayProxyResult> => {
  const params = event.queryStringParameters;
  console.log(params);
  const valid = await schema.bodySchema(params);
  if (valid) {
    const result = await getItemsForShopAndCategory(params);
    return formatJSONResponse({
      result,
      statusCode: 200,
    });
  } else {
    return formatJSONResponse({
      result: "please check your params",
      statusCode: 400,
    });
  }
};

export const main = middyfy(getItems);
```

Abbildung 3.7: Beispiel eines Handlers

Die x-Achse beschreibt die Zeitstempel und die y-Achse die Antwortzeit in Millisekunden (im Folgenden abgekürzt durch *ms*). Es ist ersichtlich, dass zunächst die Antwortzeiten der ersteren Anfragen deutlich zunehmen. Angefangen bei ca. 3.000 *ms* als Antwortzeit befindet sich der Höhepunkt der Antwortzeiten bei etwas über 10.000 *ms*. Relativ schnell nehmen die Antwortzeiten stark ab. Dies beruht auf der in Abschnitt 2.9.3erwähnten automatisierten Skalierung. Denn sobald der Server eine mögliche Auslastung bemerkt, skaliert der Server, um Ressourcen für die Verarbeitung von Anfragen anzubieten. Die Auswirkung dessen ist im weiteren Verlauf des Diagramms zu sehen.

Mithilfe dieser Skalierung nimmt die Antwortzeit jeder nachfolgenden Anfrage immens ab. Darüber hinaus ist zu sehen, dass sich die Antwortzeiten nicht nur verringern, sondern die ursprüngliche Antwortzeit von ca. 3.000 *ms* bei weitem unterschritten wird. Nachfolgend muss erwähnt werden, dass die Auswertung des Ergebnisses nur in Verbindung mit dem vorangehenden Test ohne Umsetzung von CQRS betrachtet werden muss. In Anbe-

tracht des ersten Tests kann an dieser Stelle von einer definitiven besseren Performance insgesamt gesprochen werden. Der vorherige Test in Abschnitt 3.2.3zeigte auf, dass die Antwortzeiten bei wenigen Anfragen eine relativ gute Performance bieten. Gleichzeitig gilt es, hervorzuheben, dass die Performance proportional zu der Anzahl der Anfragen abnimmt. Beim zweiten Test ist zu erkennen, dass der Server zu Beginn schlechtere Antwortzeiten liefert als im ersten Test. Der Grund hierfür ist, dass der Server anfangs entsprechend der Anzahl der Anfragen skaliert. Nachdem der Server jedoch mit der Skalierung der Ressourcen fertig ist, nehmen die Antwortzeiten stark ab. Diesbezüglich kann die Aussage getroffen werden, dass die Antwortzeiten nahezu proportional zur Anzahl der Anfragen abnehmen. Die Antwortzeiten nehmen sehr stark ab und unterschreiten die Antwortzeiten des ersten Tests bei weitem.

Abschließend ist mithin festzustellen, dass durch die Umsetzung von CQRS die Performance insgesamt besser ist. Als Nebeneffekt ist die Sicherheit gewährleistet, dass ein Server mit Lambda-Integration alle Anfragen verarbeiten kann und dies in kürzester Zeit.

