Cloud-Native Refactoring in einem mHealth Szenario

Bachelorarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Frühjahrssemester 2019

Autoren: Moritz Habegger & Micha Schena
Betreuer: Prof. Dr. Olaf Zimmermann
Experte: Dr. Gerald Reif
Gegenleser: Prof. Oliver Augenstein

1 Abstract

1.1 Aufgabenstellung

Die Bachelorarbeit untersucht eine bestehende Spring Boot Integrationslösung, die einen Ende-zu-Ende verschlüsselten Kommunikationskanal zwischen Gesundheitsdienstleistern und deren Patientengut anbietet. Diese Applikation weist eine monolithische Architektur auf, die mithilfe eines cloud-native Refactorings auf eine Microservice-Architektur flexibilisiert werden soll. Microservices sind ein branchenübergreifender Software-Architektur-Trend. Die Arbeit soll zeigen, ob und wie ein solcher Umbau in diesem Szenario möglich ist und was die Konsequenzen hinsichtlich Wirtschaftlichkeit, Stabilität und Flexibilität sind.

1.2 Vorgehen

Im Rahmen der Analyse wurden die funktionalen sowie nicht funktionalen Anforderungen an die umzubauende Applikation erfasst und die bestehende Implementierung mit der Modellierungstechnik C4 dokumentiert. Ein Event Storming lieferte eine Geschäftsprozess-Modellierung, die die Anforderungen tiefer erfasst und auch auf konkrete Bounded Contexts aus dem Domain-Driven Design hinweist, die dann wiederum in Microservices abgebildet werden können. Das erarbeitete Verständnis für die Applikation führte zu mehreren Microservice-Architekturvorschlägen, deren Eignung systematisch evaluiert wurde. Der letzte Abschnitt der Arbeit setzte den ausgewählten Architekturvorschlag prototypisch mit Spring Cloud und RabbitMQ um.

1.3 Ergebnis

Die monolithische Applikation wurde erfolgreich zu autonomen Microservices umgebaut. Das entwickelte und erprobte Vorgehen hat sich als effizient und zielführend erwiesen und ist so gewählt, dass es sich auch auf andere Branchen und Szenarien übertragen lässt. Gewisse Architektur-Entscheidungen in der bestehenden Applikation haben den Umbau zu Microservices erschwert, welche in einem neuen Projekt vermieden werden sollten. Eine Ergebnisdiskussion stellt die Vor- und Nachteile, beziehungsweise die Konsequenzen der umgesetzten Architektur kritisch gegenüber.

2 Management Summary

2.1 Ausgangslage

Viele Softwarelösungen weisen eine monolithische Architektur auf, das bedeutet, sie sind ein homogenes Gebilde aus funktionalen Elementen. In den letzten Jahren gibt es einen Trend solche Systeme zu modularisieren und in einzelne autonome Microservices aufzuteilen. Gerade grosse Firmen wie Netflix oder Spotify, die Millionen von Benutzer bedienen, setzen auf diesen Trend, um es einer grossen Anzahl von Entwickler-Teams zu ermöglichen, schneller und unabhängiger neue Änderungen einzuführen.

Diese Arbeit untersucht, ob und wie ein Umbau einer Softwarelösung der healthinal GmbH zu einer solchen Microservice-Architektur, möglich ist, die einen verschlüsselten Kommunikationskanal zwischen Gesundheitsdienstleistern und deren Patientengut anbietet. Anschliessend wird die entstandene Architektur bezüglich Wirtschaftlichkeit, Flexibilität und Stabilität bewertet.

2.2 Vorgehen / Technologien

Untersucht wird die Aufteilung des Monolithen in Microservices nach Domänen. Dies entspricht einem gängigen Vorgehen zur Aufteilung von Microservices und wird Domain-Driven Design (DDD) genannt. Im Rahmen der Analyse werden die funktionalen sowie nicht funktionalen Anforderungen an die umzubauende Applikation erfasst und die bestehende Implementierung mit der Modellierungstechnik C4 dokumentiert. Der zu behandelnde Geschäftsprozess wird mit der Workshop-Technik Event Storming modelliert. Diese Technik ist eine gängige Praxis zur Identifizierung von Domänen für die Microservices im Kontext von DDD. Zusätzlich wird die Open-Source-Software Context Mapper mit Service Cutter eingesetzt, welche mögliche Domänen anhand von Algorithmen sucht, die wiederum mit den Domänen des Event Storming verglichen werden. Mit dem erarbeiteten Verständnis für die Applikation und den Anforderungen werden Architekturvorschläge einander kritisch gegenübergestellt. Anschliessend wird der ausgewählte Architekturvorschlag prototypisch umgesetzt und bewertet.

2.3 Ergebnisse

Das Resultat der Arbeit ist zum einen das dokumentierte Vorgehen, das sich auch auf andere Branchen und Szenarien übertragen lässt, zum anderen das cloud-native Refactoring der vorgegebenen Applikation.

Das Vorgehen hat sich als zielführend erwiesen. Das iterative Herauslösen von Microservices aus dem Monolithen ist effizienter, als diese von Grund auf neu zu programmieren. So bleibt die Applikation nach jeder Iteration in einem ausführbaren Zustand und kann getestet werden. Wenn kein iteratives Vorgehen angewendet wird und die Applikation in einem Schritt umgebaut wird, kann erst am Schluss festgestellt werden, ob die geplanten Microservices technisch sinnvoll umgesetzt werden können.

Die vorgegebene Applikation wurde mit dem cloud-native Refactoring erfolgreich in eine Microservice-Architektur umgebaut. Angewendet wurden drei Refactoring-Iterationen, die zu drei autonomen Microservices führten. Um die Microservices zeitlich voneinander zu entkoppeln und somit deren vollständige Autonomie zu erreichen, wird die nachrichtenorientierte Middleware RabbitMQ eingesetzt.

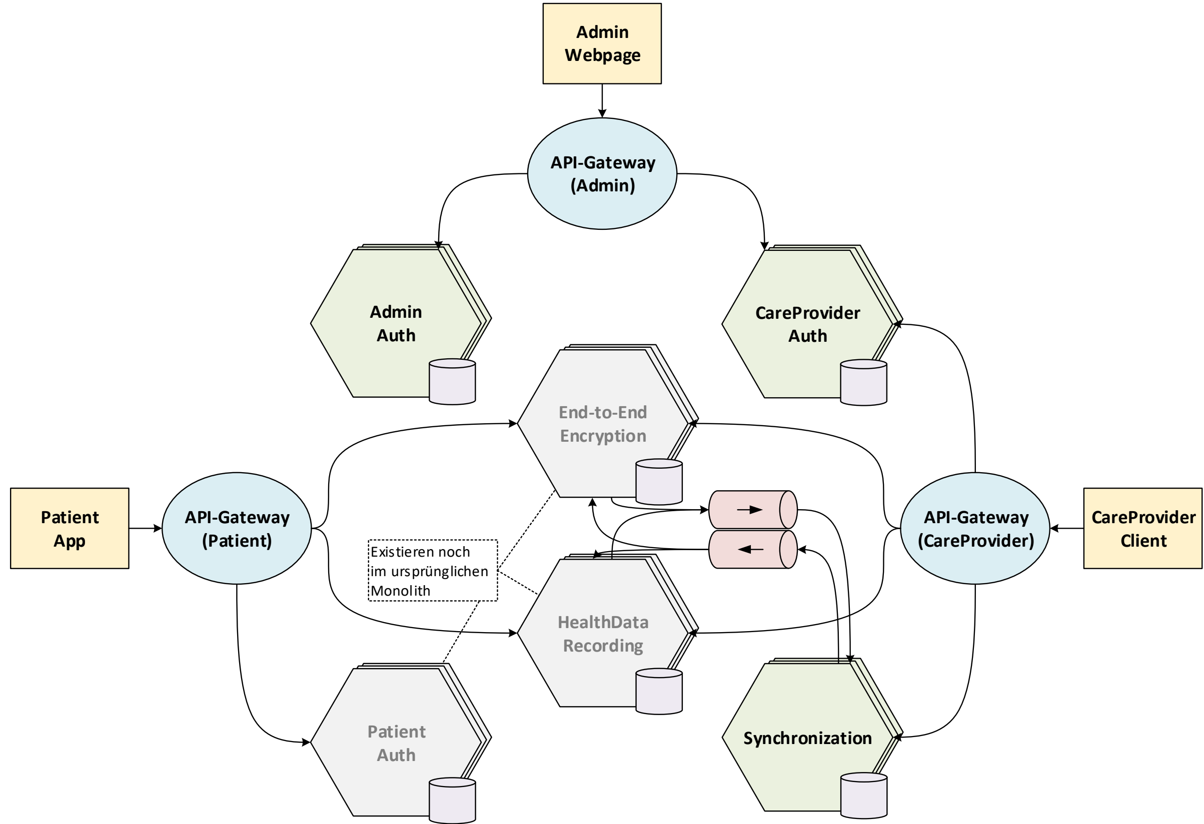


Abbildung 1 Umgesetzte Architektur des Prototyps

Wirtschaftlich betrachtet liefert der Umbau der vorgegebenen Applikation zu einer Microservice-Architektur, welche in Abbildung 1 dargestellt ist, keinen Mehrwert, sondern verursacht einen deutlich höheren Betriebs- und Konfigurationsaufwand, aufwändigere Integrationstests und somit einen grösseren Aufwand für die Entwickler. Das ist darauf zurückzuführen, dass der Umfang der Applikation eher klein ist. Erst wenn mehrere Teams an einer solchen Applikation arbeiten, überwiegen die Vorteile einer Microservice-Architektur in Bezug auf die Wirtschaftlichkeit und ermöglichen es den Entwicklern unabhängiger und somit effizienter zu arbeiten.

Ob die Stabilität der entstandenen Microservice-Architektur verbessert ist, kann nicht abschliessend beurteilt werden. Einerseits können die Microservices nun autonom agieren und erlauben somit den Ausfall eines Microservices, wie der *Synchronization Service*, ohne dass die restlichen Microservices betroffen sind, was die Stabilität erhöht. Andererseits führt die verteilte Bereitstellung und zusätzlichen Technologien zu einer komplexeren Bereitstellung, die fehleranfälliger ist und somit die Stabilität negativ beeinflussen kann. Ein Team, das Erfahrung im Betrieb von Microservices hat, ist jedoch vermutlich in der Lage, eine höhere Stabilität mit dieser Architektur zu erreichen. Falls die Ressourcen und Erfahrung für diese Aufgabe nicht vorhanden sind, wird die Stabilität nicht erhöht oder sogar negativ beeinflusst.

Die Microservices sind im Prototyp mit einem hohen Grad an Autonomie implementiert, was darauf hinweist, dass die Domänen-Grenzen im Event Storming gut gewählt wurden und somit für Erweiterungen eine flexible Basis bilden.

Aufgrund der oben aufgeführten Punkte wird für die behandelte Applikation von einem Umbau zu einer Microservice-Architektur abgeraten. Ein solcher wird sich erst lohnen, wenn mehrere Entwickler-Teams an dieser Applikation arbeiten. Deshalb empfiehlt es sich bereits jetzt Vorkehrungen zu treffen, die ein cloud-native Refactoring in der Zukunft erleichtern.

Inhalt

1 Abstract ... 1
1.1 Aufgabenstellung ... 1
1.2 Vorgehen ... 1
1.3 Ergebnis ... 1
2 Management Summary ... 2
2.1 Ausgangslage ... 2
2.2 Vorgehen / Technologien ... 2
2.3 Ergebnisse ... 2
3 Ausgangslage (Kontext) ... 7
3.1 Typ der Arbeit ... 7
3.2 Heutige Lösung ... 7
4 Vorgehen ... 8
5 Problemanalyse ... 10
5.1 Motivation ... 10
5.2 Aktuelle Architektur als C4-Modell ... 10
5.2.1 Stufe 1: System Context Diagram ... 11
5.2.2 Stufe 2: Container Diagram ... 13
5.2.3 Stufe 3: Component Diagram ... 15
5.3 Nicht funktionale Anforderungen ... 18
5.4 Domänen-Analyse mithilfe von Event Storming ... 19
5.4.1 Erwartetes Ergebnis ... 19
5.4.2 Vorgehen ... 19
5.4.3 Notation der Klebezettel ... 19
5.4.4 Schritt 1: Domain Events erfassen ... 19
5.4.5 Schritt 2: Commands erfassen ... 20
5.4.6 Schritt 3: Entity / Aggregates erfassen ... 21
5.4.7 Schritt 4: Bounded Context definieren ... 22
5.4.8 Resultat des Event Stormings / Entstandene Domänen ... 23
5.4.9 Vergleich der Bounded Contexts mit aktueller Implementierung ... 23
5.4.10 Use Cases aus Event Storming ... 24
5.5 Domänen-Aufteilung mithilfe Context Mapper und Service Cutter ... 25
5.5.1 Context Mapper ... 25
5.5.2 Service Cutter ... 26
5.5.3 Resultat ... 26
5.6 Ergebnis Problemanalyse ... 27

6 Lösungsfindung ... 28
6.1 Architekturvarianten (Longlist) ... 28
6.1.1 Notation ... 28
6.1.2 Monolith (aktuelle Implementierung des *Synchronization Servers*) ... 29
6.1.3 Variante 1: Single-Gateway-Monolith ... 29
6.1.4 Variante 2: Multi-Gateway-Monolith ... 30
6.1.5 Variante 3: Monolith-Per-Careprovider ... 30
6.1.6 Variante 4: Microservice-Direct-Invocation ... 31
6.1.7 Variante 5: Single-Gateway-Multi-Microservice ... 32
6.1.8 Variante 6: Multi-Gateway-Multi-Microservice ... 32
6.1.9 Variante 7: Gateway-Queue-Microservice ... 33
6.1.10 Variante 8: Serverless-Direct-Database ... 34
6.1.11 Resultat Longlist-Architekturvarianten ... 35
6.2 Architekturvorschläge (Shortlist) ... 36
6.2.1 Microservice-Architekturvorschlag 1 (Ausgelagerte Synchronisation) ... 36
6.2.2 Microservice-Architekturvorschlag 2 (Entkoppelte Synchronisation) ... 38
6.2.3 Microservice-Architekturvorschlag 3 (CareProvider Queues) ... 39
6.3 Vergleich der Architekturvorschläge ... 40
6.3.1 Vergleich anhand NFA ... 40
6.3.2 Vergleich anhand der sieben Microservice Tenets ... 41
6.3.3 Vergleich des Aufwandes ... 43
6.4 Architekturvorschlag Entscheid ... 43
7 Umsetzung ... 44
7.1 Implementierungsdetails ... 44
7.1.1 Schritt 1: Authentifizierung zu JWT umbauen ... 44
7.1.2 Schritt 2: API-Gateway vorschalten ... 44
7.1.3 Schritt 3: Duplikat des Monolithen für den *CareProviderAuth-Service* erstellen ... 45
7.1.4 Schritt 4: Duplikat des Monolithen für den *AdminAuth-Service* erstellen ... 46
7.1.5 Schritt 5: Duplikat des Monolithen für den *Synchronization-Service* erstellen ... 46
7.1.6 Schritt 6: Auftrennen der Datenbank ... 46
7.1.7 Schritt 7: Asynchrone Microservice-Kommunikation einführen ... 47
7.1.8 Fazit ... 47
7.2 Testing ... 48
7.3 Review der nicht funktionalen Anforderungen ... 51
8 Ergebnisdiskussion ... 54
8.1 Vorgehensweise ... 54

8.2 Umgesetzter Prototyp 54
8.2.1 Wirtschaftlichkeit 55
8.2.2 Stabilität 55
8.2.3 Flexibilität 56
8.2.4 Adressierung der NFA 6 56
8.3 Empfehlung 56
9 Ausblick 57
10 Glossar 58
11 Quellen/Referenzverzeichnis 61
11.1 Tabellenverzeichnis 61
11.2 Abbildungsverzeichnis 61
11.3 Literaturverzeichnis 62
12 Anhänge 66
12.1 Testprotokolle NFA 66
12.1.1 Testprotokoll NFA 4 66
12.1.2 Testprotokoll NFA 4 & 5 Loadtest 67
12.1.3 Testprotokoll NFA 8 & 9 70
12.1.4 Testprotokoll NFA 10 71

3 Ausgangslage (Kontext)

3.1 Typ der Arbeit

Diese Arbeit untersucht, ob und wie ein Umbau einer Softwarelösung der healthinal GmbH zu einer Microservice-Architektur, möglich ist. Diese Applikation bietet einen verschlüsselten Kommunikationskanal zwischen Gesundheitsdienstleistern und deren Patientengut an. Es wird ein diskursives Vorgehen mit branchenüblichen Architektur-Analyse-Methoden zur Findung von Architekturvorschlägen angestrebt.

