# Sperrvermerk

Auf Wunsch der Firma Fusonic GmbH, im Auftrag der Viterma Handels GmbH, ist die vorliegende Arbeit bis zum 30. November 2023 für die öffentliche Nutzung zu sperren.

Veröffentlichung, Vervielfältigung und Einsichtnahme sind ohne ausdrückliche Genehmigung der oben genannten Firma und dem Verfasser nicht gestattet. Der Titel der Arbeit sowie das Kurzreferat/Abstract dürfen jedoch veröffentlicht werden.

Dornbirn, am 30. November 2020 Unterschrift des Verfassers

Firmenstempel

FH Vorarlberg
University of Applied Sciences

# [Titel der Arbeit]

## [Untertitel der Arbeit]

Bachelorarbeit I
zur Erlangung des akademischen Grades

**Bachelor of Science in Engineering (BSc)**

Fachhochschule Vorarlberg
Informatik – Software and Information Engineering

Betreut von
Prof. (FH) Dipl. Inform. Thomas Feilhauer

Vorgelegt von
Dominic Luidold
Dornbirn, 30. November 2020

# Widmung

Trotz des Sperrvermerks, an der diese Arbeit gebunden ist, möchte ich es mir nicht nehmen lassen, den Personen einen Dank auszusprechen, die mich bei der Umsetzung und Realisierung meiner Bachelorarbeit I unterstützt haben.

Zu Beginn möchte ich meinen Betreuern Thomas Feilhauer (Fachhochschule Vorarlberg) und Michael Zangerle (Fusonic GmbH) danken, die jederzeit ein offenes Ohr für Fragen, Anliegen und Unklarheiten hatten. Durch ihre fachliche Kompetenz, ihre langjährige Erfahrung sowie ihr Know-how konnte ich vieles lernen, das ich in mein weiteres Berufsleben mitnehmen kann.

Als nächstes möchte ich meiner gesamten Familie danken, denen ich während des Schreibens dieser Arbeit wahrscheinlich mehr als nur einmal auf die Nerven gegangen bin. Nur durch die Tatkräftige Unterstützung mit Snacks, Süßigkeiten und dem besten Zimmerservice der Welt konnte ich so ungestört an der Bachelorarbeit schreiben. Ohne ihr Korrekturlesen wären zudem mehr Tippfehler vorhanden als mir lieb sind und ich auf meine Tastatur schieben kann.

Zu guter Letzt möchte ich meine Arbeit all jenen widmen, die für die Gleichbehandlung und Gleichberechtigung aller kämpfen, sich für eine Sache einsetzen, die größer als sie selbst ist und für die sie teilweise sogar ihr Leben aufs Spiel setzen. Auch 2020 braucht es weltweit immer noch laute Stimmen - Stimmen, die sich nicht unterkriegen lassen.

*„The time is always right to do what is right.“*
Dr. Martin Luther King Jr.

# Kurzreferat

## [Deutscher Titel Ihrer Arbeit]

[Text des Kurzreferats]

# Abstract

## [English Title of your thesis]

[text of the abstract]

## Geschlechtergerechte Sprache

Der Verfasser der vorliegenden Arbeit bekennt sich zu einer geschlechtergerechten Sprachverwendung.

Um die Lesbarkeit zu gewährleisten und zugunsten der Textökonomie werden die verwendeten Personen beziehungsweise Personengruppen fix männlich oder weiblich zugeordnet. Zum Beispiel wird immer „die Entwicklerin“ und „der Benutzer“ verwendet. Es wurde besonders darauf geachtet, stereotype Rollenbeschreibungen zu vermeiden. Die insgesamt eventuell dadurch hervorgerufene Irritation bei den Lesenden ist gewünscht und soll dazu beitragen, eine Bewusstheit für die bestehende, Frauen diskriminierende Sprachgewohnheit (generelle Verwendung der männlichen Begriffe für beide Geschlechter) zu wecken beziehungsweise zu stärken.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Abkürzungsverzeichnis

**API** Application Programming Interface

**BLS** Better Life System

**CI** Continuous Integration

**CD** Continuous Deployment

**CQS** Command Query Separation

**CQRS** Command Query Responsibility Segregation

**CRUD** Create, Read, Update, Delete

**DTO** Data Transfer Object

**IDE** Integrierte Entwicklungsumgebung

**JSON** JavaScript Object Notation

**LTS** Long Term Support

**MVC** Model View Controller

# 1 Einleitung

Diese Bachelorarbeit verfolgt das Ziel, einen Einblick in die Implementierung und Erweiterung des bereits bestehenden Backend-Systems des „Better Life System“ (kurz „BLS“) der Viterma Handels GmbH mit dem sogenannten „CQRS“-Pattern zu geben.

Das Better Life System - welches von Fusonic GmbH entwickelt wird und mittels einer Weboberfläche und gängigen Webbrowsern bedient werden kann - ermöglicht es Endkunden, mithilfe einer Handelsvertreterin der Firma Viterma (beziehungsweise mit einer Vertreterin eines ihrer Franchise-Partner) ein Badezimmer anhand der jeweiligen Bedürfnisse auszusuchen und zu konfigurieren, um sich schlussendlich ein entsprechendes Angebot dafür erstellen zu lassen.

## 1.1 Arbeitgeber Fusonic GmbH

Die Fusonic GmbH hat ihren Firmensitz in Götzis, Vorarlberg, und besteht aus einem Team von aktuell mehr als 25 Angestellten, Softwareentwicklerinnen und Projektleiterinnen. Diese sind aufgeteilt in diverse kleinere Teams, die intern unter anderem „Duck-Team“ beziehungsweise „Parrot-Team“ genannt werden. Die Teams arbeiten dabei an jeweils eigenständigen Projekten und setzen diverse Technologien ein. Zu den verwendeten Technologien gehören unter anderem C# mit .NET, PHP mit Symfony und JavaScript/TypeScript mit Angular (Fusonic GmbH 2020, "Übersicht aller Technologien"). Während es regelmäßigen Austausch unter den Teams gibt, besteht jedes aus Frontend- sowie Backend-Entwicklern, da die meisten Projekte - aufgrund der zugrundeliegenden Anwendungsfälle - aus einem Web-Frontend sowie serverseitigen Backend bestehen.

## 1.2 Nutzung & Umfeld des „Better Life System“

Die Hauptaufgabe des Better Life System liegt darin, als computergestütztes und plattformunabhängiges Tool, Vertreterinnen rund um die Viterma Handels GmbH bei der Konfiguration und Zusammenstellung eines Badezimmers - zusammen mit dem Endkunden (meist Haus- beziehungsweise Wohnungsbesitzer oder Hotels) - zu unterstützen und diesen Vorgang zu erleichtern. Das BLS wird des Weiteren dazu verwendet, aktuelle und abgeschlossene Angebote zu verwalten, Produkte, Produktinformationen und dazugehörige Preisaufschläge zu pflegen und Kundendaten abzulegen. Zudem dient es als Kontrollinstanz für in Auftrag gegebene Sanierungen, um sicherzustellen, dass alle benötigten Materialen und Teile in entsprechenden Mengen vorhanden und kompatibel sind.

Die Nutzung des BLS findet zu großen Teilen auf mobilen Rechnern beziehungsweise Laptops statt, die während der Beratung und Betreuung von Kunden zum Einsatz kommen. Da nicht immer gewährleistet werden kann, dass eine aufrechte Internetverbindung besteht, ist die Offline-Fähigkeit bei gleichbleibender Nutzung im Webbrowser ein wichtiger Bestandteil des Web-Frontends.