```typescript
/* eslint-disable @typescript-eslint/no-unsafe-return */
import { RDSDataService } from 'aws-sdk';
import { Item } from '../../model/item/item';

class ItemMapper {
  public mapItem(result: RDSDataService.Types.ExecuteStatementResponse): Item {
    console.info('Map item: \n', JSON.stringify(result));
    const { records } = result;
    const values = records[0]
      .map((entry: { [key: string]: number | string | boolean }) =>
        Object.entries(entry).map(item =>
          item[0] === 'isNull' ? null : item[1],
        ),
      )
      .flat(3);

    return new Item({
      itemId: values[0] as string,
      categoryId: values[1] as number,
      isHalalCertified: values[2] as boolean,
      salesUnit: values[3] as number,
      maximumOrderQuantity: values[4] as number,
      minimumOrderQuantity: values[5] as number,
      nonFood: values[6] as boolean,
      freshFood: values[7] as boolean,
      coolingRequired: values[8] as boolean,
      brand: values[9] as string,
      weight: values[10] as number,
      volumeSize: values[11] as number,
      imgUrl1: values[12] as string,
      imgUrl2: values[13] as string,
      isUnpublished: values[14] as boolean,
      eanNumber: values[15] as string,
      language: values[16] as number,
      name: values[17] as string,
      description: values[18] as string,
      originCountry: values[19] as string,
      shopId: values[21] as string,
    });
  }
```

Abbildung 3.8: Mapping der Datenbank

```
63    public getItemsByCategorySQL = (page?: string) => `
64      SELECT item.${IColumns.id},
65        item.${IColumns.category},
66        item.${IColumns.isHalal},
67        item.${IColumns.salesUnit},
68        item.${IColumns.maximumOrderQuantity},
69        item.${IColumns.minimumOrderQuantity},
70        item.${IColumns.nonFood},
71        item.${IColumns.freshFood},
72        item.${IColumns.coolingRequired},
73        item.${IColumns.brand},
74        item.${IColumns.weight},
75        item.${IColumns.volumeSize},
76        item.${IColumns.imageUrl1},
77        item.${IColumns.imageUrl2},
78        item.${IColumns.unpublish},
79        item.${IColumns.ean},
80        item_name.${LColumns.language},
81        item_name.${LColumns.name},
82        item_name.${LColumns.description},
83        item_name.${LColumns.origin},
84        item_name.${LColumns.itemId}
85      FROM ITEM.ITEM item
86      JOIN ITEM.ITEM_NAME item_name on item.${IColumns.id} = item_name.${
87      LColumns.itemId
88    }
89      WHERE item.${IColumns.category} = :${IColumns.category}
90      ${page ? `LIMIT ${page}, 30` : ''};
91      `;
92
```

Abbildung 3.9: Datenbankqueries

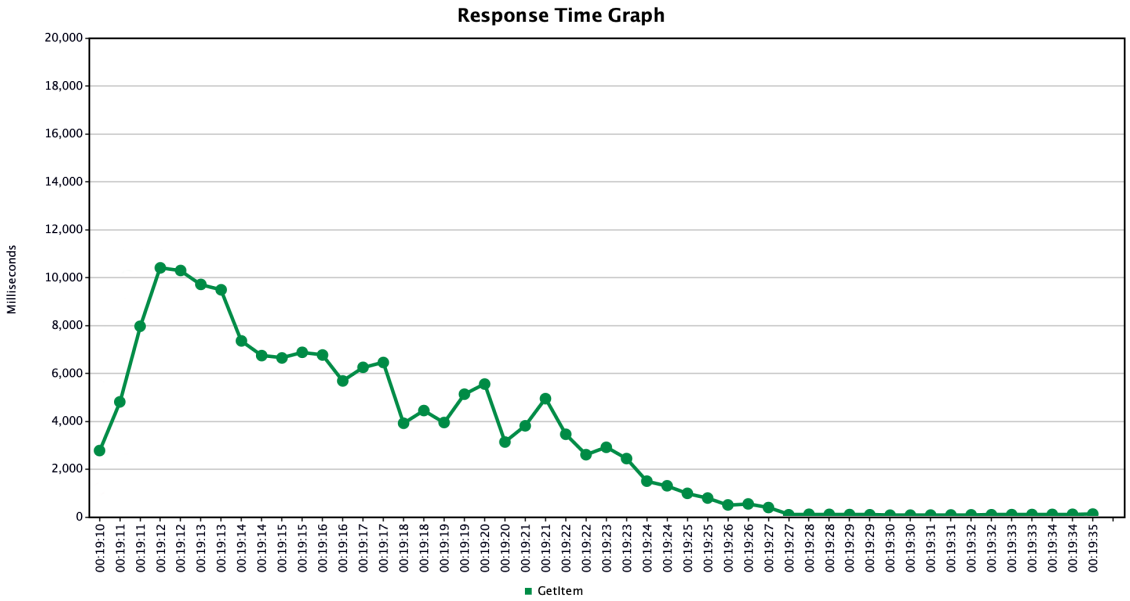

Abbildung 3.10: Testergebnis des Endpunkt: */getItems* nach CQRS

4 Schlussbemerkungen

In diesem Kapitel werden abschließend alle Ergebnisse und Erkenntnisse dieser Arbeit in komprimierter Form zusammengefasst. Zuletzt soll ein Ausblick darüber gegeben werden, wie das weitere Vorgehen für die untersuchte Anwendung sich gestalten könnte

4.1 Fazit