3.2 Heutige Lösung

Die aktuelle Lösung ist als monolithische Spring Boot Applikation [1] mit einer relationalen Datenbank implementiert. Diese Applikation wird noch nicht produktiv eingesetzt. Ihre Hauptaufgabe besteht darin, die Kommunikation zwischen einem Gesundheitsdienstleister und seinem Patientengut zeitlich zu entkoppeln.

In der Testumgebung konnte ein Szenario mit einem fehlenden Index auf der Datenbank simuliert werden, welches die Applikation mit bestimmten Anfragen zum Blockieren brachte. Folglich konnte keine andere Partei mehr mit der Applikation kommunizieren, bis das Problem in der Datenbank behoben wurde.

Darum und aus generellem Interesse an Microservices wird ein cloud-native Refactoring durchgeführt und soll bezüglich Wirtschaftlichkeit, Flexibilität und Stabilität bewertet werden.

4 Vorgehen

Dieser Abschnitt fasst zusammen, wie in dieser Arbeit vorgegangen wird, um die bestehende Applikation zu einer Microservice-Architektur umzubauen. Die weiteren Kapitel dieses Berichtes gehen dann genauer auf die jeweiligen Techniken ein und erläutern deren Resultate.

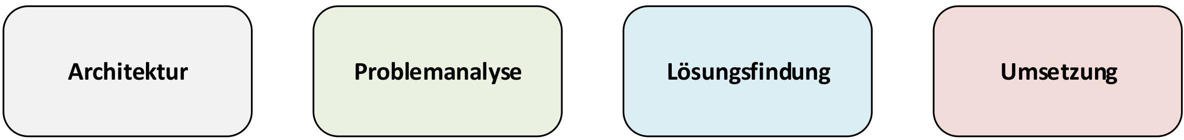


Abbildung 2 Legende: Vorgehen in einem cloud-native Refactoring

Monolithische Architektur

Technische Analyse
C4-Modellierung

Technische Analyse
CRC-Karten

Nicht funktionale Anforderungen

Funktionale Anforderungen
Event Storming

Longlist
Architekturvarianten

Shortlist
Architekturvorschläge

Vergleich
Architekturvorschläge

Architektur-Entscheid

Rahmenbedingungen schaffen
(z. B. Authentication)

API-Gateway erstellen

Monolith duplizieren

Microservice herauslösen

API-Gateway konfigurieren

Datenbank trennen

Microservice Architektur

Abbildung 3 Vorgehen in einem cloud-native Refactoring

Das im Verlauf dieser Arbeit entwickelte Vorgehen, ist nicht an eine spezifische Applikation gebunden, sondern hat den Anspruch, auch für andere cloud-native Refactorings relevant zu sein. Es beschreibt, wie eine monolithische Architektur zu einer Microservice-Architektur umgebaut werden kann.

Nachfolgende Tabelle erläutert die jeweiligen Schritte spezifischer.

<table>
<tr><th></th><th colspan="2">Technik</th><th>Erläuterung</th></tr>
<tr><td rowspan="4">Problemanalyse</td><td colspan="2">Technische Analyse (C4-Modellierung)</td><td>Anhand der C4-Modellierung [2] wird die aktuelle Implementierung erfasst und dokumentiert.</td></tr>
<tr><td colspan="2">Technische Analyse (CRC-Karten)</td><td>Class-Responsibility-Collaboration-Karten (CRC-Karten) [3] werden eingesetzt, um wichtige Komponenten in der aktuellen Implementierung zu dokumentieren.</td></tr>
<tr><td colspan="2">Nicht funktionale Anforderungen</td><td>Da viele nicht funktionale Anforderungen (NFA) einen Einfluss auf die Architektur haben, werden diese in einer Liste erfasst.</td></tr>
<tr><td colspan="2">Funktionale Anforderungen (Use Case Diagramm / Event Storming)</td><td>Event Storming [4] ist eine Workshop-Technik, die von der Analyse eines Geschäftsprozesses in mehreren Schritten zu architekturrelevanten Hinweisen führt, wie zum Beispiel Bounded Context.
Die entstandenen Anforderungen werden mit Use Case Diagrammen dokumentiert.</td></tr>
<tr><td rowspan="4">Lösungsfindung</td><td colspan="2">Longlist</td><td>Eine Longlist an möglichen Architekturvarianten und Technologien wird erarbeitet, die eingesetzt werden können um die Anforderungen zu adressieren. In diesem Schritt werden auch Varianten analysiert, die nicht alle NFA erfüllen, um einen Überblick über die technischen Möglichkeiten zu erhalten.</td></tr>
<tr><td colspan="2">Shortlist</td><td>Die Shortlist besteht aus konkreten Architekturvorschlägen, erstellt mithilfe den Architekturvarianten aus der Longlist. Diese Vorschläge berücksichtigen die Anforderungen, die an die Applikation gestellt werden. Auch werden hier mögliche Aufteilungen der Microservices aufgezeigt.</td></tr>
<tr><td colspan="2">Vergleich der Architekturvorschläge</td><td>Die Architekturvorschläge aus der Shortlist werden anhand der NFA, des Aufwandes und der Best Practices der Community verglichen.
Als Zusammenfassung der Best Practices von Microservices werden die sieben Microservice Tenets [5] eingesetzt.</td></tr>
<tr><td colspan="2">Architektur-Entscheid</td><td>Auf Grundlage des Vergleichs der Architekturvorschläge wird ein Architekturvorschlag ausgewählt.</td></tr>
<tr><td rowspan="6">Umsetzung</td><td colspan="2">Rahmenbedingungen schaffen (z. B. Authentication)</td><td>Die ausgewählte Architektur verlangt gewisse Rahmenbedingungen. In diesem Refactoring ist zum Beispiel eine stateless Authentication vorausgesetzt, die es den Microservices ermöglicht, die Identität der Anfragesteller unabhängig zu überprüfen.</td></tr>
<tr><td colspan="2">API-Gateway erstellen</td><td>Als Vorbereitung, um die Anfragen später an die zuständigen Microservices weiterzuleiten, wird ein API-Gateway erstellt. Dies hat den Vorteil, dass das Vorschalten von API-Gateways nur einmal eingeführt werden muss.</td></tr>
<tr><td rowspan="4">Iterative Vorgehensweise</td><td>Monolith duplizieren</td><td>Um einen Microservice zu erstellen, wird ein Duplikat des Monolithen erstellt. Der neu erstellte Microservice wird an dieselbe Datenbank wie der Monolith angebunden.</td></tr>
<tr><td>Microservice herauslösen</td><td>Die Funktionalität des neuen Microservices wird auf den vorgesehenen Umfang reduziert und beim Monolithen entfernt.</td></tr>
<tr><td>API-Gateway konfigurieren</td><td>Der API-Gateway wird so konfiguriert, dass die zuständigen Anfragen auf den neuen Microservice umgeleitet werden.</td></tr>
<tr><td>Datenbank trennen</td><td>Es wird eine neue Datenbank für den Microservice angelegt, die betroffenen Daten von der Datenbank des Monolithen portiert und die nicht mehr benötigten Daten auf dem Monolithen gelöscht.</td></tr>
</table>

Tabelle 1 Vorgehen cloud-native Refactoring

5 Problemanalyse

5.1 Motivation

Eine traditionelle Webapplikation besteht häufig aus einem Webserver und einer Datenbank. Typischerweise wächst ein solches System mit der Zeit, da neue Funktionalität hinzugefügt wird. Mit der zusätzlichen Funktionalität vergrössert sich auch die Komplexität und der Verantwortungsbereich der Applikation. Aus dieser Komplexität resultieren teilweise schwer nachvollziehbare Fehler, die sich durch die starke Kopplung auf die gesamte Applikation auswirken können. Ein solcher Fehler ist in der Ausgangslage beschrieben. Microservice-Architekturen, auch cloud-native Architekturen genannt, sind ein Trend, bei dem versucht wird, die Zuständigkeiten einer monolithischen Applikation in mehrere kleine Applikationen, sogenannte Microservices, aufzuteilen. Eine solche Architektur hat das Potential die Auswirkung des beschriebenen Problems zu verkleinern, indem nur ein Teil der Applikation betroffen ist und somit ausfällt.

In dieser Arbeit sollen solche Microservice-Architekturen evaluiert werden, die eine bessere Entkopplung und somit höhere Verfügbarkeit versprechen. Es wird kein cloud-native Refactoring um jeden Preis angestrebt, sondern es soll differenziert auf die Vor- und Nachteile eines solchen Umbaus eingegangen werden. Um dies zu erreichen, wird eine methodische Vorgehensweise entwickelt und erprobt, welche auch auf andere Branchen mit ähnlichen Problemstellungen angewendet werden kann.

5.2 Aktuelle Architektur als C4-Modell

Die vorgegebene Applikation wurde zuerst aufgesetzt, in Betrieb genommen und danach getestet. Die Tests generierten die API-Dokumentation, welche die Schnittstellen des Monolithen aufzeigte. Diese Tätigkeiten führten zu einem ersten Überblick über die implementierte Funktionalität der Applikation.

Weiterführend wurde die aktuelle Architektur top-down analysiert. Um dies zu bewerkstelligen, wurde ein C4-Modell [2] mithilfe des Tools Structurizr [6] erstellt.

Die C4-Modellierung kann verwendet werden, um die Architektur von Software-Systemen zu beschreiben und aufzuzeigen, wie diese untereinander kommunizieren. Aus der Modellierung resultieren Diagramme für die Abstraktionsstufen Context, Container, Component und optional Source Code. In diesem Bericht werden die ersten drei Abstraktionsstufen dokumentiert, da der Source Code Eigentum der healthinal GmbH ist.

Damit die Diagramme die vorgegebene Applikation möglichst genau abbilden, wurden die Namen der Komponenten des Source Codes übernommen.

5.2.1 Stufe 1: System Context Diagram

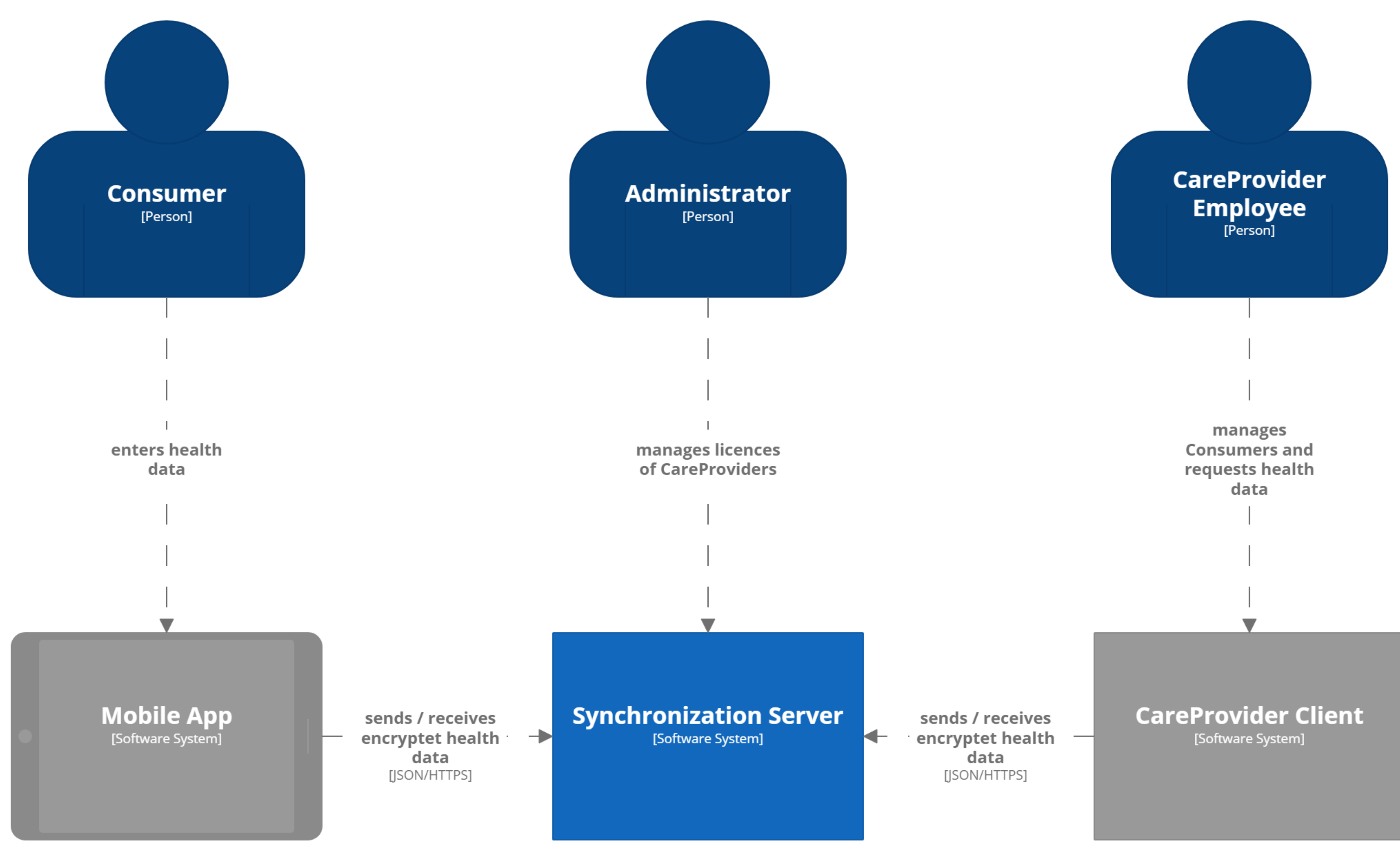


Abbildung 4 System Context Diagram (structurizr) - Synchronization Server

Das *System Context Diagram* zeigt die Systeme, Rollen und deren Interaktionen. Es gibt in dieser Architektur drei Software-Systeme und deren Rollen (Person):

5.2.1.1 *Synchronization Server*

Der *Synchronization Server* ist das Herzstück der Applikation und soll mit dem cloud-native Refactoring in eine Microservice-Architektur umgebaut werden. Dieses Software-System wird von den anderen beiden Systemen, der *Mobile App* und dem *CareProvider Client* verwendet. *Administrators* haben in diesem System die Möglichkeit *CareProvider* zu verwalten.