Aufgrund der Anforderungen der Viterma Handels GmbH, dass das Better Life System offline ebenso wie online verwendet werden kann, basiert das Frontend auf dem TypeScript Web-Framework Angular[1], welches solch eine Funktionalität ohne eine Installation oder zusätzliche Voraussetzungen am Endgerät unterstützt. Im Gegensatz zu weniger komplex gehaltenen Frontends - die primär Daten darstellen, die vom Backend eingehen - ist dieses beim BLS für einen Großteil der Ablauflogik, Berechnung von Preisen und Generierung von diversen PDFs zuständig. Das Frontend wird, da der Fokus dieser Arbeit auf der Entwicklung im Backend der Anwendung liegt, in weiterer Folge jedoch nicht genauer beleuchtet und als gegeben angenommen. Es kann davon ausgegangen werden, dass eingehende Anfragen an das Backend aus Benutzereingaben beziehungsweise zeitlich gesteuerten Events resultieren.

[1] Angular (https://angular.io)

Das Backend, welches alle API-Anfragen bearbeitet und als Schnittstelle zur Datenbank dient, ist in der Programmiersprache PHP geschrieben und verwendet das Framework Symfony[2] sowie diverse „Bundles“, die darauf aufbauen. Ein detaillierterer Überblick über den Aufbau, bisher angewandte Konzepte sowie technische Begebenheiten werden in Kapitel 2 ab Seite 18 behandelt.

Während die Nutzung des Better Life System auf allen Plattformen - die einen modernen Webbrowser anbieten - möglich ist, findet die Entwicklung primär auf der Linux-Distribution Ubuntu statt. Der Grund für diese Betriebssystemwahl ist damit zu erklären, dass sowohl die lokale Entwicklung als auch die Continuous Integration Umgebung (genutzt für automatisierte Tests) und das Continuous Deployment mittels Docker[3] Containern stattfindet. Für die Entwicklung im Backend-Bereich der Anwendung wird die IDE PhpStorm verwendet, während im Frontend auf Visual Studio Code gesetzt wird.

## 1.3 Problemstellung

Die Entwicklung des BLS hat im Frühjahr 2017 begonnen und die Anwendung ist seit dem stetig weiterentwickelt worden. Über die Zeit haben sich entsprechend sowohl die Anforderungen und Wünsche des Kunden als auch die technischen Begebenheiten, Best Practices und Möglichkeiten verändert.

Die bisher bei der Umsetzung des Projekts verwendete Struktur sowie der Aufbau wurde stark an die empfohlene Herangehensweise von Symfony angelehnt und die Code-Basis entsprechend darauf ausgelegt (siehe dazu Symfony 2020, „Use the Default Directory Structure“). Durch wachsende Anforderungen und entsprechend benötigten Programmcode, der diese erfüllt, hat sich jedoch ein komplexes System ergeben, dessen *Controller*, *Entities*, *Manager (Services)* und *Data Transfer Objects (DTOs)* auf viele Verzeichnisse und Unterverzeichnisse verteilt sind. Die Abbildung 1.1 auf Seite 16 zeigt einen Ausschnitt von Ordnern die im Projekt vorkommen und jeweils alle Klassen ihrer Art beinhalten (im Verzeichnis „Controller“ befinden sich, um ein Beispiel zu nennen, alle Controller des Projekts).

[2]Symfony (https://symfony.com)
[3]Docker (https://docker.com)

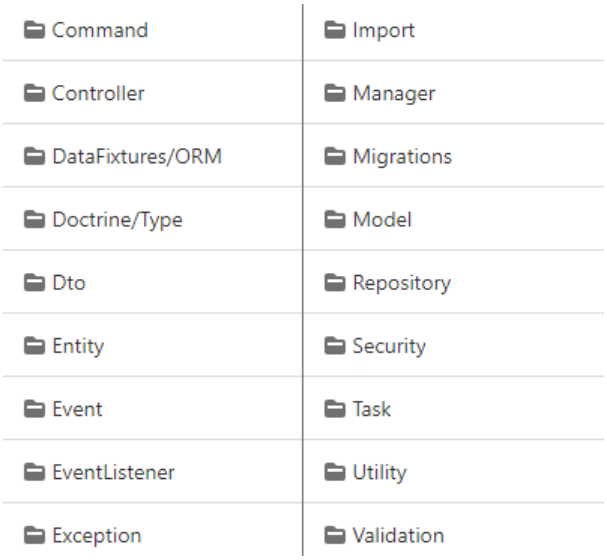


Abbildung 1.1: Teilausschnitt der BLS-Ordnerstruktur

Durch die bisher gewählte Herangehensweise und den entsprechenden Aufbau hat sich jedoch ergeben, dass eine logische Gruppierung beziehungsweise Kapselung von zusammengehörenden Klassen, Controllern, Managern etc. eines API Endpunkts nur schwer möglich ist. Das hat zur Folge, dass das System an manchen Stellen sehr komplex aufgebaut ist und externe Personen eine gewisse Zeit brauchen, um sich mit internen Abläufen vertraut machen zu können. Neben der Verständlichkeit ist dadurch in weiterer Folge auch die Wartbarkeit und einfache Erweiterbarkeit im Falle von neuen Funktionalitäten nicht im vollen Ausmaß gegeben. Auf das Testen einzelner Bestandteile mittels sogenannter *Unit Tests* wird im gesamten Projekt gesetzt. Primär werden beim Better Life System *Smoke Tests* sowie *Funktionale Tests* verwendet, die im Problemfall darauf hindeuten können, wo ein Fehler aufgetreten ist.

Da sich die Anforderungen an das Backend auch in Zukunft noch weiterentwickeln können und eine Skalierung der Infrastruktur beziehungsweise Abspaltung einzelner Teile (in sogenannte Microservices) nicht ausgeschlossen werden kann, bedarf es Möglichkeiten, dies möglichst effektiv und ohne großen, zusätzlichen Aufwand umsetzen zu können. Die von Symfony empfohlene Herangehensweise kann dies dabei nur bedingt unterstützen, was auch in diesem Fall einen limitierenden Faktor darstellt.

## 1.4 Anforderungen

Die in Abschnitt 1.3 angeführten Umstände und den daraus resultierenden Herangehensweisen, die nicht mehr in vollem Maße zufriedenstellend sind, führen zu neuen Anforderungen an das Better Life System. Diese sollen entsprechend umgesetzt werden, um ein zukunftssicheres Backend für die gesamte Anwendung garantieren zu können und die Entwicklung neuer sowie bestehender Funktionalitäten zu erleichtern und zu vereinheitlichen.

Zum Start des Berufspraktikums Anfang Juli 2020 stand bereits fest, dass in weiterer Folge das CQRS-Pattern (was für *Command Query Responsibility Segregation* steht) zum Einsatz kommen wird. Andere Projekte der Fusonic GmbH, die primär auf C# und .NET basieren, haben bereits gute Erfahrungen damit gemacht und gehen deshalb davon aus, dass der Einsatz des Patterns die Qualität der bestehenden Code-Basis verbessern wird.

Die Anforderungen, die im Zuge des Berufspraktikums und folglich dieser Bachelorarbeit zu erfüllen sind, bestehen darin, das aktuell bestehende System, neben dem „Tagesgeschäft“ (sprich Bugfixes, Feature Requests etc.), laufend umzustellen und neue Funktionalitäten entsprechend mit der neuen Struktur und dem Pattern umzusetzen.