In der vorliegenden Arbeit wurden zunächst grundlegende Konzepte der Softwarearchitektur erläutert und einige wichtige Begriffe definiert, um ein Verständnis für die Grundlagen aufzubauen. Daraufhin wurden die Konsequenzen aufgezeigt, die entstehen, wenn eine Softwarearchitektur nicht richtig umgesetzt wird. Anschließend wurden die zu vergleichenden Architekturen eingeführt, um diese im Anschluss zu testen. Hinsichtlich der Tests ist festzuhalten, dass sowohl CQRS als die Schichtenarchitektur funktionsfähig sind. Zusammenfassend gilt, dass die Wahl der richtigen Softwarearchitektur ein entscheidender Faktor für den Erfolg von Softwareprojekten ist. Jedoch ist dies nicht das Kriterium, anhand dessen ein Vergleich gezogen werden soll. Vielmehr ist es die Performance, welche als Kriterium herangezogen wurde. Die Tests sind für eine bestimmte Anwendung durchgeführt worden, sodass diese auch fairerweise in diesem Kontext betrachtet werden müssen. Die Tests haben gezeigt, dass die Umsetzung von CQRS insgesamt mehr Performance liefert als die Schichtenarchitektur. Die Schichtenarchitektur liefert je nach Anzahl der Anfragen schlechtere Antwortzeiten. Es ist festzustellen, dass mit zunehmender Anzahl der Anfragen die Performance abnimmt bzw. die Antwortzeiten steigen. Hingegen hat die Umsetzung von CQRS insgesamt eine Verbesserung der Performance herbeigeführt. Wenngleich die Antwortzeiten der Anfragen anfangs weitaus höher sind, nehmen diese jedoch nach Skalierung mithilfe der Lambda-Technologie stark ab und unterschreiten die zu Beginn schlechten Antwortzeiten erheblich. Sogar die Antwortzeiten von CQRS unterschreiten jene der Schichtenarchitektur bei weitem. Es muss jedoch betont werden, dass nicht alle Operationen separiert sind, sondern nur die meist genutzten.

Demnach gilt, dass eine komplette Umsetzung der CQRS-Architektur einen noch positiveren Effekt haben kann und die Ergebnisse sich erheblich verbessern könnten. Zudem hat sich gezeigt, dass die Wahl der richtigen Tools und Technologien ebenfalls von entscheidender Bedeutung ist. Hierbei sollten nicht nur die Funktionen der Tools, sondern gleichsam deren Integration in den Entwicklungsprozess und die Schulung der Entwickler berücksichtigt werden.

Insgesamt bietet das Thema Softwarearchitektur zahlreiche spannende und wichtige Aspekte, die nicht nur in der Forschung, sondern auch in der Praxis von großer Bedeutung sind. Die Erkenntnisse dieser Arbeit können dazu beitragen, bessere Entscheidungen bei der Wahl der richtigen Architektur und Tools zu treffen und somit erfolgreiche Softwareprojekte zu realisieren.

4.2 Ausblick

Die vorliegende Arbeit hat zur Erkenntnis geführt, dass es bei der Wahl der richtigen Softwarearchitektur entscheidend ist, den individuellen Anforderungen und Bedürfnissen des Projekts Rechnung zu tragen. Eine zukünftige Forschung in diesem Bereich könnte sich beispielsweise mit der Entwicklung von allgemein gültigen Entscheidungskriterien befassen, die bei der Wahl der optimalen Architektur helfen können. Hierbei könnten auch Aspekte wie Performance, Skalierbarkeit und Flexibilität berücksichtigt werden. Ein weiterer spannender Aspekt wäre die Weiterentwicklung der verglichenen Architekturen. Hierbei könnte untersucht werden, wie sich die Architekturen in der Praxis bewähren und welche Erfahrungen dabei gesammelt werden. Ferner könnte eruiert werden, wie sich die Architekturen in Zukunft weiterentwickeln lassen und welche neuen Konzepte und Technologien dabei zum Einsatz kommen können. Ein weiterer interessanter Bereich wäre die Untersuchung der Auswirkungen von Softwarearchitektur auf die Zusammenarbeit im Entwicklerteam. Hierbei könnten beispielsweise verschiedene Architekturen hinsichtlich ihrer Auswirkungen auf die Kommunikation, Koordination und Zusammenarbeit im Team untersucht werden. Ebenso könnte erforscht werden, welche Faktoren dabei eine Rolle spielen und wie die Architektur die Zusammenarbeit im Team beeinflussen kann.