5.2.1.2 *CareProvider Client*

Der *CareProvider Client* ist ein eigenes Software-System, das bei einem Gesundheitsdienstleister, wie zum Beispiel einem Spital, eingesetzt wird. Die *CareProvider Employee*, die Angestellten von diesem Gesundheitsdienstleister, können über diesen *CareProvider Client* dem *Synchronization Server* neue *Consumer* (Patienten) und deren Geräte hinzufügen. Ebenfalls können die *CareProvider Employee* Daten, die der *Consumer* zum *Synchronization Server* gesendet hat, vom *Synchronization Server* anfordern. Der *CareProvider Client* besteht aus einer Web Applikation (Angular [7]), einem Web Server (Spring Boot) und einer Datenbank (PostgreSQL [8]).

5.2.1.3 *Mobile App*

Die *Mobile App* wird vom *Consumer* (Patient) verwendet, um verschlüsselte Daten, zum Beispiel den Blutdruck, an den *Synchronization Server* zu senden, damit der *CareProvider Client* diese dann beim *Synchronization Server* abholen kann.

5.2.2 Stufe 2: Container Diagram

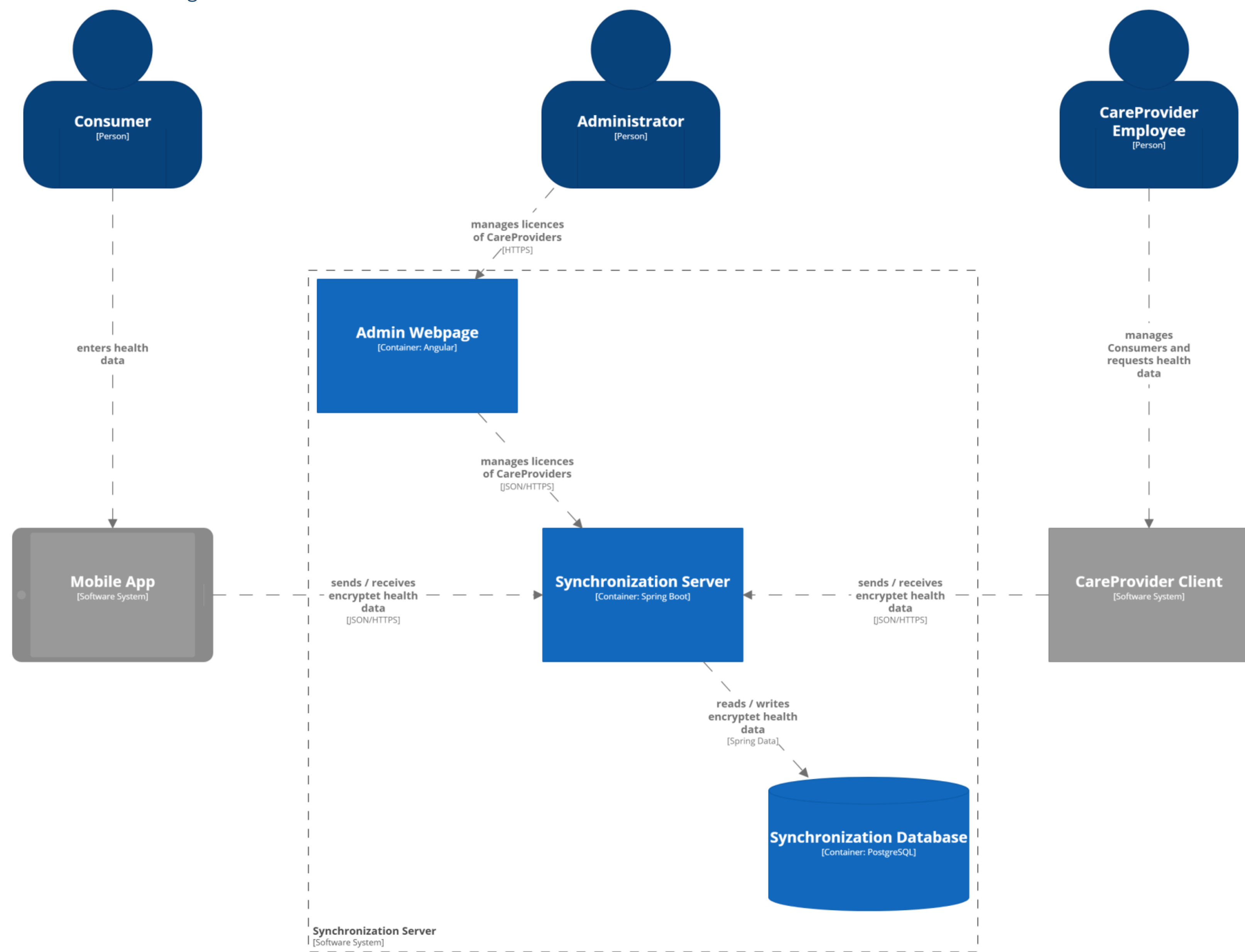


Abbildung 5 Container Diagram (structurizr) - Synchronization Server

Das *Container Diagram* beschreibt die Architektur des *Synchronization Servers* näher, indem diese genauer aufgezeigt wird. Der *Synchronization Server* besteht aus folgenden Containern:

5.2.2.1 *Admin Webpage*

Die *Admin Webpage* ist eine Angular Webseite, mithilfe derer der *Administrator* mit dem *Synchronization Server* kommunizieren kann.

5.2.2.2 *Synchronization Server*

Der *Synchronization Server* ist eine Spring Boot Applikation, die eine API für die anderen Systeme bereitstellt. Seine Daten sind in der *Synchronization Database* persistiert.

<table>
<tr><th colspan="2">*Component:* Synchronization Server</th></tr>
<tr><td>*Responsibilities:*
• Erstellt und verwaltet *CareProvider* und deren Lizenzen
• Benachrichtigt die *Mobile App* falls ein neuer *DataRequest* vom *CareProvider* für ihn vorhanden ist
• Gibt dem *CareProvider Client* auf Anfrage eine Liste von noch nicht synchronisierten *DataRequests* und deren verschlüsselten Gesundheitsdaten zurück
• Stellt sicher, dass die persistierten *DataRequests* von der *Mobile App* an den *CareProvider Client* geliefert werden</td><td>*Collaborators (Interfaces to/from):*
• *Admin Webpage (from)*
• *CareProvider Client (from)*
• *Mobile App (from)*
• *Synchronization Database (to)*</td></tr>
<tr><td colspan="2">*Known uses (implementations):*
• *Synchronization Server (Software System)*</td></tr>
</table>

Tabelle 2 Class-Responsibility-Collaboration-Karte (CRC-Karte) - Synchronization Server

5.2.2.3 *Synchronization Database*

Die *Synchronization Database* ist eine PostgreSQL Datenbank, die Daten für den *Synchronization Server* persistiert.

5.2.3 Stufe 3: Component Diagram

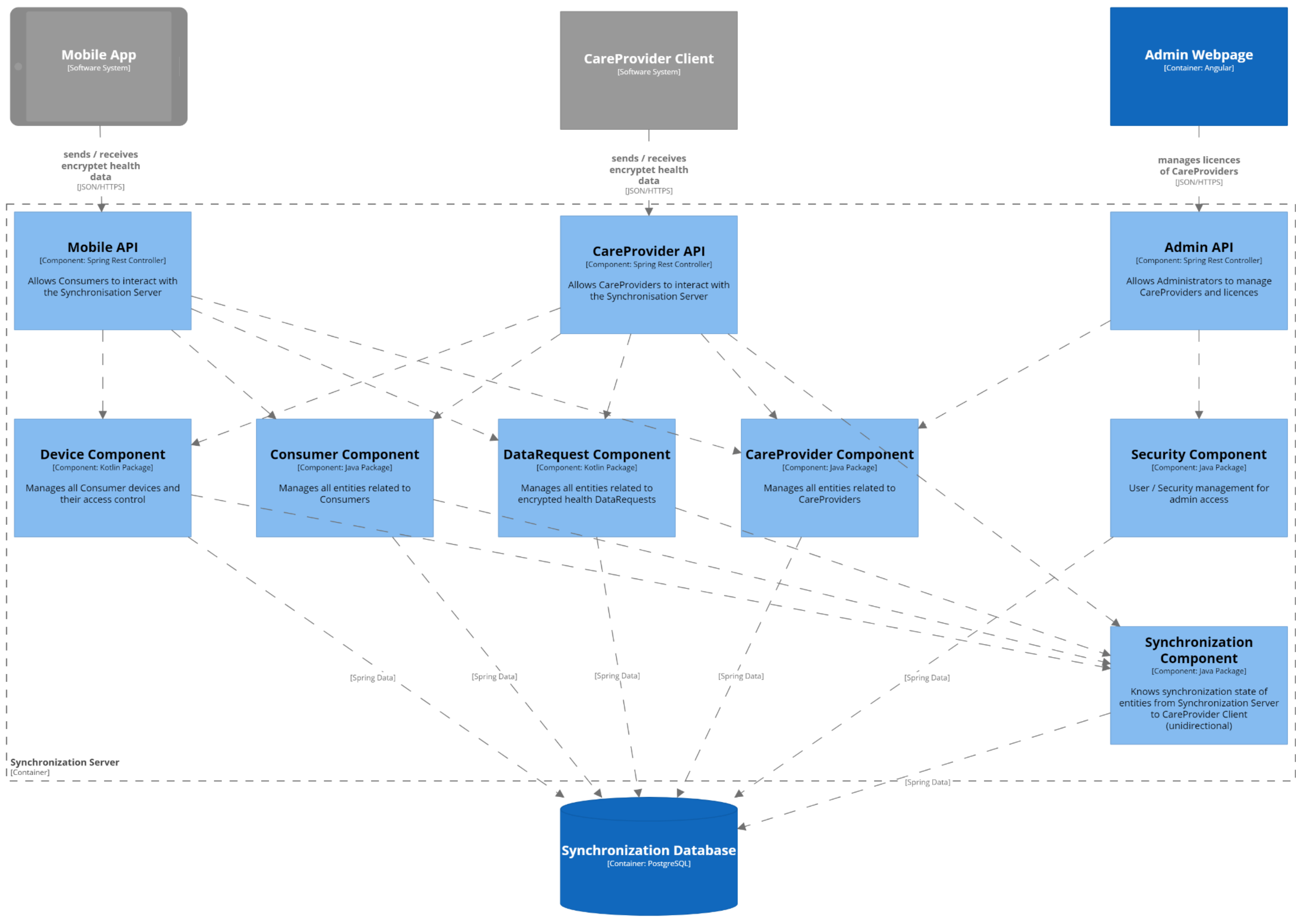


Abbildung 6 Component Diagram (structurizr) - Synchronization Server

Das *Component Diagram* des *Synchronization Servers* besteht aus folgenden *Components*:

5.2.3.1 *Mobile API*

Die *Mobile API* ermöglicht dem *Consumer* über die *Mobile App* mit dem *Synchronization Server* zu kommunizieren.

5.2.3.2 *CareProvider API*

Die *CareProvider API* ermöglicht dem *CareProvider Employee* über den *CareProvider Client* mit dem *Synchronization Server* zu kommunizieren.

5.2.3.3 *Admin API*

Die *Admin API* ermöglicht dem *Administrator* über die *Admin Webpage* mit dem *Synchronization Server* zu kommunizieren, um *CareProvider* und deren Lizenzen zu verwalten.

5.2.3.4 *Device Component*

Die *Device Component* verwaltet alle *Devices* (Geräte) der *Consumer* (Patienten) mithilfe der *Synchronization Database*.

5.2.3.5 *Consumer Component*

Die *Consumer Component* verwaltet alle Entitäten im Zusammenhang mit *Consumers* mithilfe der *Synchronization Database*. Der *Consumer* enthält die Daten des Patienten.

5.2.3.6 *DataRequest Component*

Die *DataRequest Component* verwaltet alle verschlüsselten Entitäten im Zusammenhang mit *DataRequests* mithilfe der *Synchronization Database*. Ein *DataRequest* ist eine Anfrage des *CareProviders* für Gesundheitsdaten an einen *Consumer*, beispielsweise den Blutdruck. Diese Blutdruckwerte werden auf einen *DataRequest* hin vom *Consumer* erfasst, verschlüsselt übertragen und im *DataRequest* gespeichert.

<table>
<tr><th colspan="2">Component: DataRequest Component</th></tr>
<tr><td>Responsibilities:
• Persistiert DataRequests und deren verschlüsselten Gesundheitsdaten.
• Weist die erfassten und verschlüsselten Gesundheitsdaten des Consumers dem DataRequest zu</td><td>Collaborators (Interfaces to/from):
• CareProvider API (from)
• Mobile API (from)
• Synchronization Component (to)
• Synchronization Database (to)</td></tr>
<tr><td colspan="2">Known uses (implementations):
• Synchronization Server</td></tr>
</table>

Tabelle 3 CRC-Karte - DataRequest Component

5.2.3.7 *CareProvider Component*

Die *CareProvider Component* verwaltet alle Entitäten im Zusammenhang mit *CareProviders* mithilfe der *Synchronization Database*. *CareProviders* enthalten die Gesundheitsdienstleister und deren Lizenzen. Sie Authentifiziert ausserdem den *CareProvider* über die Lizenz.

Component: **CareProvider Component**	
Responsibilities: • Verwaltet die Entitäten der *CareProviders* • Erstellt und aktualisiert *CareProviders* • Generiert die Lizenz für die *CareProviders* • Authentifiziert die *CareProvider*	*Collaborators (Interfaces to/from):* • *Admin API (from)* • *CareProvider API (from)* • *Mobile API (from)* • *Synchronization Database (to)*
Known uses (implementations): • *Synchronization Server*	

Tabelle 4 CRC-Karte - CareProvider Component

5.2.3.8 *Security Component*

Die *Security Component* ist für das Verwalten der *Administrator* User und die Zugriffsberechtigung für die Verwaltung des *Synchronization Servers* mithilfe der *Synchronization Database* zuständig.

5.2.3.9 *Synchronization Component*

Die *Synchronization Component* verwaltet mithilfe der *Synchronization Database* den Status der Entitäten vom *Synchronization Server* zum *CareProvider Client* (unidirektional). Durch das Persistieren des Status (ausgeliefert, nicht ausgeliefert) der zu übertragenden Daten entscheidet der *Synchronization Server,* welche Daten der *CareProvider Client* noch aktualisieren muss.