## 1.5 Zielsetzung

Die in den Abschnitten 1.3 und 1.4 angeführten Punkte ergeben folgende Ziele, die im Verlauf des Berufspraktikums - so weit wie möglich - umzusetzen sind. Als Ziele einzustufen sind:

- Vorbereitung der bestehenden Infrastruktur auf CQRS in Zusammenarbeit mit Mitarbeitern der Fusonic GmbH
- Umstellung der bestehenden API Endpunkte auf CQRS bei Beibehaltung von möglichst viel Programmlogik
- Umsetzung neuer Funktionalitäten mittels CQRS
- Abwickeln des Tagesgeschäfts

# 2 Stand der Technik

Das folgende Kapitel gibt einen Überblick über den aktuellen Stand des Backends des Better Life System, dessen technischem Aufbau und dem zugrundeliegenden Konzept. Anschließend wird die von Symfony vorgeschlagene Architektur beleuchtet und auf das zukünftig eingesetzte CQRS-Pattern eingegangen. Am Ende des Kapitels werden die gängige Schichtenarchitektur und CQRS gegenübergestellt und Vor- sowie Nachteile aufgezeigt.

## 2.1 Aufbau des Backends

Das Backend des BLS basiert auf der Programmiersprache PHP und setzt hierbei auf die zum aktuellen Zeitpunkt (Stand August 2020) neueste Version *PHP 7.4*. Zur Verwaltung benötigter Abhängigkeiten kommt das Paketverwaltungssystem Composer[1] zum Einsatz, welches alle benötigten Symfony Bundles und anderweitige Packages - sowohl im Produktivsystem als auch während der Entwicklung - zur lokalen Installation sowie Nutzung zur Verfügung stellt.

Als Grundgerüst für das Backend dient das PHP-Framework Symfony in der LTS-Version *4.4*, wobei zum Zeitpunkt des Verfassens dieser Arbeit Vorbereitungen für einen Umstieg auf Version *5.x* getroffen werden. Das Backend dient als Schnittstelle zwischen dem Angular-Frontend und einem Datenspeicher, wobei hierfür eine relationale Datenbank auf Basis von MySQL eingesetzt wird. Um von Symfony aus auf die Datenbank zugreifen zu können, wird Doctrine[2] eingesetzt, welches ein objektrelationales Mapping vornimmt und in der Anwendung zur Verfügung stellt.

[1] Composer (https://getcomposer.org)
[2] Doctrine (https://www.doctrine-project.org)

Der grundsätzliche Aufbau des Symfony Backends entspricht aktuell der empfohlenen Vorgehensweise laut Symfony Dokumentation und wird im nachfolgenden Abschnitt 2.2 erläutert.

## 2.2 Gängige Schichtenarchitektur

Symfony ist ein Webframework, welches aus vielen verschiedenen Komponenten und Bundles besteht, die alle unabhängig voneinander entwickelt werden. In Summe ergeben diese die Grundlage für eine Webapplikation auf Basis von PHP und werden auch von anderen Frameworks verwendet (beispielsweise Laravel[3], welches auf diversen Symfony-Komponenten basiert). Symfony orientiert sich dabei an der weit verbreiteten *Model View Controller* Architektur (kurz *MVC*) und bietet mittels flexiblem URI Routing, Session beziehungsweise Security Management und Logging unterstützende Funktionalitäten. (Tutorials Point (I) Pvt. Ltd. 2020b)

### 2.2.1 Empfohlener Ansatz

Symfony selbst gibt in der eigenen Dokumentation keine verpflichtenden Vorgaben an, wie genau man eine Applikation beziehungsweise Anwendung umsetzen soll. (Symfony 2020) Der Quellcode 1 auf Seite 20 zeigt den empfohlenen Aufbau einer Anwendung, bei der man die Anlehnung an das MVC-Pattern jedoch gut erkennen kann:

- **Model:** Dem Model entsprechen die im *Entity*-Verzeichnis abgelegten Klassen. Diese beinhalten das objektrelationale Mapping und können zusätzliche, domänenspezifische Funktionalitäten zur Verfügung stellen. Die Klassen, die im *Repository*-Verzeichnis zu finden sind, stellen den Zugriff auf die Datenbank her und befüllen die Entities beziehungsweise Models mit Daten. (Tutorials Point (I) Pvt. Ltd. 2020a)

- **View:** Im Falle der hier angenommenen Standardstruktur, wird der View-Teil mittels der Template Engine *Twig* im gleichnamigen Verzeichnis ab-

[3]Laravel (https://laravel.com)

gehandelt. Im Falle des BLS ist dies mittels Data Transfer Objects gelöst, die bei den jeweiligen API Endpunkten als serialisierte JSON-Response zurückgegeben werden. (Tutorials Point (I) Pvt. Ltd. 2020a)

- **Controller:** Die von der Namensgebung her am leichtesten zu erkennende Ähnlichkeit stellt das *Controller*-Verzeichnis dar. Die darin vorhandenen Klassen werden verwendet, um eingehende Requests zu mappen. Diese Controller-Klassen sollten jedoch nur wenige Zeilen Code beinhalten, um Berechtigungen zu überprüfen sowie die Response zu erstellen. Die tatsächliche Businesslogik wird in separate Klassen aufgeteilt, die vom Controller aufgerufen werden, um diese möglichst zu entkoppeln. (Symfony 2020) Beim BLS liegt die Anwendungslogik beispielsweise in einem *Manager*-Verzeichnis.

```
1  project/
2     bin/
3        console
4     config/
5        packages/
6        services.yaml
7     public/
8        build/
9        index.php
10    src/
11       Command/
12       Controller/
13       DataFixtures/
14       Entity/
15       EventSubscriber/
16       Form/
17       Migrations/
18       Repository/
19       Security/
20       Twig/
21    tests/
22    var/
23       cache/
24       log/
25    vendor/
```

Quellcode 1: Von Symfony empfohlener Projektaufbau. Quelle: Symfony 2020

### 2.2.2 Funktionsweise

Die Funktionsweise der von Symfony vorgeschlagenen Architektur kann, unter Bezugnahme auf die in Abschnitt 2.2.1 beschriebenen Prinzipien, mithilfe der Abbildung 2.1 genauer beleuchtet werden.

Im Zuge der Entwicklung einer Webapplikation mit entsprechendem Backend oder einer API wird oftmals auf CRUD beziehungsweise einen CRUD-ähnlichen Aufbau gesetzt. Die Nutzung eines einzelnen Models, dem jeweiligen DTO (sofern vorhanden) sowie der dazugehörenden Businesslogik ist dabei ein einfacher Weg, um schnell und anfangs unkompliziert Funktionalitäten für die CRUD Anwendungsfälle umsetzen zu können. Auch bei weiterführenden Nutzungsfällen, bei denen beispielsweise mehrere Models zusammengeführt und entsprechend validiert werden müssen, wird die bestehende Logik herangezogen und keine spezifische Auftrennung von ausgeführten Aktionen vorgenommen. Dadurch entsteht jedoch eine Vermischung von lesenden und schreibenden Zugriffen in der Anwendungslogik sowie der Funktionalitäten der entsprechenden Models. (Fowler 2011) Die Abbildung 2.1 stellt diesen Umstand anschaulich im hellgrau hinterlegten Rechteck „Application“ dar:

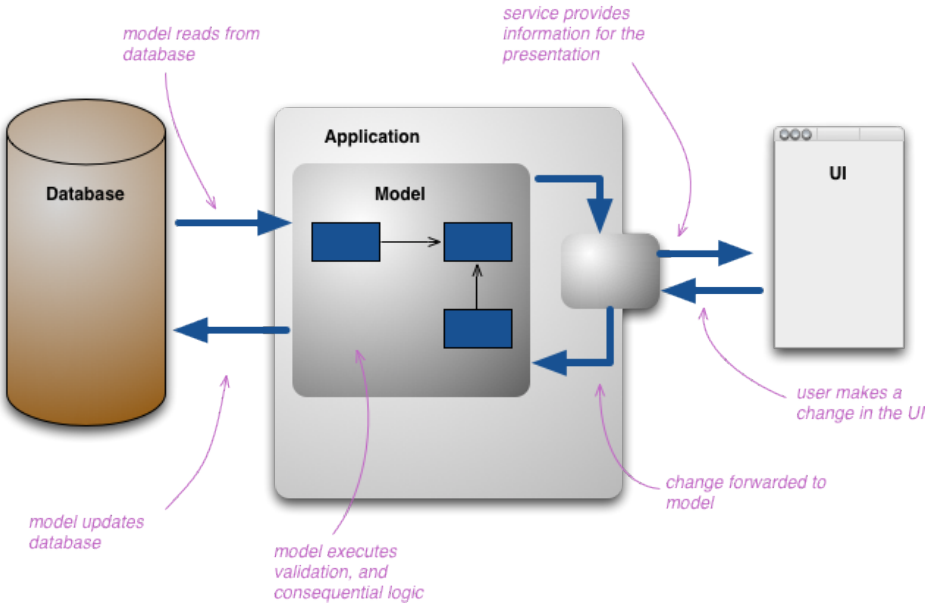


Abbildung 2.1: Aufbau einer gängigen Schichtenarchitektur.
Quelle: Fowler 2011

## 2.3 CQRS Pattern

Das Command Query Responsibility Segregation Pattern (nachfolgend CQRS genannt) soll als Grundlage für die zukünftige Struktur des Better Life System dienen. Die Entscheidung für den Wechsel auf dieses Pattern beziehungsweise die zugrundeliegende Architektur basiert primär auf Erfahrungswerten, die Fusonic GmbH bei anderen Projekten sammeln konnte. Der Einsatz dieses Patterns soll die im Kapitel 1 angesprochenen Problemstellungen in Angriff nehmen und die Komplexität, Verständlichkeit, Testbarkeit und Erweiterbarkeit des Systems langfristig verbessern.

Die bisherigen Erfahrungen, die Fusonic mit dieser Herangehensweise gemacht hat, basieren auf C# und .NET Core Projekten. Ein entsprechender Einsatz mit der Programmiersprache PHP und dem Framework Symfony wurde bisher nur konzeptionell durchgeplant und findet damit erstmalig Einzug in ein Kundenprojekt.

### 2.3.1 Grundlagen von CQRS

Um über das CQRS-Pattern sprechen zu können, werden einige grundlegende Begrifflichkeiten sowie Konzepte benötigt, die im weiteren Verlauf dessen Ursprung definieren und Verständnis schaffen sollen.

CQRS, beziehungsweise die Terme *Command* und *Query*, beziehen sich auf das Prinzip der *Command Query Separation* (kurz *CQS*). (Young 2010) CQS definiert auf der einen Seite lesende und auf der anderen Seite schreibende Aktionen in einem System, die voneinander getrennt sind und unterschiedlich agieren sowie verschiedene Resultate als Folge haben:

- Lesende Aktionen, sogenannte **Queries**, greifen auf Daten in einem System zu und liefern diese bei einem Aufruf als Ergebnis zurück. Durch das Ausführen der Aktion wird der Zustand eines Objekts oder des Systems im Ganzen nicht verändert. (Fowler 2005)

- Schreibende Aktionen, sogenannte **Commands**, verändern Daten in einem System, führen jedoch zu keinem Rückgabewert. Durch das Aus-

führen der Aktion wird der Zustand eines Objekts oder des Systems im Ganzen bleibend verändert. (Fowler 2005)

Fowler beschreibt den Vorteil dieser Aufteilung damit, dass Methoden, die den Zustand eines Objekts oder Systems verändern, von denen logisch getrennt werden können, die nur lesend darauf zugreifen. Queries können dadurch „gefahrlos“ und in beliebiger Reihenfolge an jeder Stelle in einer Anwendung eingesetzt werden, während dies bei Commands nicht gegeben ist. (Fowler 2005)

### 2.3.2 Funktionsweise von CQRS

Die Funktionsweise des CQRS Konzepts basiert, wie im Abschnitt 2.3.1 beschrieben, auf dem Einsatz von Commands und Queries. Durch deren Einsatz wird eine effektive Aufteilung in zwei separate und optimalerweise unabhängige Modelle erzielt, wodurch zum einen ein Fokus auf die jeweilige Verantwortlichkeit des Modells gelegt werden kann und zum anderen die übermäßige Optimierung hin zu einer Datenbank wegfällt. Dadurch rückt die Anwendungsbeziehungsweise Businesslogik deutlich mehr in den Vordergrund und kann entsprechend besser ausgebaut und getestet werden. (Heimeshoff und Jander 2013)

Anhand der Abbildung 2.2 auf Seite 24 wird verdeutlicht, wie der Ablauf in einem auf CQRS basierenden System abläuft:

**Ändern von Daten:** Eine vom Benutzer ausgelöste Änderung der Daten (im Falle des BLS durch eine Aktion im Angular-Frontend) führt dazu, dass diese an ein Command Model weitergeleitet werden. Das Model beinhaltet die Daten, die abgeändert und persistiert werden sollen und stellt diese gegebenenfalls einem Domänenmodell mit entsprechenden Funktionalitäten - auf Basis des Domain-Driven Designs - zur Verfügung (Heimeshoff und Jander 2013). Das Command Model, dass die Integrität der eingehenden Daten sichergestellt hat, wird abschließend in der Datenbank persistiert, womit die Aktion somit - im Regelfall ohne Rückgabewert - abgeschlossen wird. (Fowler 2011)

**Abfragen von Daten:** Ein Abfragen der Daten (ebenfalls durch das Frontend ausgelöst) führt dazu, dass diese von der Datenbank geladen und mittels des Query Models (in manchen Fällen auch mehrerer) im entsprechenden Format serialisiert werden. Die Aktion wird damit abgeschlossen, dass die Daten

nach außen als Rückgabewert(e) zurückgegeben oder dargestellt werden. (Fowler 2011)

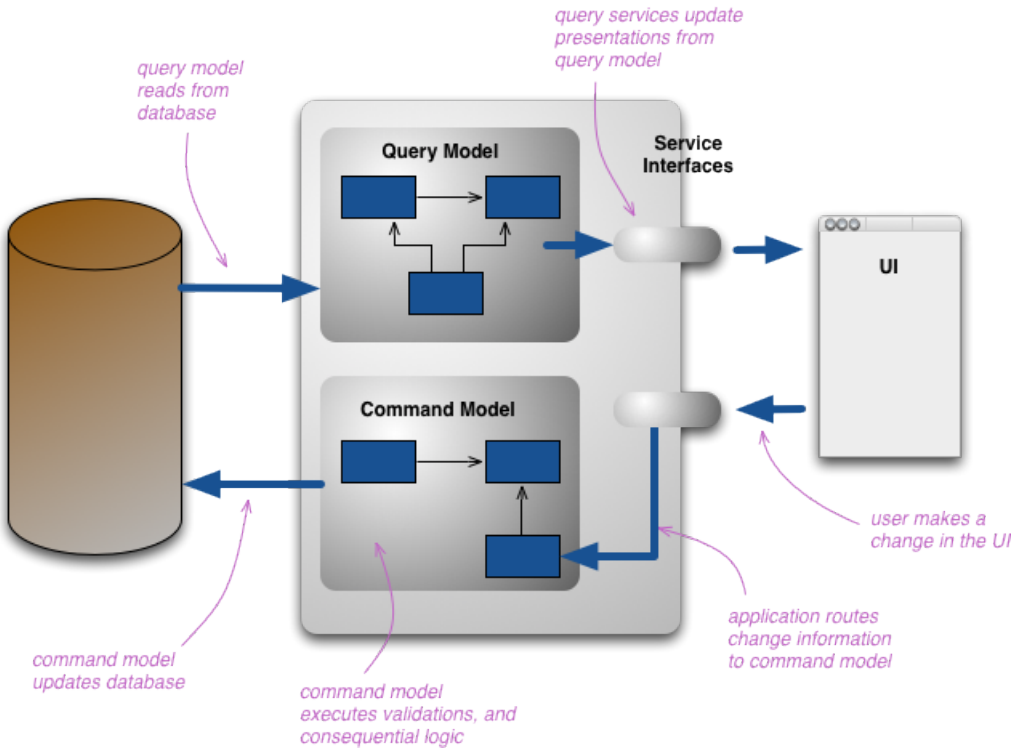


Abbildung 2.2: Schematischer Aufbau eines auf CQRS basierenden Systems. Quelle: Fowler 2011

## 2.4 Vergleich Schichtenarchitektur und CQRS

Mit Blick auf die Abschnitte 2.2 sowie 2.3, die einen Einblick in die von Symfony vorgeschlagene Architektur und das CQRS Pattern geben, werden nachfolgend auf die Vor- beziehungsweise Nachteile der beiden Herangehensweisen eingegangen.