Insgesamt bietet das Thema „Vergleich von Softwarearchitekturen“ eine Reihe spannender Forschungsfragen und Entwicklungsmöglichkeiten, die sowohl in der Forschung als auch in der Praxis von wesentlicher Bedeutung sind. Die Erkenntnisse dieser Arbeit können

dazu beitragen, die zukünftige Forschung und Entwicklung in diesem Bereich zu fördern und die Leistungsfähigkeit sowie Qualität von Softwarelösungen zu verbessern.

Literaturverzeichnis

[1] AWS, Amazon: Was ist AWS Lambda. . – URL https://docs.aws.amazon.com/de_de/lambda/latest/dg/welcome.html

[2] Comartin, Derek: Is CQRS Complicated? (27 Februar 2019). – URL https://codeopinion.com/is-cqrs-complicated/

[3] Evans, Eric: *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison Wesley, 2003. – URL https://www.google.de/books/edition/Domain_driven_Design/xColAAPGubgC?hl=de&gbpv=1&dq=Domain-Driven+Design:+Tackling+Complexity+in+the+Heart+of+Software.&printsec=frontcover. – ISBN 9780321125217

[4] Flügge, Andreas: ARCHITEKTURSICHTEN (TEIL IV). (2012). – URL https://www.objectsystems.de/artikel-softwarearchitektur-teil-iv-architektursichten.html

[5] Fowler, Martin: CQRS. (14 Juli 2011). – URL https://martinfowler.com/bliki/CQRS.html

[6] Inc., Serverless: Serverless Framework Concepts. (2022). – URL https://www.serverless.com/framework/docs/providers/aws/guide/intro

[7] Lilienthal, Carola: *Langlebige Software-Architekturen.* dpunkt.verlag, 2016. – URL https://books.google.com/books?id=mmo0jgEACAAJ&dq=carola+lilienthal&hl=de&newbks=1&newbks_redir=1&sa=X&ved=2ahUKEwiXlvqL9aH8AhW4QvEDHVBSALUQ6AF6BAgCEAE. – ISBN 9783864907296

[8] Rechenberg, Salim: *Diplomarbeit Wechselwirkungen zwischen Performance und Architektur – Analyse, Optimierung und Evaluation am Beispiel des JCommSy*, Universität Hamburg, Dissertation, 2009

[9] Schön, Hendrik: *DIPLOMARBEIT - EINFLUSS VON SOFTWAREARCHITEKTUR AUF DEN WERT EINES SOFTWARESYSTEMS*, Technische Universität Dresden, Dissertation, 2016

[10] Starke, Gernot: *Software Architektur kompakt - angemessen und zielorientiert*. Spektrum Akademischer Verlag, 2011. – URL https://www.google.de/books/edition/Software_Architektur_kompakt/Dd8rMlyPfVQC?hl=de&gbpv=0. – ISBN 9783827428349

[11] Starke, Gernot: What's in a Name: Architecture. (2016). – URL https://www.innoq.com/en/blog/whats-in-a-name-architecture/

[12] Vernon, Vaughn: *Implementing Domain-driven Design*. Vaughn Vernon, 2013. – URL https://www.google.de/books/edition/Implementing_Domain_driven_Design/aVJsAQAAQBAJ?hl=de. – ISBN 9780321834577

[13] Wikipedia: Apache JMeter. (10 Februar 2023). – URL https://de.wikipedia.org/wiki/Apache_JMeter

[14] Zdun, Oliver Vogel Ingo Arnold Arif Chughtai Edmund Ihler Timo Kehrer Uwe Mehlig U.: *Software-Architektur Grundlagen - Konzepte - Praxis*. Spektrum Akademischer Verlag Heidelberg, 2009. – URL https://link.springer.com/book/10.1007/978-3-8274-2267-5. – ISBN 9783827422675

A Anhang

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original