Component: **Synchronization Component**	
Responsibilities: • Synchronisiert die Daten vom *Synchronization Server* zum *CareProvider* • Persistiert den Status der Daten (ausgeliefert, nicht ausgeliefert) • Liefert auf Anfrage unsynchronisierte Daten aus • Aktualisiert den Status der Daten bei bestätigter Auslieferung	*Collaborators (Interfaces to/from):* • *CareProvider API (from)* • *Device Component (from)* • *Consumer Component (from)* • *DataRequest Component (from)* • *Synchronization Database (to)*
Known uses (implementations): • *Synchronization Server*	

Tabelle 5 CRC-Karte - Synchronization Component

5.3 Nicht funktionale Anforderungen

Die nachfolgende Tabelle listet die nicht funktionalen Anforderungen an das Software-System auf. Die Liste wurde im Rahmen dieser Arbeit aus den bestehenden Anforderungen und der Aufgabenstellung erarbeitet. Sie dient als Grundlage für die Architektur-Entscheidungen, die im weiteren Verlauf der Arbeit getroffen werden.

Qualitätsattribut	Qualitätsmerkmal
Funktionalität	1. Ein QR-Code muss auch ausgedruckt und erfolgreich gescannt werden können, wenn im Spital keine Internet-Verbindung vorhanden ist. 2. Die Komponente beim Gesundheitsdienstleister kann auch benutzt werden, wenn keine Internet-Verbindung bei diesem vorhanden ist. Die getätigten Aktionen werden synchronisiert, sobald wieder eine Internet-Verbindung vorhanden ist. 3. Der Patient wird per Push-Benachrichtigung über neue Daten informiert.
Zuverlässigkeit	4. Der Patient kann in 99.5 % der Zeit Daten an den Server senden und lesen. 5. Das System soll in den Spitzenzeiten von bis zu 10'000 Anfragen in der Sekunde in der Grösse von 50 KB 99.5 % dieser Anfragen abarbeiten können. 6. Die Komponente beim Gesundheitsdienstleister und der App des Patienten sollen sich nicht gegenseitig blockieren können.
Benutzbarkeit	7. Die Applikation kann bei einem Gesundheitsdienstleister bereitgestellt und betrieben werden, ohne dass eine Konfiguration an der Firewall vorgenommen werden muss.
Effizienz	8. Überall, wo ein Mensch mit der Applikation interagiert, darf die Reaktionszeit in 99.5 % der Fälle nicht grösser als 2 Sekunden sein. 9. Überall, wo eine Maschine mit einer Maschine kommuniziert, darf die Reaktionszeit in 99.5 % der Fälle nicht grösser als 4 Sekunden sein. 10. Das Senden einer Anfrage von einem Patienten zu einem Gesundheitsdienstleister oder umgekehrt, darf in 99.5 % der Fälle nicht länger als 20 Sekunden dauern, wenn beide Parteien eine aktive Internet-Verbindung haben.
Wartbarkeit	11. Es sollen nicht mehr als 4 besser 3 verschiedene Frameworks/Technologien (z. B. Spring, Angular, PostgreSQL) in der ganzen Applikation eingesetzt werden, um ein möglichst grosses Wissen auf diesen Technologien zu haben. 12. Die Applikation weist eine Test-Abdeckung von mindestens 75 % vor, um neue Änderungen schnell einführen zu können.
Übertragbarkeit	13. Die Dokumentation der API der Web-Server soll bis spätestens zwei Wochen nach dem Commit in den Masterbranch und erfolgreichem Testdurchlauf konsistent mit der effektiven Implementierung sein.
Sicherheit	14. Die ausgetauschten Daten zwischen einem Patienten und dem Gesundheitsdienstleister können nur auf der Komponente beim Gesundheitsdienstleister und auf dem Smartphone des Patienten gelesen werden. Die Daten sind mit einer kryptographischen Stärke von mindestens 128 Bit Ende-zu-Ende verschlüsselt.

Tabelle 6 Nicht funktionale Anforderungen

5.4 Domänen-Analyse mithilfe von Event Storming

Event Storming ist eine Workshop-Methode, die von Alberto Brandolini im Kontext von Domain-Driven Design (DDD) entwickelt wurde [4]. Event Storming kann eingesetzt werden, um einen Geschäftsprozess zu modellieren und funktionale Anforderungen zu erfassen. Diese Workshop-Methode fokussiert sich auf die Events, die in einer Domäne auftreten.

5.4.1 Erwartetes Ergebnis

Das erwartete Ergebnis ist ein Einblick in die Problemstellung und die Identifikation von unabhängigen Domänen und Konzepten. Da es sich um eine DDD-Analyse-Methode handelt, ist eine weitere Erwartung, dass die gewonnenen Erkenntnisse in das Finden von Architekturvorschlägen einfliessen werden.

5.4.2 Vorgehen

Als Leitfaden für das Vorgehen der Analyse wurde die Anleitung zu Event Storming aus dem Buch Domain-Driven Design kompakt von Vernon [9] verwendet. Die folgenden Kapitel dokumentieren das effektiv stattgefundene Vorgehen, jedoch keine ausführliche Anleitung zu Event Storming.

Aus Gründen der Lesbarkeit wurde im Text auf den Klebezettel die männliche Form gewählt, nichtsdestoweniger beziehen sich die Angaben auf Angehörige beider Geschlechter.

5.4.3 Notation der Klebezettel

Die Klebezettel haben folgende Bedeutung:

Klebezettel-Farbe	Bedeutung
Pink	Das **Domain Event** ist ein Ereignis, das im Geschäftsprozess vorkommt
Grün	Der **Command** ist der Auslöser, der zu einem Domain Event führt
Gelb (kleine Zettel)	Die **Rolle**, die ein Command auslöst
Orange	Ein **Prozess** der einen komplexeren Ablauf abstrahiert
Gelb (grosse Zettel)	Das **Entity** oder **Aggregate** das zum Command-Event-Paar gehört
Weiss	Name der **Domäne**

Tabelle 7 Event Storming Klebezettel Notation

5.4.4 Schritt 1: Domain Events erfassen

Im ersten Schritt wurden die Events erfasst, die sich in der Applikation ereignen. Als Use Case wurde eine durch den Arzt oder die Ärztin verordnete Blutdruck-Erfassung genommen.

5.4.4.1 *Resultat*

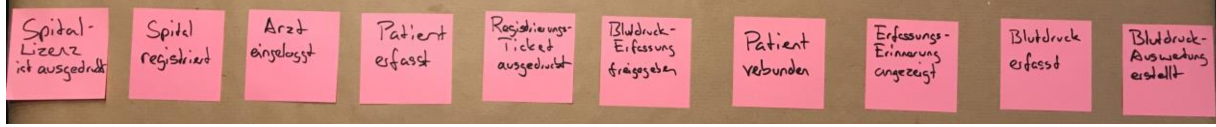


Abbildung 7 Event Storming Schritt 1 - Erfassung der Events

In der Ausführung stellte sich die Frage, ob die Events *Spital-Lizenz ist ausgedruckt* und *Spital ist registriert* zu technisch sind und hier nicht reingehören. Vernon empfiehlt komplexe Prozesse, die zwingend zur Hauptdomäne gehören, als einzelnen Prozess zu modelieren:

> *« Höchstwahrscheinlich ist ein Benutzerregistrierungsprozess eine Notwendigkeit, aber vermutlich nicht als ein Kern-Feature Ihrer Anwendung. Modellieren Sie den Registrierungsprozess als ein einzelnes grobgranulares Ereignis UserRegistered, und machen Sie weiter. »* [9, S. 115]

Bei der Spital-Registrierung scheint es sich um ein ähnliches Problem zu handeln.

5.4.5 Schritt 2: Commands erfassen

Den erstellten Events werden nun die Commands zugeordnet, die zum jeweiligen Event führen. Vernon ermutigt zum Erstellen von neuen Domain-Events, falls welche gefunden werden. Den Commands wurde die jeweilige Rolle zugeordnet, sofern eine solche existiert.

5.4.5.1 *Resultat*

Abbildung 8 Event Storming Schritt 2 - Erfassung der Commands Teil 1

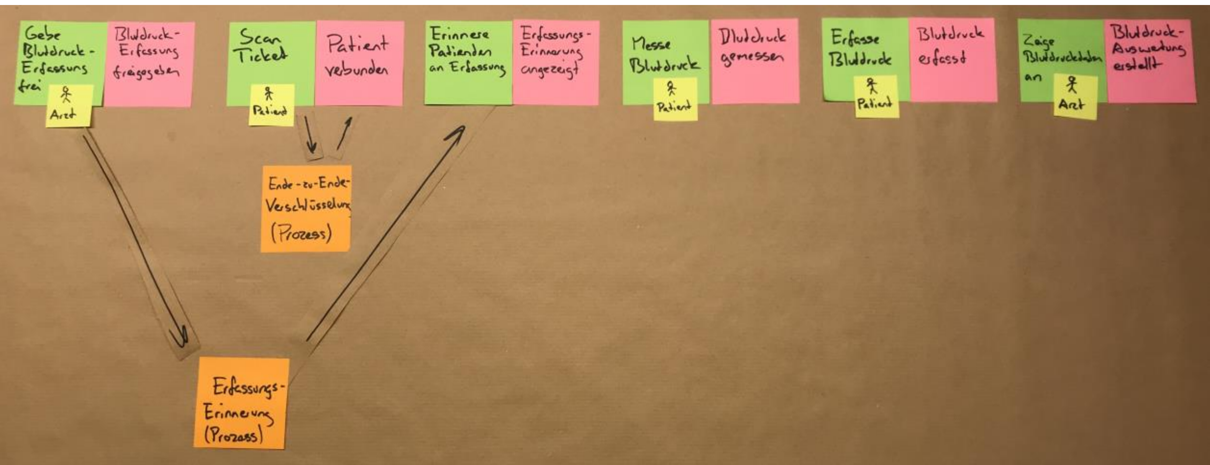


Abbildung 9 Event Storming Schritt 2 - Erfassung der Commands Teil 2

Während der Erfassung der Commands wurde der Event *Blutdruck gemessen* gefunden. Interessant ist, dass dieser erst während dem Nachdenken über den auslösenden Command gefunden wurde. Es wurde festgestellt, dass weitere Schritte nötig sind, bis es zum Event *Blutdruck erfasst* kommt. Das *Erfasse Blutdruck* Beispiel zeigt auch, warum es sinnvoll ist die Rolle zu erwähnen, denn es ist eine wichtige Information, dass der Patient oder die Patientin den Blutdruck selber misst und erfasst.

Interessanterweise kann nicht immer direkt aus einem Event auf den Command geschlossen werden. Das wird am Beispiel *Scan Ticket -> Patient verbunden* sichtbar. Das Suchen nach dem Command hat hier den Prozess *Ende-zu-Ende-Verschlüsselung aufbauen* sichtbar gemacht.

Der Command *Erinnere Patienten an Erfassung* konnte keiner Rolle zugeordnet werden. Das hat den Prozess *Erfassungs-Erinnerung* aufgedeckt, der ursprünglich vom Command *Gebe Blutdruck-Erfassung frei* initiiert wird.

5.4.6 Schritt 3: Entity / Aggregates erfassen

In diesem Schritt wurde jedem Command-Event-Paar die zugehörige Entität zugeordnet. Auch in diesem Schritt ermutigt Vernon neue Events zu erstellen, falls diese gefunden werden.

5.4.6.1 *Resultat*

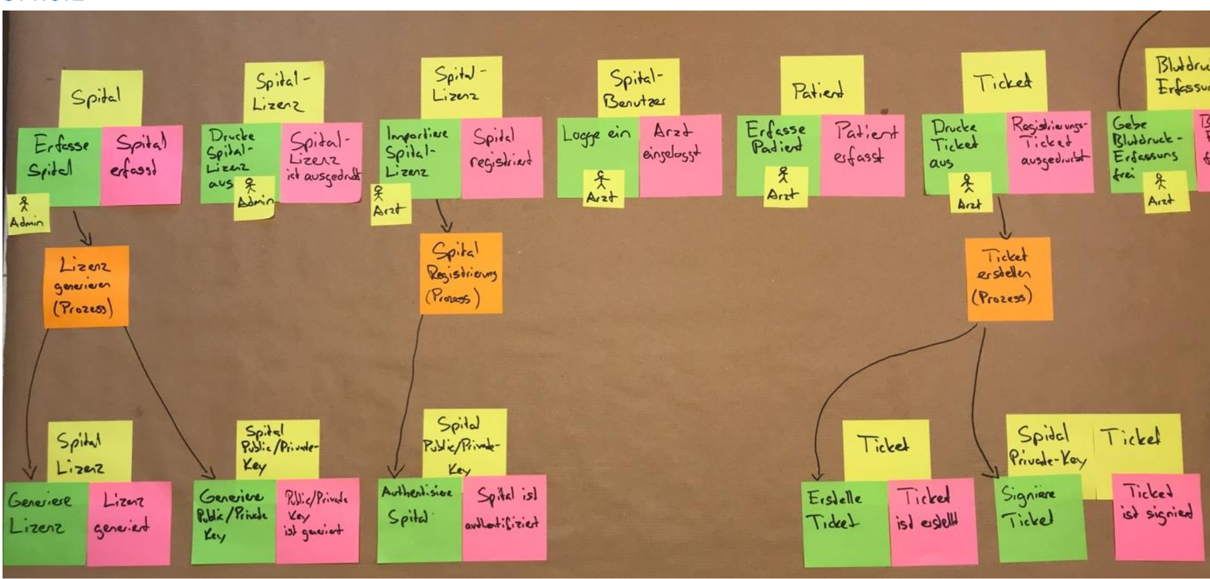


Abbildung 10 Event Storming Schritt 3 - Erfassung der Entitäten Teil 1

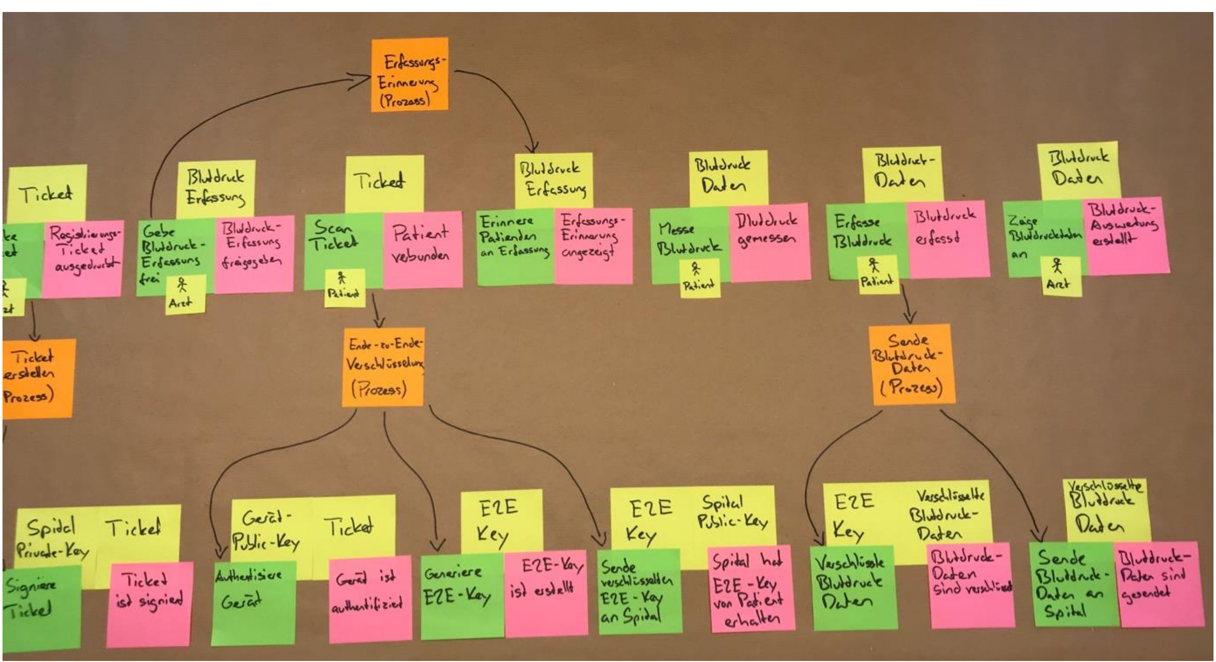


Abbildung 11 Event Storming Schritt 3 - Erfassung der Entitäten Teil 2

Beim Durchspielen der Events mit den Entitäten wurden neue Events entdeckt. Insbesondere bei den Prozessen wurde ersichtlich, dass weitere Entitäten existieren. Ersichtlich wird dies zum Beispiel bei dem Prozess *Ende-zu-Ende-Verschlüsselung*. Hier stellte sich die Frage, ob diese technischen Details überhaupt modelliert werden sollen. Vernon hat hier folgenden Ratschlag:

« Nach meiner Erfahrung tendieren Teams dazu, innerhalb derselben Sessions zwischen Big-Picture- und Designebene zu wechseln. Am Ende werden Sie hoffentlich auf der Designebene landen, denn hier werden Sie die entscheidenden Details erfahren. » [9, S. 118].