### 2.4.1 Vorteile von CQRS

Die Vorteile von CQRS - im Vergleich zu einer gängigen Schichtenarchitektur beziehungsweise der Herangehensweise von Symfony - bauen im ersten Schritt darauf auf, dass komplexe und umfassende Domänen (und die dazugehörigen Prozesse) durch die Verwendung des Patterns gut aufgeteilt werden können. Das Verwenden von *Commands* und *Queries* ermöglicht eine einfache Handhabung der Code-Basis, indem die Wartbarkeit und Erweiterbarkeit verbessert wird. Darüber hinaus kann die Performanz des gesamten Systems durch die Aufteilung der Aktionen in lesende und schreibende Zugriffe verbessert werden und bietet die Möglichkeit, spezifische Anpassungen je Modell vorzunehmen. (Ingeno 2018, Seite 240)

Die Aufteilung in unabhängige Commands und Queries bietet zudem den Vorteil, dass Applikationen mit einem hohen Performanzanspruch besser eingeteilt und skaliert werden können. Anhand des jeweiligen Anwendungsfalls kann die Last von lesenden und schreibenden Zugriffen unabhängig voneinander getrennt und, sofern notwendig, auch auf mehrere Datenbanksysteme verteilt werden. (Fowler 2011)

Sicherheit, beziehungsweise der besser passende, englische Begriff *Security*, ist ein wichtiger Bestandteil moderner (Web-)Anwendungen. Durch CQRS ist es möglich, den Zugriff und das Bearbeiten von Daten genauer einzugrenzen und nur berechtigten Klassen schreibende Aktionen zu erlauben. Dieser Umstand lässt sich im weiteren Verlauf zudem einfacher testen, was bedeutet, dass ungewollte Sicherheitslücken beziehungsweise das Preisgeben von Daten minimiert werden kann. (Ingeno 2018, Seite 240)

### 2.4.2 Nachteile von CQRS

Neben den Vorteilen, die CQRS mit sich bringt, gibt es jedoch auch Nachteile gegenüber einem Aufbau, der ohne spezifische Auftrennung der Aktionen auskommt (siehe dazu Abschnitt 2.2.2 auf Seite 21).

Für Anwendungen oder APIs, die lediglich einfache *Create*, *Read*, *Update* und *Delete* Funktionalitäten benötigen, stellt die Implementierung von CQRS einen Mehraufwand dar, der womöglich nicht gerechtfertigt ist. Das Pattern bringt

zwar die in Abschnitt 2.4.1 genannten Vorteile mit sich, die Komplexität des Systems steigt dadurch aber gleichzeitig an und eignet sich daher nicht für alle Anwendungsfälle, oder zumindest nicht für eine Implementierung im gesamten System. (Ingeno 2018, Seite 241)

Der Vorteil der Auftrennung in Commands und Queries und damit einhergehenden getrennten Datenbanken führt zu einer weiteren Herausforderung, da sichergestellt werden muss, dass diese untereinander konsistent gehalten werden, um nicht unbeabsichtigt mit veralteten Daten zu arbeiten. (Ingeno 2018, Seite 241)

Heimeshoff und Jander beschreiben weiters den Umstand, dass der Planungs- und Konzeptionsaufwand durch den Einsatz von CQRS bei kleineren Anwendungen nicht direkt im Verhältnis zur Implementierung steht. Die Verlagerung des Fokus muss jedoch nicht unbedingt als ein direkter Nachteil gesehen werden, darf laut ihnen dabei aber nicht außer Acht gelassen werden. (Heimeshoff und Jander 2013)

Auch Fowler rät von einem unbedachten Einsatz des Patterns ab und betont, dass viele Nutzungsfälle auch mit einem typischen Ansatz ohne Trennung auskommen. (Fowler 2011)

# 3 Umsetzung

Dieses Kapitel beleuchtet nachfolgend die Umsetzung der in Abschnitt 1.5 auf Seite 17 beschriebenen Ziele anhand von zwei Beispielen und geht dabei genauer auf die technische Herangehensweise sowie Implementierung des CQRS Patterns ein. Anhand der Beispiele wird des Weiteren auf damit verbundene Schwierigkeiten sowie projektspezifische Eigenheiten eingegangen.

## 3.1 Entwicklungsumgebung

Für die Dauer des Berufspraktikums wurde hauptsächlich die IDE beziehungsweise Entwicklungsumgebung PhpStorm[1] zur aktiven Umsetzung neuer Funktionalitäten im Better Life System verwendet. Selten kam Visual Studio Code[2] zum Einsatz, welches aufgrund von persönlichen Präferenzen für das Auflösen von Merge Konflikten, die während des Rebasings aufgetreten sind, eingesetzt wurde.

Aufgrund der Komplexität des gesamten Projektes und der Notwendigkeit eines Servers, welches das PHP Backend zur Verfügung stellt, sind die einzelnen Komponenten (dazu gehören unter anderem ein Webserver, MySQL Datenbanken, ein Cache-Server sowie ein lokaler Mailserver) jeweils in einem Docker Container untergebracht. Mithilfe eines während des Berufspraktikums erstellten *Makefiles*, welches die am häufigsten verwendeten Befehle (unter anderem starten, stoppen und ausführen weiterer Aktionen) beinhaltet, kann die gesamte Applikation, samt dem dazugehörigen Frontend, lokal aufgesetzt und zur Entwicklung verwendet werden.

[1] PhpStorm (https://www.jetbrains.com/phpstorm)
[2] Visual Studio Code (https://code.visualstudio.com)

Damit zu jedem Zeitpunkt der Entwicklung und auch rückwirkend der jeweilige Stand entsprechend verwendet werden kann, ist ein kompatibler Datenbank Dump in der Git Versionshistorie inkludiert.

## 3.2 Git Branching-Strategie

Das Projekt rund um das Better Life System verfolgt, ähnlich wie andere Projekte der Fusonic GmbH, eine möglichst einheitliche und leicht verständliche Git Branching-Strategie, die das Zusammenarbeiten im Team erleichtern soll.

Neben einem *stage* und *production* Branch, auf denen die Produktivsysteme mit Kundenzugriff basieren, gibt es einen *master* Branch, der als Ausgangspunkt für alle neu entstehenden Funktionalitäten und Codeänderungen dient. Wenn eine neue Funktion entwickelt oder ein Bug behoben werden muss, wird ein sogenannter „Feature Branch“ erstellt, der alle entsprechenden Änderungen beinhaltet. Nach abgeschlossener Entwicklung wird eine Pull- (oder auch Merge-) Request erstellt, ein entsprechendes Code Review vorgenommen und der Feature Branch schlussendlich in den master Branch gemerged.

Die hier beschriebene und beim Better Life System verwendete Herangehensweise orientiert sich am sogenannten „Git Feature Branch Workflow“. Die Entwicklung von neuen Funktionalitäten sowie Codeänderungen werden in einem Feature Branch gekapselt und sind unabhängig vom master Branch, was ein einfaches Zusammenarbeiten ermöglicht. Durch die Verwendung von jeweils separaten Branches und dazugehörigen Pull Requests ist des Weiteren der Vorteil gegeben, dass der *master* Branch zu jedem Zeitpunkt funktionsfähigen Programmcode beinhaltet. Somit kann dieser für eine entsprechende Veröffentlichung oder als Ausgangspunkt für weitere Branches herangezogen werden. (Atlassian 2020)

## 3.3 Implementierung von CQRS anhand des „Region“ API Endpunktes

Das Better Life System unterscheidet in der Verwaltung von Produkten, der Angebotserstellung und anderweitigen Bereichen im System zwischen verschiedenen Regionen. Um dem Frontend entsprechende Informationen bereitstellen zu können, bedarf es eines API Endpunktes, der diese Informationen zur Verfügung stellt und Änderungen entgegen nimmt. Wie die meisten anderen API Endpunkte besteht der „Region“ Endpunkt bereits in der ersten Version (nachfolgend *v1* genannt) beziehungsweise auf Basis der gängigen Schichtenarchitektur (siehe dazu Abschnitt 2.2 auf Seite 19).

Im Zuge der Umstellung auf das CQRS Pattern dient die Version 2 (nachfolgend *v2* genannt) des Region Endpunktes als Beispiel für einen einfach umzustellenden Endpunkt, der ansonsten kaum Abhängigkeiten mit sich bringt und repräsentativ für andere Endpunkte ist.

### 3.3.1 Aufbau des v1 Endpunktes

Der bisherige Aufbau des Region API Endpunktes baut darauf auf, dass der entsprechende `Controller` - welcher schematisch auf Seite 30 dargestellt ist und die Anfragen des Frontends von Symfony zugespielt bekommt - einen Großteil der Ablauflogik übernimmt. Dazu gehört, je nach Typ der Anfrage („GET“, „POST“, „PATCH“ etc.), unter anderem:

- Definition der URI des API Endpunktes
- Authentifizierung des Benutzers
- Zusammentragen benötigter Daten mittels Aufruf des `RegionManagers`
- Serialisierung des `RegionDto` für eine entsprechende Antwort
- Erstellung des tatsächlichen `Response` Objekts

```php
<?php

[...]

final class RegionController
{
    /**
     * @Route("/regions", name="get_regions", methods={"GET"})
     */
    public function cgetAction(Request $request): Response
    {
        // Authentication
        // Several function calls of "RegionManager"
        // Preperation of Data Transfer Objects
        // Returning response with DTOs
    }

    /**
     * @Route("/regions/{id}", name="get_region", methods={"GET"})
     */
    public function getAction(int $id): Response { /* Implementation */ }

    /**
     * @Route("/regions/{id}", name="patch_region", methods={"PATCH"})
     */
    public function patchAction(int $id, Request $request): Response { /*
    ↪  Implementation */ }

    [...]
}
```

Quellcode 2: Schematischer Aufbau des v1 „RegionController“

Der Controller beinhaltet zudem Methoden, die bei der Serialisierung aufgerufen werden und festlegen, welche Attribute nach außen hin benötigt und daher mit Daten befüllt werden müssen.

Der angesprochene `RegionManager`, welcher vom Controller aufgerufen wird, um die entsprechenden Daten zu aggregieren, beinhaltet eine Vielzahl an Methoden und Funktionalitäten. Diese umfassen sowohl spezifische Anwendungs- beziehungsweise Businesslogik als auch allgemein gehaltene Funktionalitäten, die über die klassische objektorientierte Vererbung sowie über die von PHP eingesetzten `Traits` zur Verfügung gestellt werden. Der Einsatz des Managers

- der in weiterer Folge das von Doctrine verwendete `Repository` (in diesem Fall dem `RegionRepository`) aufruft - hat zur Folge, dass lesende und schreibende Zugriffe in derselben Klasse verwendet und mit sonstiger Businesslogik vermischt werden.

Die Abbildung 3.1 auf Seite 31 gibt nochmals einen Überblick über den oben beschriebenen Aufbau des v1 Endpunktes anhand eines grundlegenden UML Diagramms.

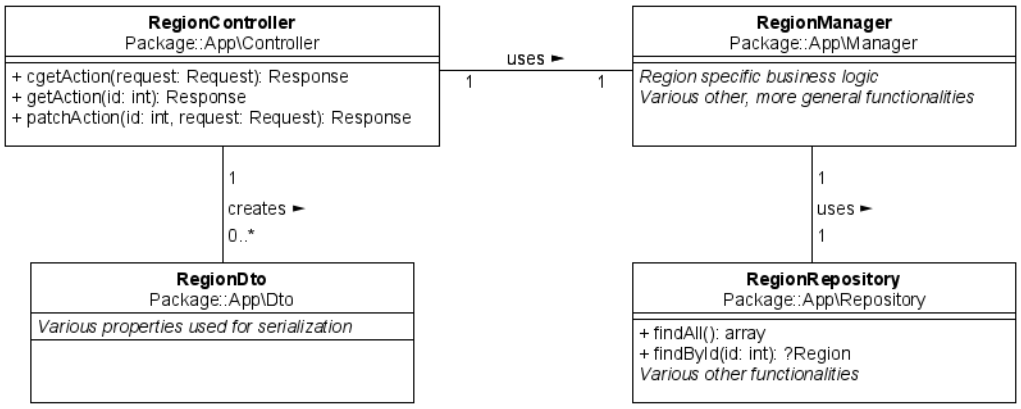


Abbildung 3.1: Grundlegender Aufbau des v1 „Region“ Endpunktes samt interner Abhängigkeiten

### 3.3.2 Aufbau des v2 Endpunktes

Für die Umsetzung des CQRS Patterns mit dem Symfony Framework wird im Better Life System auf die von Symfony zur Verfügung gestellte `Messenger`[3] Komponente zurückgegriffen. Diese ermöglicht es, Commands und Queries in sogenannte `Envelopes` zu verpacken und von einem entsprechenden `Command`- beziehungsweise `QueryHandler` abarbeiten zu lassen. Diese Komponente kommt mit einer Vielzahl von weiteren Funktionalitäten, Konzepten und Patterns, welche im Verlauf dieser Bachelorarbeit jedoch nicht weiter beleuchtet werden.

[3]The Messenger Component - (https://symfony.com/doc/4.4/components/messenger.html)

Die Struktur des in Quellcode 2 auf Seite 30 aufgezeigten Controllers bleibt auch in der CQRS-Version nahezu gleich. Primär verändert sich hierbei die Verantwortung und damit die Größe der einzelnen Methoden, da diese nur noch für die Authentifizierung und Rückgabe der von Symfony benötigten Response zuständig sind. Jegliche weitere Ablauf- und/oder Businesslogik wurde in den jeweiligen Command- beziehungsweise QueryHandler verschoben, um eine strikte Trennung gewährleisten zu können.