Auf dieser Grundlage wurde entschieden, dass die technischen Details modelliert werden sollen. Ein weiterer Grund dafür war, dass es sich um ein wichtiges Konzept im Kontext der Applikation handelt und deshalb diskutiert werden sollte.

Abweichend von der Anleitung für Event Storming wurden einem Command-Event-Pair teilweise mehrere Entitäten zugeordnet, da zum Beispiel bei der Verschlüsselung die Blutdruck-Daten sowie der Schlüssel benötigt werden.

5.4.7 Schritt 4: Bounded Context definieren

Die chronologisch aufgelisteten Events wurden in diesem Schritt in Domänen gruppiert. Diese sollen möglichst unabhängig voneinander sein und die verschiedenen Konzepte kapseln, die in der Applikation existieren.

5.4.7.1 *Resultat*

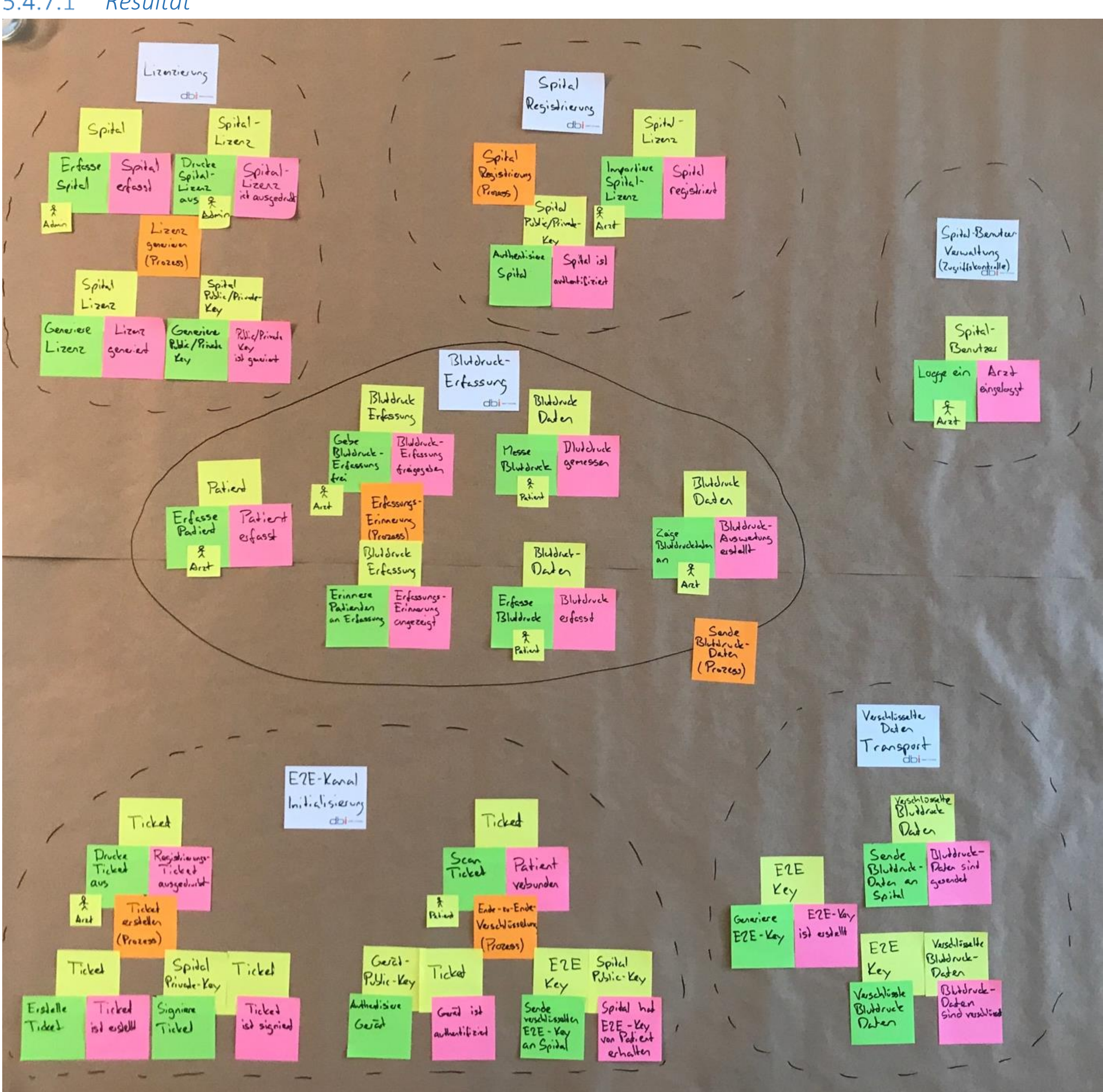


Abbildung 12 Event Storming Schritt 4 - Erstellen der Bounded Context

Das Erstellen der Bounded Context hat am meisten Zeit in Anspruch genommen, da ausführliche Diskussionen über die Grenzen der Domänen geführt wurden. Ein Diskussionspunkt war zum Beispiel, ob die Domäne *Spital-Benutzer-Verwaltung* und *Spital-Registrierung* zusammengehören

oder nicht. Schlussendlich wurde aber festgestellt, dass das Einloggen eines Arztes oder einer Ärztin in die Applikation nicht viel mit der Registrierung der Spital-Software beim *Synchronization Server* zu tun hat. Diese Erkenntnis führte dazu, dass diese Konzepte getrennt wurden.
Generell war hier der Hinweis von Vernon [9, S. 24] hilfreich, dass Konzepte in einer Domäne gekapselt werden können.

Auch die beiden Domänen *E2E-Kanal-Initialisierung* und *Verschlüsselter-Daten-Transport* wurde anfangs als eine einzige Domäne modelliert. Diese Trennung erwies sich aber trotzdem als sinnvoll, denn die Domäne *E2E-Kanal-Initialisierung* wird nur benutzt, um die Ende-zu-Ende-Verschlüsselung im *Verschlüsselten-Daten-Transport* Kontext zu ermöglichen. Es gibt also hier das Konzept Schlüssel-Austausch anhand eines Tickets und das Konzept verschlüsselter Datentransport.

5.4.8 Resultat des Event Stormings / Entstandene Domänen

Die folgenden Ausführungen erläutern die während des Workshops gefundenen Domänen und deren Zuständigkeiten.

5.4.8.1 *Blutdruck-Erfassung (Hauptdomäne)*

Am Beispiel der Blutdruck-Erfassung wird die Hauptaufgabe der Applikation sichtbar. Der Zweck der Applikation ist das Übermitteln von medizinischen Daten. Diese Domäne beinhaltet nur die Komponenten, die diesem Zweck dienen. Diese Domäne kann auch in der nicht digitalen Welt beobachtet werden: Ein Arzt oder eine Ärztin verordnet eine Blutdruck-Erfassung, der Patient oder die Patientin führt diese aus und im Anschluss wertet der Arzt oder die Ärztin diese aus.

5.4.8.2 *E2E-Kanal-Initialisierung (unterstützende Domäne)*

Diese Domäne beschäftigt sich mit der Frage, wie eine Ende-zu-Ende verschlüsselte Verbindung zwischen der Gesundheitsfachperson und dem Patienten oder der Patientin hergestellt werden kann. Sie beinhaltet Elemente wie ein Ticket für den Aufbau der Verbindung und asymmetrische Schlüsselpaare, die die Vertraulichkeit zwischen den zwei Parteien garantieren. Die Hauptaufgabe dieser unterstützenden Domäne ist das Erstellen und Verteilen eines symmetrischen Verschlüsselungs-Schlüssels für den vertraulichen Datentransport.

5.4.8.3 *Verschlüsselter Datentransport (unterstützende Domäne)*

Sobald ein symmetrischer Schlüssel zur Verschlüsselung der bidirektionalen Kommunikation vorliegt, kann diese Domäne ihre Funktion erfüllen. Sie ist dafür zuständig, dass die kommunizierten Daten verschlüsselt vom Patientengut zum Gesundheitsdienstleister und umgekehrt synchronisiert werden. Diese Domäne ist auch dafür verantwortlich, das zeitlich entkoppelte Senden und Empfangen der Daten zu koordinieren, was zur Erfüllung der nicht funktionalen Anforderung 2 beiträgt. Dazu gehören zum Beispiel auch Push-Notifikationen, die auf dem Smartphone erscheinen und die App und somit das Patientengut über neue Daten informiert.

5.4.8.4 *Lizenzierung (unterstützende Domäne)*

Diese Domäne beschäftigt sich mit der Frage, ob ein Spital autorisiert ist eine Aktion auszuführen. Neben dem finanziellen Aspekt der Lizenzierung ist sie auch dafür verantwortlich, ein Spital eindeutig zu identifizieren und authentifizieren.

5.4.8.5 *Spital-Benutzer-Verwaltung (unterstützende Domäne)*

Die Verantwortung dieser Domäne ist die Authentifizierung des Spital-Personals zur Benutzung der Anwendung.

5.4.9 Vergleich der Bounded Contexts mit aktueller Implementierung

Das *C4 Component Diagram* in Abbildung 6 zeigt, dass die Struktur der Komponenten (Java Packages) im Monolithen anhand der Entitäten, wie zum Beispiel *CareProvider* oder *Device,* gemacht wurde.

Beim Erstellen der Bounded Context im Schritt 4 des Event Stormings wurden die gleichen Entitäten und ihre zugehörigen Events und Commands jeweils im selben Bounded Context modelliert. Im Vergleich zu der bestehenden Implementierung, die nach Entitäten strukturiert ist, kann darum eine tiefere Kopplung und höhere Kohäsion vom cloud-native Refactoring nach Domänen erwartet werden.

5.4.10 Use Cases aus Event Storming

Folgendes Use Case Diagramm veranschaulicht die Interaktionen der Rollen mit dem *Synchronization Server*, die im Event Storming diskutiert wurden.

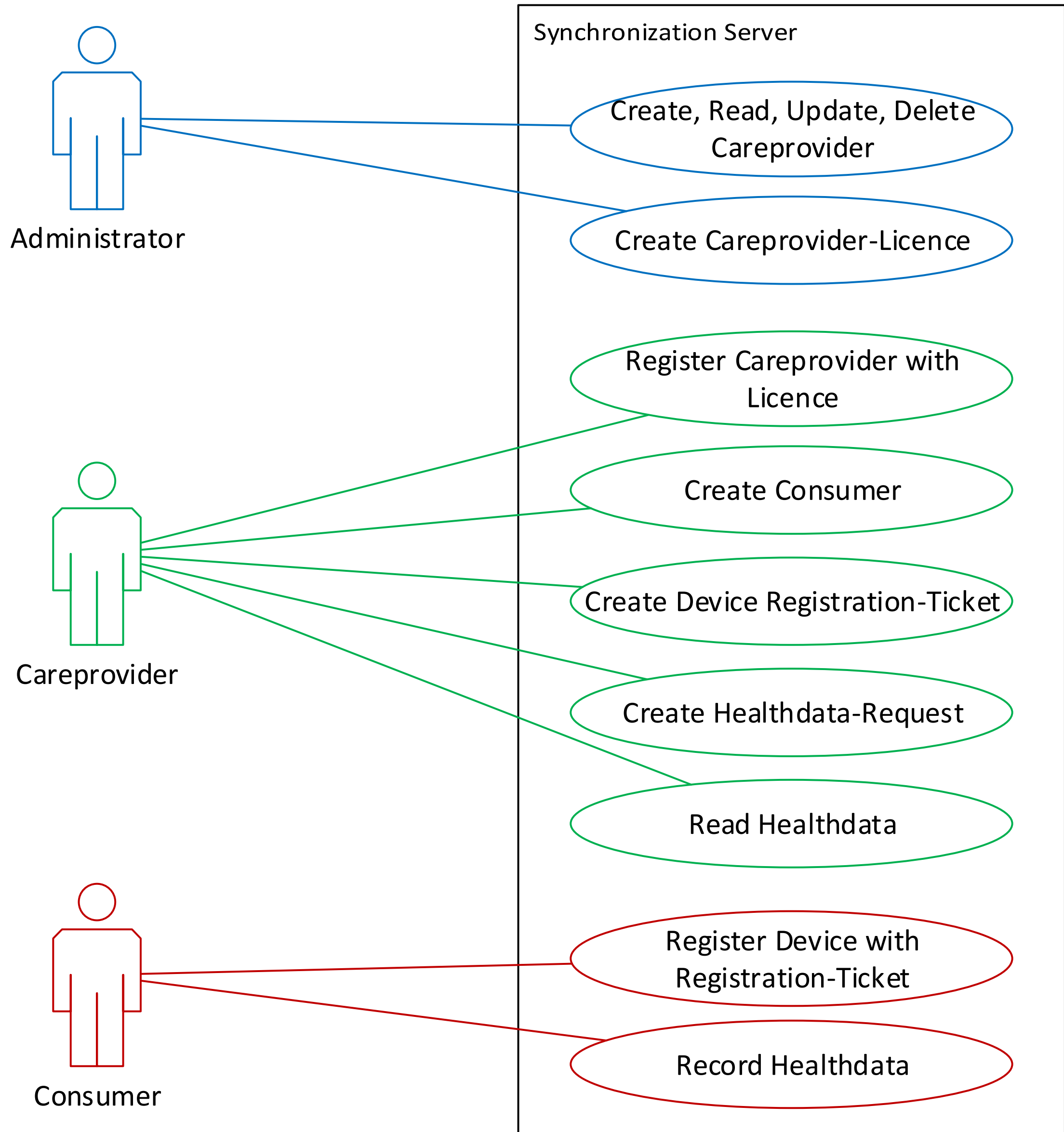


Abbildung 13 Use Case Diagramm funktionale Anforderungen

5.5 Domänen-Aufteilung mithilfe Context Mapper und Service Cutter

Der Context Mapper [10] ist ein Open Source Projekt, das mithilfe der Context Mapper Domain Specific Language (CML) [11], welche auf Domain-Driven Design (DDD) Patterns [12] basiert, eine Domäne in mehrere Bounded Contexts unterteilt. Mit der Integration des Service Cutter [13] lässt sich die CML-Modellierung anhand der Coupling Criterias (CC) [14] mithilfe von zwei Algorithmen, dem Leung [15]- und Girvan-Newman [16]-Algorithmus, in Microservices aufteilen.