Der Quellcode 3 auf Seite 32 zeigt den Aufbau der v2 Struktur auf, bei der bereits deutlich die Trennung von Commands und Queries ersichtlich ist, ohne die zugrundeliegende Logik zu kennen.

```
src/
    Region/
        Controller/
            RegionController.php
        Message/
            Command/
                UpdateRegionCommand.php
            CommandHandler/
                UpdateRegionCommandHandler.php
            Query/
                GetAllRegionsQuery.php
                GetRegionQuery.php
            QueryHandler/
                GetAllRegionsQueryHandler.php
                GetRegionQueryHandler.php
            Response/
                RegionResponse.php
        Repository/
```

Quellcode 3: Ordnerstruktur des v2 Region Endpunktes

#### 3.3.2.1 Das „UpdateRegionCommand“ und der entsprechende CommandHandler

Wie in Abschnitt 2.3.1 auf Seite 23 beschrieben, werden schreibende und lesende Aktionen voneinander getrennt. Das Updaten beziehungsweise Aktualisieren einer Region entspricht einer schreibenden Aktion und wird daher mit einem Command abgehandelt.

Ein Command besteht in der im BLS praktizierten Umsetzung aus privaten Attributen und dazugehörigen Gettern und Settern, die öffentlich aufgerufen werden können. Die Daten, die für die Aktualisierung herangezogen werden sollen, sind bereits vom Angular Frontend validiert, werden - wie in Quellcode 4 auf Seite 33 ersichtlich ist - jedoch auch noch einmal vom Backend überprüft, um fehlerhafte Eingaben vorzubeugen. Die angewandte Validierung sowie das vorherige Setzen der Daten wird hierbei von Symfony in Zusammenspiel mit entsprechenden Erweiterungen vorgenommen.

```php
<?php

[...]

final class UpdateRegionCommand
{
    /**
     * @Assert\NotNull
     * @Assert\Positive(message="Id should be a positive integer.")
     */
    private int $id;

    /**
     * @Assert\NotBlank(message="Regulations should not be blank.")
     * @Assert\Length(max=65535)
     */
    private string $regulations;

    // Publicly accessible getters

    // Publicly accessible setters
}
```

Quellcode 4: Die „UpdateRegionCommand“ Klasse

Der zum `UpdateRegionCommand` dazugehörende CommandHandler umfasst jene Ablauf- und Businesslogik, die sich zuvor im v1 `RegionController` sowie `RegionManager` befunden hat und für das Aktualisieren der Daten tatsächlich benötigt wird. Da der Region Endpunkt in der bereits bestehenden Version ohne viel Logik auskommt, spiegelt sich dies auch in der neuen Umsetzung wider, wie der Quellcode 5 auf Seite 34 aufzeigt.

```php
<?php

[...]

final class UpdateRegionCommandHandler
{
    // Dependency injection of Doctrine repository, etc.

    public function __invoke(UpdateRegionCommand $command): RegionResponse
    {
        $region = $this->repo->findById($command->getId());
        if (!$region) {
            throw new HttpNotFoundException();
        }

        $region->update($command->getRegulations());
        $this->repo->saveEntity($region);

        return new RegionResponse($region);
    }
}
```

Quellcode 5: Die „UpdateRegionCommandHandler“ Klasse

Dem CommandHandler wird, anhand des Typs der beim Formalparameter `$command` der `__invoke` Funktion angeben ist, das entsprechende UpdateRegionCommand von Symfony übergeben. Im weiteren Verlauf wird das bereits bestehende Entity aus der Datenbank entnommen, mit den neuen Daten aktualisiert und entsprechend wieder abgespeichert.

Im Regelfall werden schreibende Aktionen beziehungsweise Commands ohne einen Rückgabewert abgeschlossen (Fowler 2005). In der im Better Life System angewandten Umsetzung wird hierbei jedoch ebenfalls eine Rückgabe in Form einer `RegionResponse` erzeugt, damit das Frontend die aktualisierten Daten ohne weitere API Aufrufe verwenden kann. Die RegionResponse gleicht Responses anderer v2 Endpunkte und besteht primär aus einer globalen Variable, in der das Entity abgespeichert ist. Zudem beinhaltet sie öffentlich aufrufbaren Getter, die Symfony im Zuge der Serialisierung aufruft, um einen entsprechenden JSON String zu erzeugen.

#### 3.3.2.2 Das „GetRegionQuery“ und der entsprechende QueryHandler

Das Query, welches genutzt wird um eine Region auszulesen und dem Frontend zur Verfügung stellen zu können, gleicht im Endeffekt dem auf Seite 33 dargestellten Quellcode 4. Bei Queries sind im Regelfall jedoch nur eine globale `$id` Variable und gegebenenfalls Werte für die Pagnination (sprich die Unterteilung der Daten auf mehrere, kleinere Blöcke) samt dazugehörigen Gettern und Settern gegeben. Etwaige anderweitige Daten spielen hierbei keine Rolle, da es sich bei einem Query um einen lesenden Zugriff handelt.

Ähnlich wie in Abschnitt 3.3.2.1 auf Seite 32 kommt auch bei einem Query ein entsprechender QueryHandler zum Einsatz, der das Entity aus der Datenbank ausliest und eine `RegionResponse` als Rückgabewert zur Folge hat. Im direkten Vergleich von Quellcode 5 (Seite 34) und Quellcode 6 (Seite 35) ist ersichtlich, dass der `GetRegionQueryHandler` keinerlei Änderung an bestehenden Daten vornimmt und - wie bei CQRS beschrieben - ausschließlich für einen lesenden Zugriff zuständig ist.

```php
<?php

[...]

final class GetRegionQueryHandler
{
    // Dependency injection of Doctrine repository, etc.

    public function __invoke(GetRegionQuery $query): RegionResponse
    {
        $region = $this->repo->findById($query->getId());
        if (!$region) {
            throw new HttpNotFoundException();
        }

        return new RegionResponse($region);
    }
}
```

Quellcode 6: Die „GetRegionQueryHandler“ Klasse

## 3.4 Testen von CQRS anhand des „ProjectContact“ API Endpunktes

Der neue „ProjectContact“ Endpunkt, samt der dazugehörigen (Business-)Logik, ist eine verbesserte sowie erweiterte Umsetzung einer in gewissen Teilen bereits bestehenden Kontaktverwaltung. Diese umfasst Personen beziehungsweise Ansprechpartner, die zu einem Projekt zugehörig sind. Durch die Anforderung des Kunden, dass der Client online sowie offline verwendet werden können muss, muss für größere Änderungen an bestehenden API Schnittstellen eine Rückwärtskompatibilität gewährleistet werden.

Die tatsächliche Umsetzung aus Sicht von CQRS unterscheidet sich nur wenig von der bereits beschriebenen Herangehensweise und den Charakteristiken, die CQRS im Better Life System aufweist (siehe dazu Abschnitt 3.3 auf Seite 29). Die Schwierigkeit stellt in diesem Fall primär die bestehende Logik im `ProjectManager`, `ProjectDto` und anderweitigen Klassen dar, die weiterhin - trotz der intern veränderten Strukturen - entsprechend funktionieren muss.

Während dem Arbeiten an den neuen Funktionalitäten und der Einarbeitung der Änderungen in die bestehenden API Schnittstellen spielten die bereits bestehenden Unit Tests - von denen es zum Zeitpunkt des Schreibens dieser Arbeit mehr als 1000 im Better Life System gibt - eine wichtige Rolle bei der Aufrechterhaltung der Rückwärtskompatibilität.

### 3.4.1 Bestehende Unit Tests und ihr Aufbau

Die zu einem früheren Zeitpunkt erstellten Unit Tests, die die Logik des „Project“ API Endpunktes (sowie der weiteren Endpunkte) überprüfen, verfolgen alle ein ähnliches Konzept. Zuerst wird eine Anfrage an das PHP Backend simuliert, dann die Antwort beziehungsweise `HTTP Response` ausgewertet und abschließend auf spezifische Merkmale getestet.

Der Quellcode 7 auf Seite 37 repräsentiert einen (stark vereinfachten, aber typischen) Test, der vom Aufruf hin bis zur tatsächlichen Antwort und der Anzahl an Datenbankaufrufe alle Komponenten testet und dadurch auf Fehler hindeuten kann.

```php
<?php

[...]

class ProjectControllerTest
{
    public function testGetProjects(): void
    {
        $this->makeJsonRequest('GET', '/api/projects');
        $response = \json_decode(
            self::$client->getResponse()->getContent(),
            true
        );
        self::assertSame(
            Response::HTTP_OK,
            self::$client->getResponse()->getStatusCode()
        );

        self::assertCount(1, $response['_embedded'];
        // Other specific response validation
        $this->validateProjectData($response['_embedded'][0]);
        self::assertSame(5, $this->getNumberOfDbQueries());
    }

    [...]
}
```

Quellcode 7: Vereinfacht dargestellter v1 Unit Test

### 3.4.2 Unit Tests für CQRS Komponenten

Im Vergleich zu den „herkömmlichen“ beziehungsweise bisher im BLS eingesetzten Unit Tests testen die auf CQRS ausgelegten Unit Tests deutlich weniger, als - wie bisher - den kompletten Ablauf von simulierter Anfrage bis hin zur Rückgabe. Primär wird auf das Testen der Logik des entsprechenden `CommandHandler` oder `QueryHandler` gesetzt, wie der Quellcode 8 auf Seite 38 aufzeigt.

Der Einsatz von deutlich spezifischeren Tests bringt den Vorteil mit sich, dass fehlgeschlagene Tests deutlich präziser darauf hinweisen können, wo Fehler aufgetreten sind. Zudem lässt sich mittels *Mocking* der Einsatz einer tatsächlich existierenden Testdatenbank vermeiden, was die Ausführbarkeit und Performanz deutlich erhöht und externe Fehlerquellen ausschließt.

```php
<?php

[...]

class GetProjectContactQueryHandlerTest
{
    public function testGetProjectContact(): void
    {
        $query = new GetProjectContactQuery();
        $query->setProjectId(1);
        $query->setContactId(1);

        $projectRepo = $this->createMock(ProjectRepositoryInterface::class);
        $projectRepo
            ->method('findById')
            ->with($query->getProjectId())
            ->willReturn(new Project());

        // Mocks of other required Doctrine repositories and return values

        $handler = new GetProjectContactQueryHandler([...]);
        $response = $handler($query);

        self::assertInstanceOf(ProjectContactResponse::class, $response);
        self::assertSame($query->getContactId(), $response->getContactId());
        // Other assertions
    }

    [...]
}
```

Quellcode 8: Auf das Testen von CQRS ausgelegter Unit Test

# 4 Zusammenfassung & Ausblick

Das nachfolgende Kapitel fasst die während des Berufspraktikums durchgeführten Tätigkeiten noch einmal zusammen, gibt einen Überblick über die erreichten Ergebnisse und wägt ab, ob die geplante Zielsetzung erreicht wurde. Des Weiteren wird ein Ausblick auf das Better Life System und die damit in Zusammenhang stehende Verwendung des CQRS Patterns gegeben.

## 4.1 Zusammenfassung

Aufgrund interner Erfahrungen und Evaluierungen stand bereits zu Beginn des Berufspraktikums fest, dass das Better Life System sukzessive von einer klassischen Schichtenarchitektur auf das CQRS Pattern umgestellt werden soll. Im Zuge der Einführung und Einarbeitung in das seit längerem bestehende Projekt wurden grundlegende Ziele beziehungsweise Aufgaben definiert. Zu den Zielen, die in Abschnitt 1.5 auf Seite 17 angeführt sind, gehören die Umstellung bestehender API Endpunkte, die Umsetzung neuer Funktionalitäten mittels CQRS und das Abwickeln des Tagesgeschäftes.

Die Anforderungen der Viterma Handels GmbH an Fusonic waren beziehungsweise sind, die Umstellung auf die neue Herangehensweise parallel zur allgemeinen Weiterentwicklung und Implementierung neuer Funktionalitäten durchzuführen. Daraus hat sich ergeben, dass zu Beginn zuerst neue Funktionalitäten implementiert und die Grundlagen für die Verwendung von CQRS im bestehenden System geschaffen wurden. Ab der Mitte des dreieinhalb Monate dauernden Berufspraktikums wurde dann der Fokus auf die Umstellung kleinerer und leicht anpassbarer API Endpunkte gelegt, während neu benötigte Funktionalitäten ab diesem Zeitpunkt bereits mittels CQRS umgesetzt wurden.

Mit Abschluss des Berufspraktikums und der damit einhergehenden Übergabe an die restlichen Mitglieder des Entwicklungsteams konnte festgestellt werden, dass die zu Beginn definierten Ziele erfüllt wurden. Das laufende Tagesgeschäft (sprich kleinere Änderungen, Bugfixes etc.) sowie die Umstellung und Neuentwicklung ausgewählter Endpunkte konnte ohne größere Probleme abgewickelt beziehungsweise erreicht werden.

Neben den definierten Zielen wurde während der Entwicklung ein Augenmerk auf eine möglichst hohe Codequalität gelegt, die mittels Code Review und dem Einsatz von entsprechend getestetem Programmcode erreicht wurde. Bestehende Tests wurden dabei für abgeänderte Funktionalitäten angepasst, während neue Unit Tests sowohl für den bisher verwendeten Ansatz der Schichtenarchitektur als auch für CQRS Komponenten erstellt wurden. Fortlaufende Code Reviews wurden im Zuge von Pull Requests stetig von allen Mitgliedern des Entwicklungsteams vorgenommen.

## 4.2 CQRS in Retrospektive

TBD

## 4.3 Ausblick

TBD

# Literatur


Atlassian (2020). *Git Feature Branch Workflow*. en. URL: https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow (besucht am 13. 09. 2020).

Fowler, Martin (Dez. 2005). *CommandQuerySeparation*. URL: https://martinfowler.com/bliki/CommandQuerySeparation.html (besucht am 11. 08. 2020).

– (Juli 2011). *CQRS*. URL: https://martinfowler.com/bliki/CQRS.html (besucht am 10. 08. 2020).

Fusonic GmbH (2020). *Web, Mobile, Desktop-Anwendungen*. de. URL: https://www.fusonic.net/de/leistungen/ (besucht am 04. 08. 2020).

Heimeshoff, Marco und Philip Jander (Feb. 2013). *CQRS – neues Architekturprinzip zur Trennung von Befehlen und Abfragen*. de. URL: https://www.heise.de/developer/artikel/CQRS-neues-Architekturprinzip-zur-Trennung-von-Befehlen-und-Abfragen-1797489.html?seite=all (besucht am 10. 08. 2020).

Ingeno, Joseph (Aug. 2018). *Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts*. English. Birmingham: Packt Publishing. ISBN: 978-1-78862-406-0.

Symfony (2020). *The Symfony Framework Best Practices (Symfony Best Practices)*. en. Library Catalog: symfony.com. URL: https://symfony.com/doc/current/best_practices.html (besucht am 06. 08. 2020).

Tutorials Point (I) Pvt. Ltd. (2020a). *Symfony - Architecture*. URL: https://www.tutorialspoint.com/symfony/symfony_architecture.htm (besucht am 07. 08. 2020).

– (2020b). *Symfony - Introduction*. URL: https://www.tutorialspoint.com/symfony/symfony_introduction.htm (besucht am 07. 08. 2020).

Young, Greg (Nov. 2010). *CQRS Introduction*. en. URL: https://cqrs.wordpress.com/documents/cqrs-introduction/ (besucht am 09. 08. 2020).

# Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit I selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Dornbirn, am 30. November 2020 Dominic Luidold