In diesem Kapitel wird untersucht, ob ein solches Tool sinnvoll für die Unterteilung des *Synchronization Servers* in Microservices eingesetzt werden kann.

5.5.1 Context Mapper

Mit der CML wurden die Bounded Context aus dem Event Storming für den *Synchronization Server* modelliert. Zusätzlich wurden damit die Use Cases als Interaktionen mit den Entitäten dieser Bounded Context modelliert. Die Use Cases sind aus dem Code und dem Event Storming abgeleitet.

Der Use Case *Spital Registrierung* aus dem Event Storming in Abbildung 10 zeigt beispielsweise, dass auf den Entitäten *CareProviderLicence* und *CareProviderKeyPair* gelesen und der Entität *CareProviderAuthKey* geschrieben wird. Die Entität *CareProviderAuthKey* stellt den Zugriffsschlüssel des Spitals dar und wurde im Event Storming nicht identifiziert, was daran lag, dass dieser nicht weiter von Bedeutung für den modellierten Geschäftsprozess war.

```
UseCase RegisterCareProvider {
	reads "CareProviderLicence.careProviderId", "CareProviderKeyPair.publicKey"
	writes "CareProviderAuthKey.authKey"
}
```

Abbildung 14 Beispiel Use Case RegisterCareProvider, modelliert mit der CML des Context Mapper

Die Attribute *careProviderId*, *publicKey* und *authKey* werden von dem Use Case *RegisterCareProvider* verwendet.

Aus den geschriebenen CML-Dateien konnte mit dem Context Mapper der Input für den Service Cutter generiert werden.

5.5.2 Service Cutter

Die generierten Dateien aus dem Context Mapper können beim Service Cutter importiert werden um diese darzustellen und mithilfe von gewichteten Coupling Criterias (CC) aufzuteilen.

Als Resultat der importierten Dateien vom Context Mapper lieferte der Service Cutter mehrere Services, die Abhängigkeiten untereinander aufzeigen:

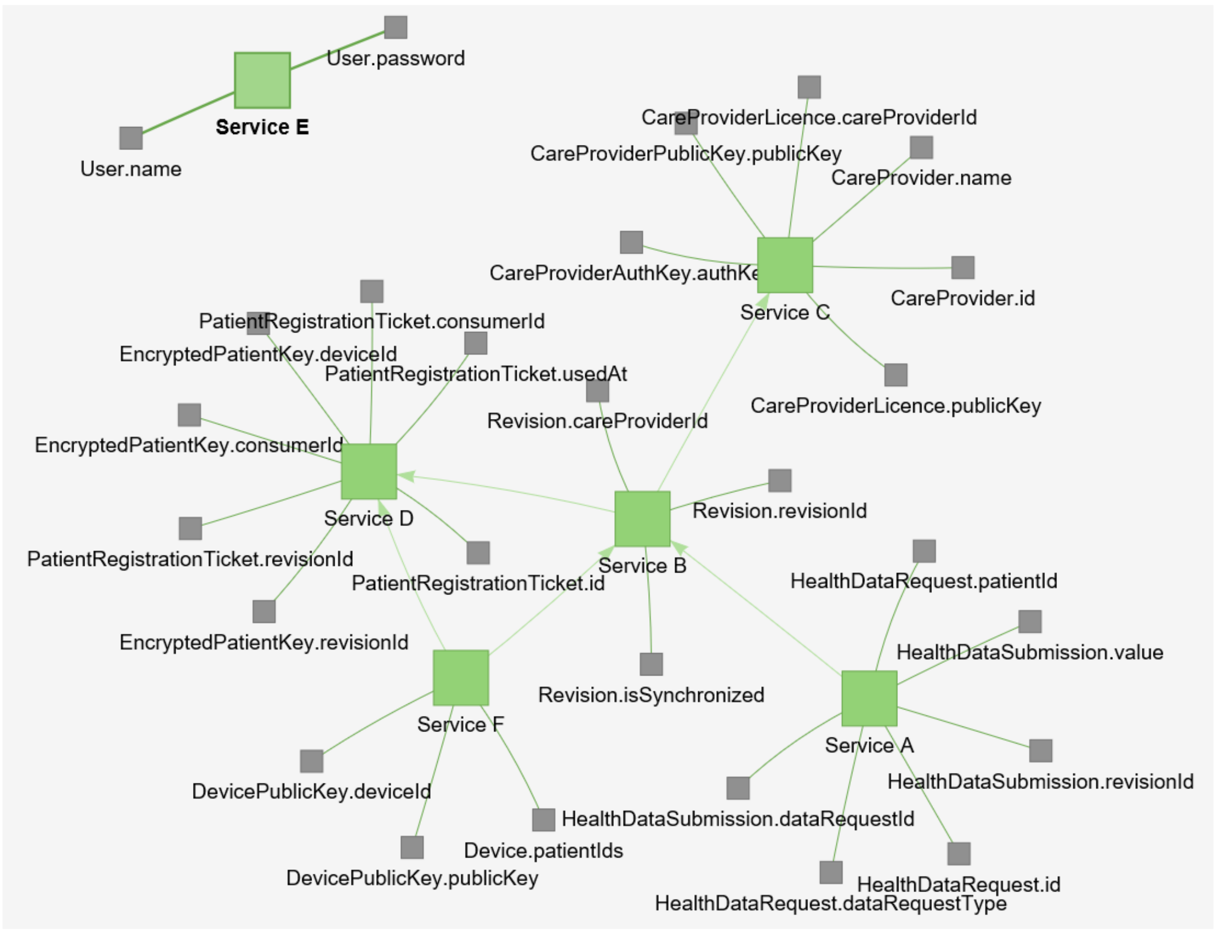


Abbildung 15 Service Cutter Resultat

Mithilfe des Leung Algorithmus konnte der Service Cutter die Services aufteilen, entgegen dem Girvan-Newman Algorithmus, mit welchem mit den erstellten Dateien keine brauchbare Aufteilung erreicht werden konnte.

Das Resultat des Service Cutters sind sechs Services mit Abhängigkeiten (grüne Pfeile) untereinander. Diese Aufteilung war zu erwarten, da mit der CML bereits sechs Bounded Contexte erstellt wurden und somit diese Aufteilung bereits von Hand gemacht wurde. Für eine bessere Unterteilung müssten die CC in der CML hinzugefügt werden, was aber sehr zeitintensiv wäre.

Von Nutzen sind die Pfeile zwischen den Services, die durch die Use Cases entstanden sind und die Abhängigkeiten der Services aufzeigen. Diese Pfeile bedeuten, dass zwischen diesen Services eine Kommunikation stattfinden wird. Dadurch ist ersichtlich, dass zum Service B, welcher für die Revision (Synchronisation) verantwortlich ist, grosse Abhängigkeit besteht.

5.5.3 Resultat

Der Einsatz des Context Mapper und Service Cutter für ein cloud-native Refactoring eines Monolithen ist nicht zu empfehlen, da es sehr aufwändig ist, alle Abhängigkeiten im Code in der CML abzubilden. Wenn die CML-Modellierung nicht vollständig ist, generiert der Service Cutter eine unbrauchbare

oder falsche Microservice-Aufteilung. Falls die CML-Modellierung aus dem Code des Monolithen generiert werden könnte, würde der Einsatz des Service Cutter vermutlich mit angemessenem Aufwand möglich sein.

Das Modellieren der Bounded Context aus dem Event Storming führte dazu, dass der Service Cutter für jeden modellierten Bounded Context ein Microservice vorschlug und somit keine neuen Erkenntnisse zur Aufteilung in Microservices lieferte. In diesem Refactoring konnte der Context Mapper in Kombination mit dem Service Cutter also keine hilfreichen Hinweise für die Microservices liefern.

Von Nutzen war jedoch das Modellieren der Use Cases in der CML, denn dadurch musste der Code sehr genau analysiert werden, was neue Abhängigkeiten aufdeckte, welche im Event Storming nicht ersichtlich waren.

5.6 Ergebnis Problemanalyse

Die Analyse der aktuellen Implementierung mithilfe der C4-Modellierung, dem Event Storming, dem Context Mapper mit Service Cutter und dem Erfassen der funktionalen und nicht funktionalen Anforderungen ergibt folgende Verbesserungspunkte:

- Single Point of Failure *Synchronization Server*:
  Ist der *Synchronization Server* nicht verfügbar, können weder *CareProvider* (Gesundheitsdienstleister) noch *Consumer* (Patienten und Patientinnen) ihre Daten schicken und abfragen. Jegliche Interaktionen zwischen dem *Consumer* und dem *CareProvider* führen über diesen *Synchronization Server* und können durch die *Synchronization Database* potentiell das gesamte System blockieren. Zum Beispiel kann das Registrieren eines neuen Spitals dazu führen, dass die Datenbankabfrage blockiert und somit jegliche weitere Interaktion mit dem System verhindert, was die Erfüllung von der NFA 6 ausschliesst.

- Grosse Abhängigkeiten:
  Das *C4 Component Diagram* des *Synchronization Servers* in Abbildung 6 und das Service Cutter Resultat haben ergeben, dass innerhalb des *Synchronization Servers* grosse Abhängigkeiten zur *Synchronization Component* (Java-Package) bestehen. Diese Abhängigkeiten gilt es mit den Microservices aufzulösen.

- Namensgebung:
  Die Namensgebung ist bei der vorgegebenen Applikation absichtlich möglichst generisch gewählt, damit auch andere Probleme mit dieser Architektur ohne Namensänderung gelöst werden können. Ab den Architekturvarianten werden die Namen jedoch spezifischer und intuitiver gewählt. Als Beispiel ist nicht mehr von *Consumer* die Rede, sondern von *Patienten*.

6 Lösungsfindung

Die Lösungsfindung dient dazu, aus einer Longlist an Architekturvarianten eine Shortlist an Architekturvorschlägen zu erstellen, diese untereinander zu vergleichen und ein Architekturvorschlag für den prototypischen Umbau auszuwählen.

6.1 Architekturvarianten (Longlist)

In diesem Kapitel werden verschiedene Architekturvarianten diskutiert. Es wurde bewusst eine vielfältige Auswahl an Varianten gesucht. Dabei handelt es sich nicht nur um Microservice-Architekturen und auch nicht jede Variante erfüllt alle nicht funktionalen Anforderungen (NFA). Damit wird erreicht, dass auch Lösungen gefunden werden können die auf den ersten Blick nicht viel Sinn machen, aber bei einer weiteren Diskussion trotzdem zu einer brauchbaren Variante erweitert werden könnten.

Folgende Architekturvarianten werden untersucht:

- Monolith (aktuelle Implementierung des *Synchronization Servers*)
- Variante 1: Single-Gateway-Monolith
- Variante 2: Multi-Gateway-Monolith
- Variante 3: Monolith-Per-Careprovider
- Variante 4: Microservice-Direct-Invocation
- Variante 5: Single-Gateway-Multi-Microservice
- Variante 6: Multi-Gateway-Multi-Microservice
- Variante 7: Gateway-Queue-Microservice
- Variante 8: Serverless-Direct-Database

6.1.1 Notation

Die Architekturvorschläge in diesem Kapitel nutzen folgende Notation:

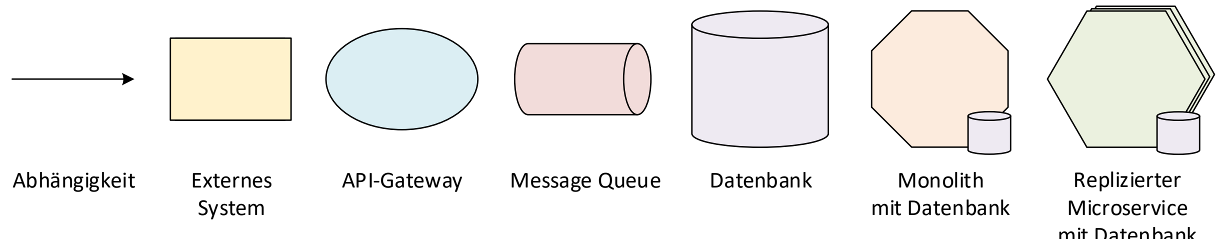


Abbildung 16 Architekturvorschläge Legende

Die kleinen Datenbanksymbole beim Monolithen und beim replizierten Microservice stehen für einen skalierten Datenbankcluster. Die replizierten Microservices greifen jeweils alle auf denselben Datenbankcluster zu.

6.1.2 Monolith (aktuelle Implementierung des *Synchronization Servers*)

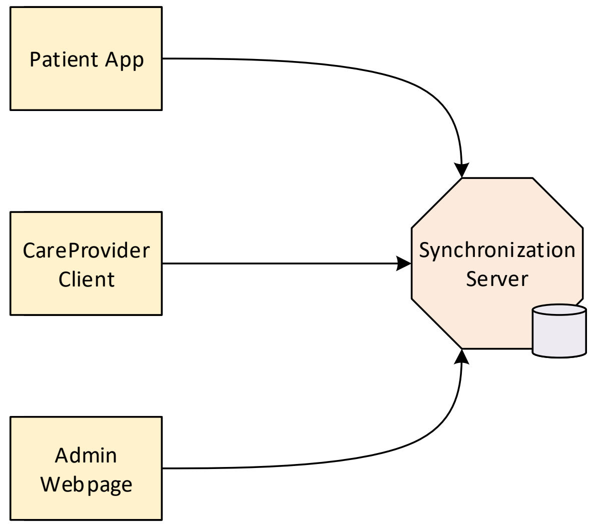


Abbildung 17 Architekturvariante Monolith

Dieses Diagramm repräsentiert die aktuell implementierte Architektur. Sie besteht aus dem monolithischen *Synchronization Server* der von allen drei externen Systemen, der *Admin Webpage*, dem *CareProvider Client* und der *Patient App*, aufgerufen wird.

6.1.3 Variante 1: Single-Gateway-Monolith

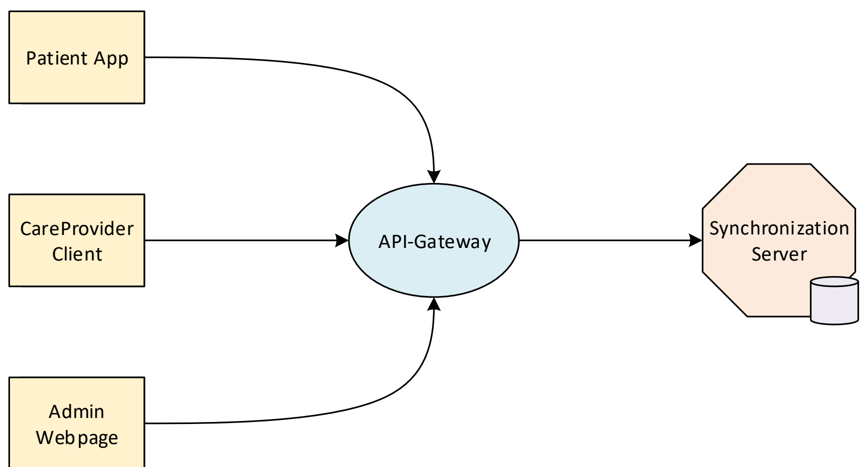


Abbildung 18 Architekturvariante Single-Gateway-Monolith

Ein einzelnes vorgeschaltetes API-Gateway beim Monolithen kann die Aufgabe der Lastenverteilung zu den verschiedenen Instanzen des Monolithen übernehmen. Zusätzlich könnte das API-Gateway Protokollkonvertierungen und spezifische Endpunkte für die verschiedenen Parteien anbieten.

6.1.3.1 *Eignungsuntersuchung*

Ungeeignet: Das Vorschalten eines API-Gateways beim Monolithen verbessert die Fehleranfälligkeit des Monolithen nicht. Dieser ist nach wie vor der Single-Point-of-Failure. Der Ansatz, dass gewisse Routing-, Konvertierungs- oder Authentifizierungs-Logik in ein vorgelagertes API-Gateway ausgelagert werden, hat aber trotzdem seine Berechtigung, da das API-Gateway diese Aufgaben übernehmen und horizontal skaliert werden kann.

6.1.4 Variante 2: Multi-Gateway-Monolith

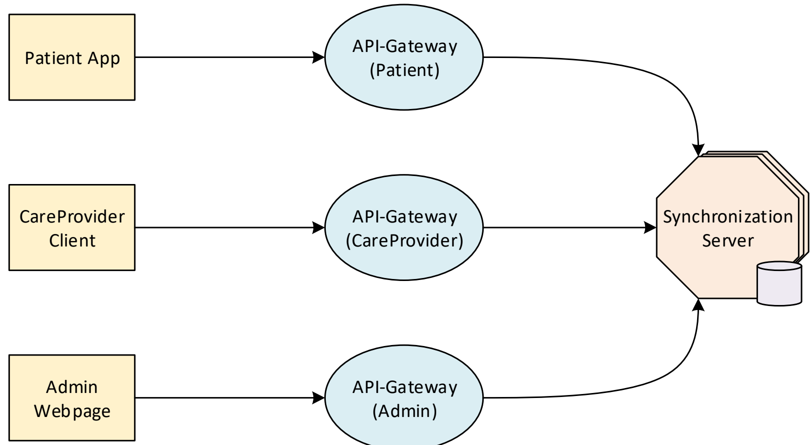


Abbildung 19 Architekturvariante Multi-Gateway-Monolith

Diese Variante geht noch einen Schritt weiter als Variante 1 und lagert die Verantwortung für die Zugriffsberechtigungen und die Komposition der Resultate in verschiedene API-Gateways aus. Dieser Ansatz kann mit dem Microservice-Pattern Backends for Frontends (BFF) [17] von Sam Newman verglichen werden, ausser, dass sich hinter diesen API-Gateways jeweils keine Microservices, sondern ein Monolith befindet. Die API-Gateways wären, in der Terminologie von Sam Newman, Edge Services.

6.1.4.1 *Eignungsuntersuchung*

Ungeeignet: Diese Variante könnte ein erster Schritt in einem Refactoring sein, löst jedoch die Probleme mit dem Single-Point-of-Failure nicht, da alle Parteien mit demselben Monolith kommunizieren.

6.1.5 Variante 3: Monolith-Per-Careprovider

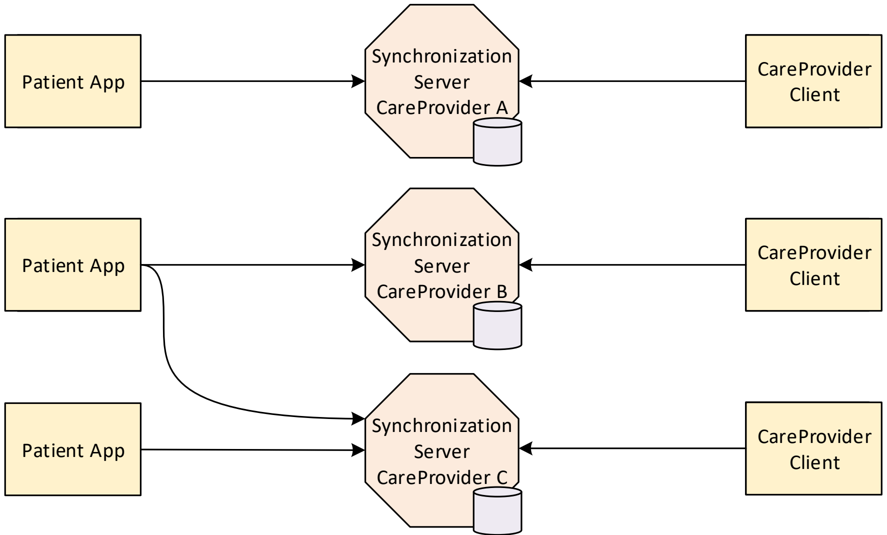


Abbildung 20 Architekturvariante Monolith-Per-Careprovider

Einen Monolithen pro *CareProvider* (Mandant) bereitzustellen ist eine mandantenbasierte Variante. Hierbei handelt es sich aber nicht um einen Microservice-Ansatz, da die Business-Logik des Monolithen nicht in unabhängige Services aufgeteilt wird.

6.1.5.1 *Eignungsuntersuchung*

Potentiell geeignet: Diese Variante würde das Problem der Lastenverteilung lösen, da jedes Spital mit seinem eigenen monolithischen *Synchronization Server* kommuniziert. Zusätzlich könnte man argumentieren, dass eine höhere Sicherheit gewährleistet wird, da die Daten pro Spital an einem separaten Ort gespeichert werden. Weiterhin besteht jedoch das Problem, dass die Komplexität des Monolithen durch den zusätzlichen Verwaltungsaufwand immer grösser wird. So müsste bei einem Update dieses bei jedem *Synchronization Server* eingespielt werden. Ebenfalls kann das Entwickeln der Applikation nicht direkt auf mehrere Teams aufgeteilt werden oder in mehreren Programmiersprachen geschrieben werden, wie es bei Microservices der Fall ist. Zudem wird bei dieser Variante die NFA 6 zwar verbessert, aber noch nicht erfüllt, die besagt, dass sich *CareProvider Client* und die *Patient App* nicht gegenseitig blockieren können sollen.

Diese Variante wird nicht weiterverfolgt, da dieser Bericht sich auf das cloud-native Refactoring in eine Microservice-Architektur konzentriert.

6.1.6 Variante 4: Microservice-Direct-Invocation

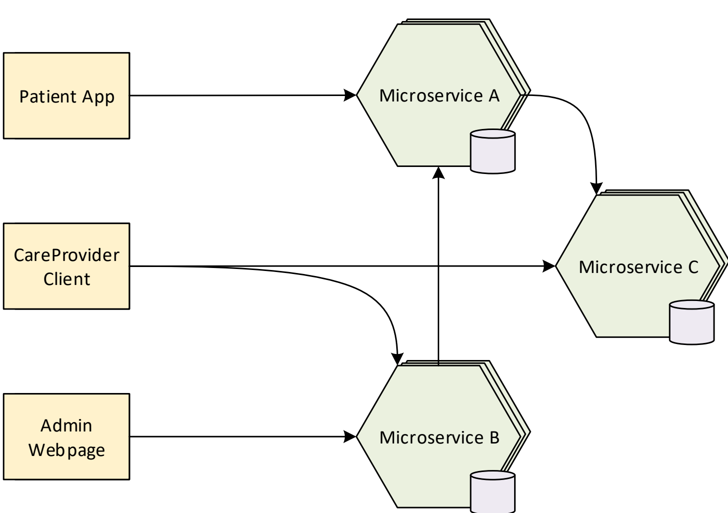


Abbildung 21 Architekturvariante Microservice-Direct-Invocation

In dieser Architekturvariante wurde der Monolith zur Illustration in mehrere Microservices, *Microservice A*, *Microservice B* und *Microservice C* aufgeteilt und können zum Teil direkt miteinander kommunizieren. Die jeweiligen Aufgaben der Microservices sind in der Longlist nicht von Bedeutung. Diese Microservices werden von den Clients jeweils direkt aufgerufen. Somit liegt die Verantwortung für die Orchestrierung der Abfragen bei den jeweiligen Clients.

6.1.6.1 *Eignungsuntersuchung*

Ungeeignet: Durch die fehlenden API-Gateways wird es komplizierter die Lastenverteilung sowie die Authentifizierung zu implementieren.

6.1.7 Variante 5: Single-Gateway-Multi-Microservice

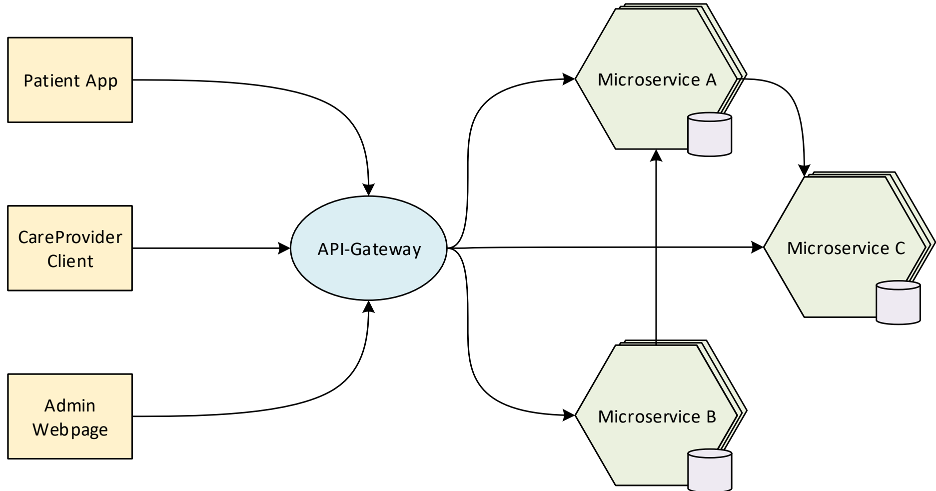


Abbildung 22 Architekturvariante Single-Gateway-Multi-Microservice

Im Gegensatz zur Variante *Microservice-Direct-Invocation*, werden die Microservices von den verschiedenen Parteien über ein API-Gateway aufgerufen.

6.1.7.1 *Eignungsuntersuchung*

Geeignet: Dieser Architekturvorschlag ist eine klassische Microservice-Architekturvariante, die durch das vorgeschaltete API-Gateway eine Lastenverteilung ermöglicht und die Anfragen an die zuständige Microservice-Instanz weiterleitet. Dies entspricht dem Enterprise Integration Pattern (EIP) Content-Based Router [18]. Im Optimalfall kann ein einzelner Microservice eine Abfrage komplett selber beantworten, ohne mit einem anderen Microservice kommunizieren zu müssen.

6.1.8 Variante 6: Multi-Gateway-Multi-Microservice

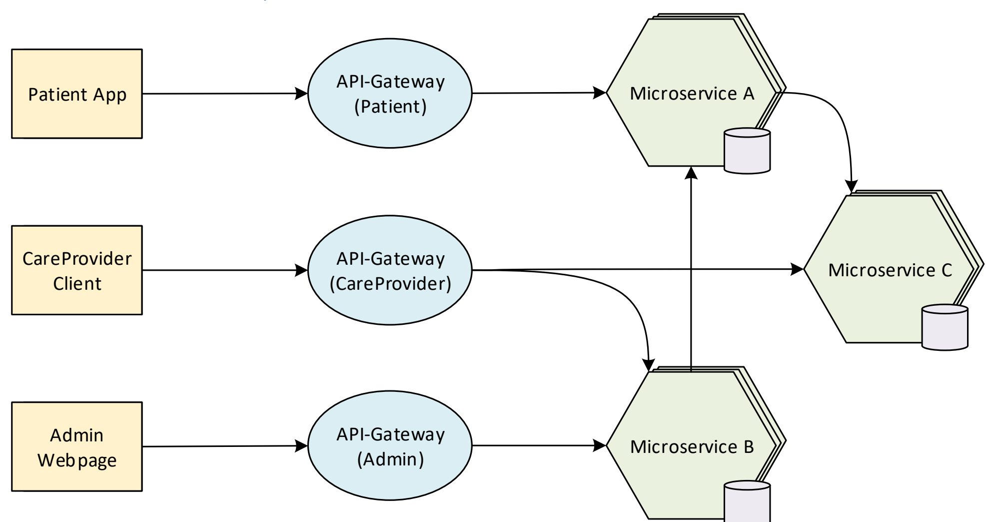


Abbildung 23 Architekturvariante Multi-Gateway-Multi-Microservice

Diese Variante ist eine Weiterentwicklung der *Single-Gateway-Microservice*-Variante und delegiert die Verantwortung für die Zugriffsberechtigung an die verschiedenen API-Gateways.

Wie schon bei der Variante *Multi-Gateway-Monolith,* sind die API-Gateways an dem Backends for Frontends-Pattern angelehnt.

6.1.8.1 *Eignungsuntersuchung*

Geeignet: Als Weiterentwicklung des vorhergehenden Architekturvorschlags *Single-Gateway-Multi-Microservice* besitzt diese Variante eine erhöhte Entkopplung, da die API-Gateways weiter aufgeteilt wurden und spezifischer für die jeweiligen Clients entwickelt werden können.

6.1.9 Variante 7: Gateway-Queue-Microservice

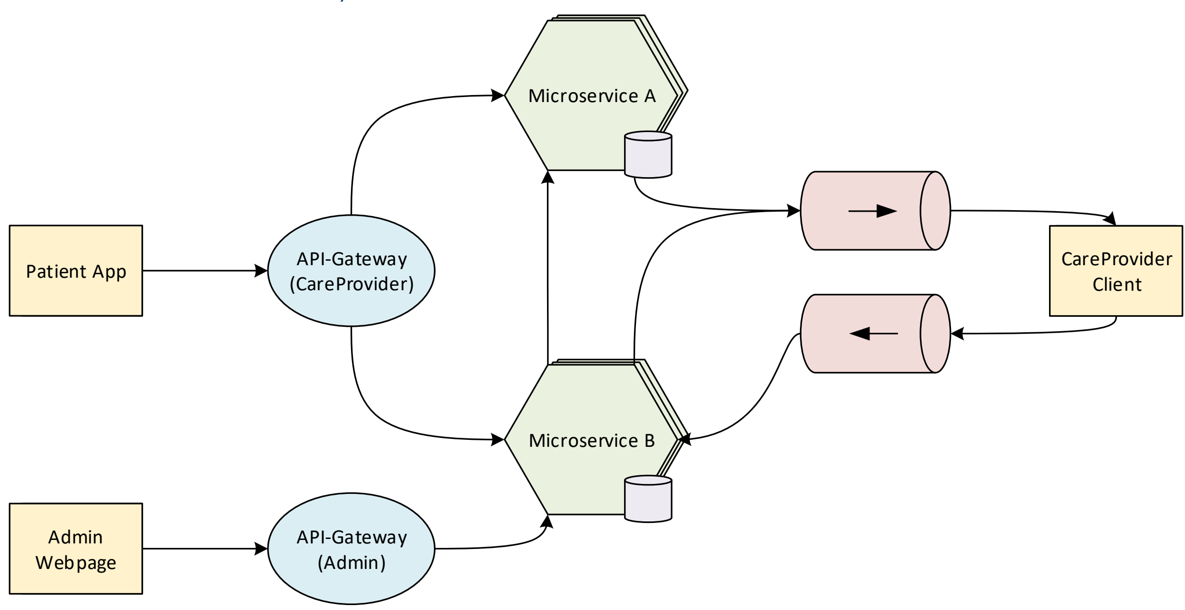


Abbildung 24 Architekturvariante Gateway-Queue-Microservice

Die Microservices werden durch das Einführen von Message Queues vom *CareProvider Client* getrennt. Die Queues müssten das EIP Guaranteed-Delivery [19] implementieren und würden ein Teil der aktuellen Datenbank ersetzen und als Speicher zwischen Spital und dem Microservice dienen, der für die Synchronisierung zuständig ist. Die Schnittstelle für die *Patient App* muss in dieser Variante nicht angepasst werden. Daten, die vom Patienten an das Spital gesendet werden, werden vom zuständigen Microservice in die Queue gespeichert. Daten, die wiederum vom Spital an den Patienten gesendet werden, werden in der Datenbank des zuständigen Microservices gespeichert, bis diese von der *Patient App* abgeholt werden. Die Queues können zum Beispiel mit RabbitMQ [20] implementiert werden. Ausserdem würden die Queues das Microservice-Pattern Smart Endpoints and Dumb Pipes [21] implementieren, indem sie nur die Daten speichern und transportieren und keine Mutationen an diesen vornehmen.

6.1.9.1 *Eignungsuntersuchung*

Geeignet: Diese Variante scheint ein geeigneter Ansatz zu sein. Zum einen wird die Dekomposition des Monolithen einfacher da die Komponenten durch die Message Queues zeitlich entkoppelt werden. Zum anderen, weil das Zwischenspeichern und Synchronisieren der Daten vom Patienten zum Spital von den Queues übernommen wird.

Diese Variante würde also nicht nur eine Dekomposition in Microservices vorsehen, sondern zusätzlich einen Teil der Business-Logik in eine andere Technologie, in diesem Falle Queues, auslagern.

6.1.10 Variante 8: Serverless-Direct-Database

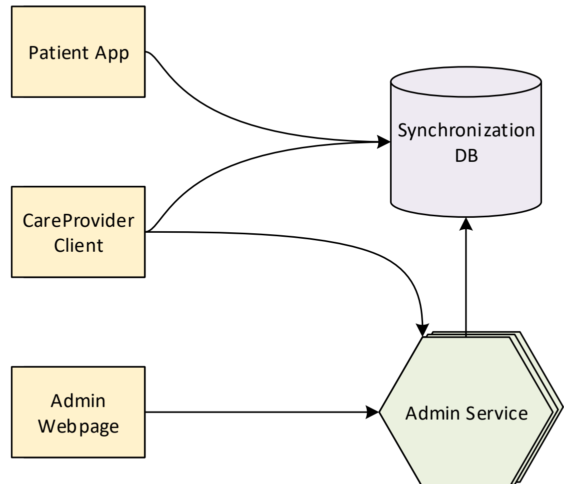


Abbildung 25 Architekturvariante Serverless-Direct-Database

Diese Variante besteht im Kern aus einer gemeinsamen Datenbank, auf welche die verschiedenen Parteien direkt zugreifen. Für komplexere Aufgaben, wie *CareProvider* erstellen und *CareProvider* registrieren, existiert der Microservice *Admin Service*. Dieser Microservice könnte auch mit einer Function as a Service (FaaS) [22] ersetzt werden. Bei einer solchen Variante könnte eine Datenbank eingesetzt werden die *Partition Tolerance* und *Availability* vom CAP-Theorem [23] unterstützt und auf *Consistency* verzichtet, um eine hohe Verfügbarkeit zu gewährleisten. Dieser Ansatz gleicht aber eher einem Serverless-Ansatz als einer Microservice-Architektur. Zusätzlich müsste die Datenbank ein sehr feingranulares und testbares Zugriffsberechtigungs-System anbieten.

6.1.10.1 *Eignungsuntersuchung*

Ungeeignet: Für das Szenario in dieser Arbeit ist diese Variante nicht geeignet, da sie zu wenig flexibel ist, wie zum Beispiel für Use Cases wie das Scannen eines Tickets oder die Registrierung eines neuen Spitals. Auch die Konfiguration der Zugriffsberechtigungen könnte sehr kompliziert werden, da diese direkt auf der Datenbank verwaltet wird und potentiell ist die Gewährleistung der Sicherheit schwieriger. In weniger sicherheitskritischen Systemen wäre ein solcher Ansatz jedoch durchaus interessant, da das Implementieren der Microservice-Middleware zwischen Datenbank und End-Benutzer wegfällt. Ein weiterer unerwünschter Nebeneffekt einer solchen Implementierung ist, dass die Kopplung zwischen Client und Datenbank sehr eng ist und jeder Client das Protokoll der Datenbank verwenden muss.

6.1.11 Resultat Longlist-Architekturvarianten

Nr.	Architekturvariante	Resultat
1	Single-Gateway-Monolith	**Ungeeignet**
2	Multi-Gateway-Monolith	**Ungeeignet**
3	Monolith-Per-Careprovider	**Potentiell geeignet**
4	Microservice-Direct-Invocation	**Ungeeignet**
5	Single-Gateway-Multi-Microservice	**Geeignet**
6	Multi-Gateway-Multi-Microservice	**Geeignet**
7	Gateway-Queue-Microservice	**Geeignet**
8	Serverless-Direct-Database	**Ungeeignet**

Tabelle 8 Resultat Longlist-Architekturvarianten

Die Longlist-Architekturvarianten-Analyse hat ergeben, dass drei Architekturvarianten für die weitere Analyse geeignet sind und für die weitere Shortlist-Analyse als Ausgangslage dienen, um weiterentwickelt und gegebenenfalls miteinander kombiniert zu werden:

- Single-Gateway-Multi-Microservice
- Multi-Gateway-Multi-Microservice
- Gateway-Queue-Microservice

Die Mandantenbasierte Architekturvariante *Monolith-Per-Careprovider* gilt als potentiell geeignet, wird aber nicht weiterverfolgt, da dieser Bericht sich auf das cloud-native Refactoring in eine Microservice-Architektur konzentriert.

6.2 Architekturvorschläge (Shortlist)

Die Microservices wurden anhand der entstandenen Domänen des Event Storming aus Abbildung 12 aufgeteilt. Eins-zu-eins ist das jedoch nicht möglich und sinnvoll, da das Event Storming wie umgesetzt das gesamte System mit den externen Applikationen betrachtet und die Microservice-Architektur sich nur auf den *Synchronization Server* (siehe *C4 System Context Diagram* in Abbildung 4) bezieht. Diese Aufteilung wurde danach mit den im vorhergehenden Abschnitt als geeignet bewerteten Architekturvarianten dargestellt. Die so entstandenen Architekturvorschläge sind somit eine Weiterentwicklung der Architekturvarianten und kommen gegebenenfalls miteinander kombiniert vor.

6.2.1 Microservice-Architekturvorschlag 1 (Ausgelagerte Synchronisation)

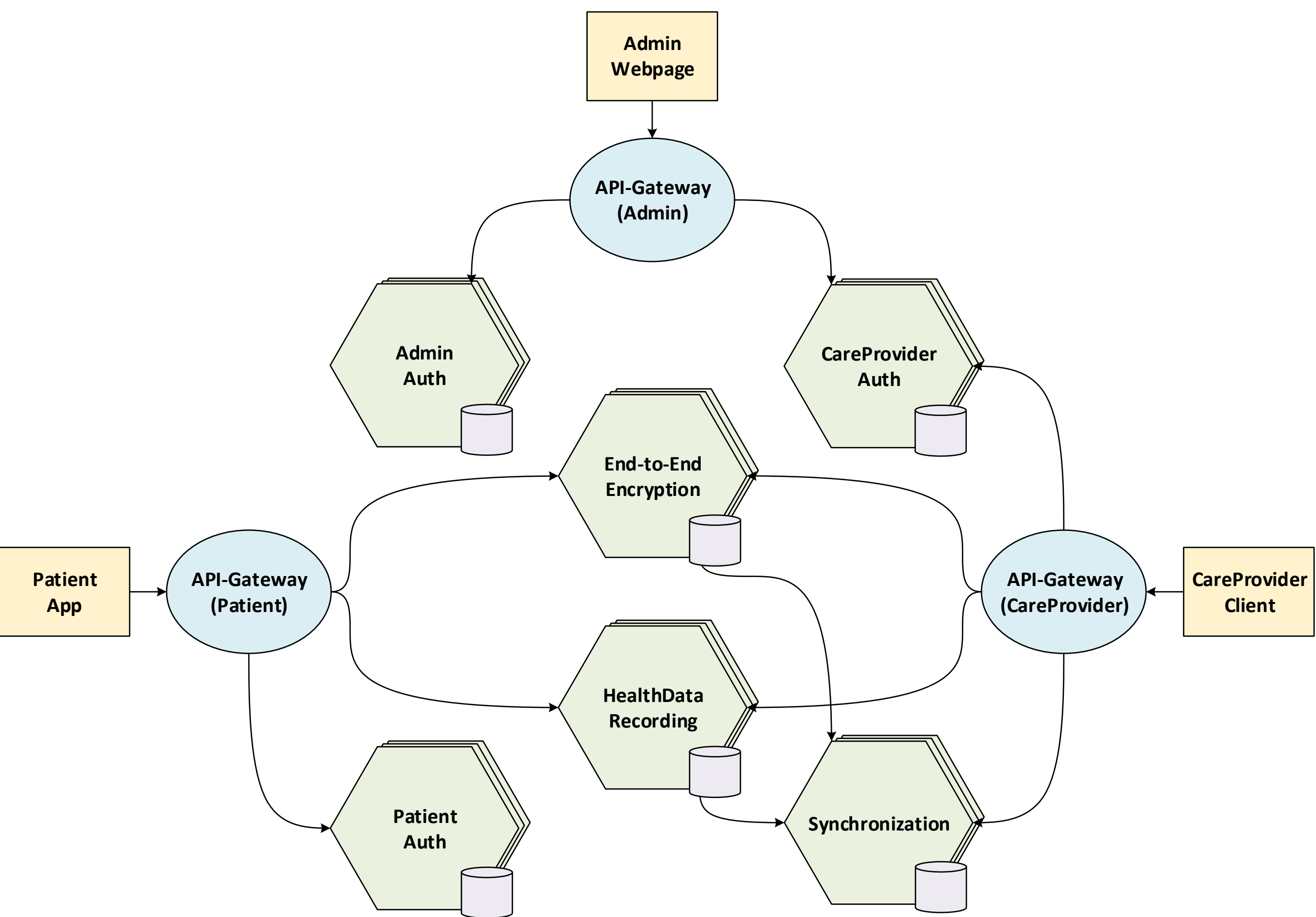


Abbildung 26 Microservice-Architekturvorschlag 1 (Ausgelagerte Synchronisation)

Dieser Architekturvorschlag ist eine Weiterentwicklung von Architekturvariante 6 *Multi-Gateway-Multi-Microservice* aus Abbildung 24.

Ein wichtiger Umbau, den diese Architektur verlangt, ist der Einsatz von einer stateless Authentication wie JSON Web Token (JWT) [24] für die Authentifizierung der verschiedenen Parteien. Aus Sicht der Performance ist es unabdingbar, dass jeder Microservice die Identität einer Anfrage identifizieren kann, ohne dabei eine Anfrage auf einen anderen Microservice oder eine Datenbank zu machen. Die Verschiedenen Auth-Services in Abbildung 26 müssen ein signiertes JWT ausstellen, deren Gültigkeit von den anderen Microservices festgestellt werden kann. Somit müssen die Microservices, welche die Business-Logik beinhalten, nicht über die Implementierung des Authentifizierungs-Mechanismus informiert sein.

Diese Architektur erlaubt es, abgesehen von der Authentifizierung, dass keine Änderungen an der API gemacht werden müssen. Sie konzentriert sich nur auf das, was hinter der API passiert. Somit beschränkt sich das cloud-native Refactoring ausschliesslich auf den *Synchronization Server*.

Ein weiteres Merkmal dieses Vorschlags ist der *Synchronization-Service*. Das Konzept der Synchronisierung ist technischer und nicht fachlicher Natur, weshalb es im Event Storming nicht identifiziert wurde. Der Microservice *Synchronization* in Abbildung 26 speichert sich die Information, welche Entitäten bereits synchronisiert wurden und welche nicht. Der Service wurde ausgelagert, da er viele Anfragen von den *CareProvider Clients* erhält und den Synchronisierungsstatus aus einer grossen Anzahl Datensätze berechnen muss.

6.2.1.1 *Vergleich der Aufteilung mit Event Storming*

Event Storming Domäne[1]	**Microservice**[2]
Verschlüsselter Daten-Transport	HealthDataRecording
Blutdruck-Erfassung	HealthDataRecording
E2E-Kanal-Initialisierung	End-to-End-Encryption
E2E-Kanal-Initialisierung	PatientAuth
Spital Registrierung	CareProviderAuth
Lizenzierung	CareProviderAuth
Spital-Benutzer-Verwaltung (Zugriffskontrolle)	*Microservice nicht vorhanden auf Synchronization Server*
Nicht modelliert	AdminAuth
Nicht modelliert	Synchronization

Tabelle 9 Vergleich Event Storming mit Microservice-Architekturvorschlag 1 (Ausgelagerte Synchronisation)

Wie aus dem Vergleich ersichtlich ist, konnte ein gutes Mapping von den Domänen des Event Stormings auf die Microservices abgeleitet werden. Dennoch ist es aus technischer Sicht nicht immer sinnvoll, die Grenzen genau gleich zu ziehen, wie es das Ergebnis des Event Storming vorsieht. Das wird am Beispiel der Domäne *E2E-Kanal-Initialisierung* aus dem Event Storming ersichtlich, die in die zwei Microservices *PatientAuth* und *End-to-End-Encryption* aufgeteilt wurde. Aus technischer Sicht macht es somit Sinn, die Authentifizierung des Gerätes und den Aufbau der Ende-zu-Ende-verschlüsselten Verbindung zu trennen.

[1] Domänen aus Event Storming in Abbildung 12
[2] Microservices aus Abbildung 26

6.2.2 Microservice-Architekturvorschlag 2 (Entkoppelte Synchronisation)

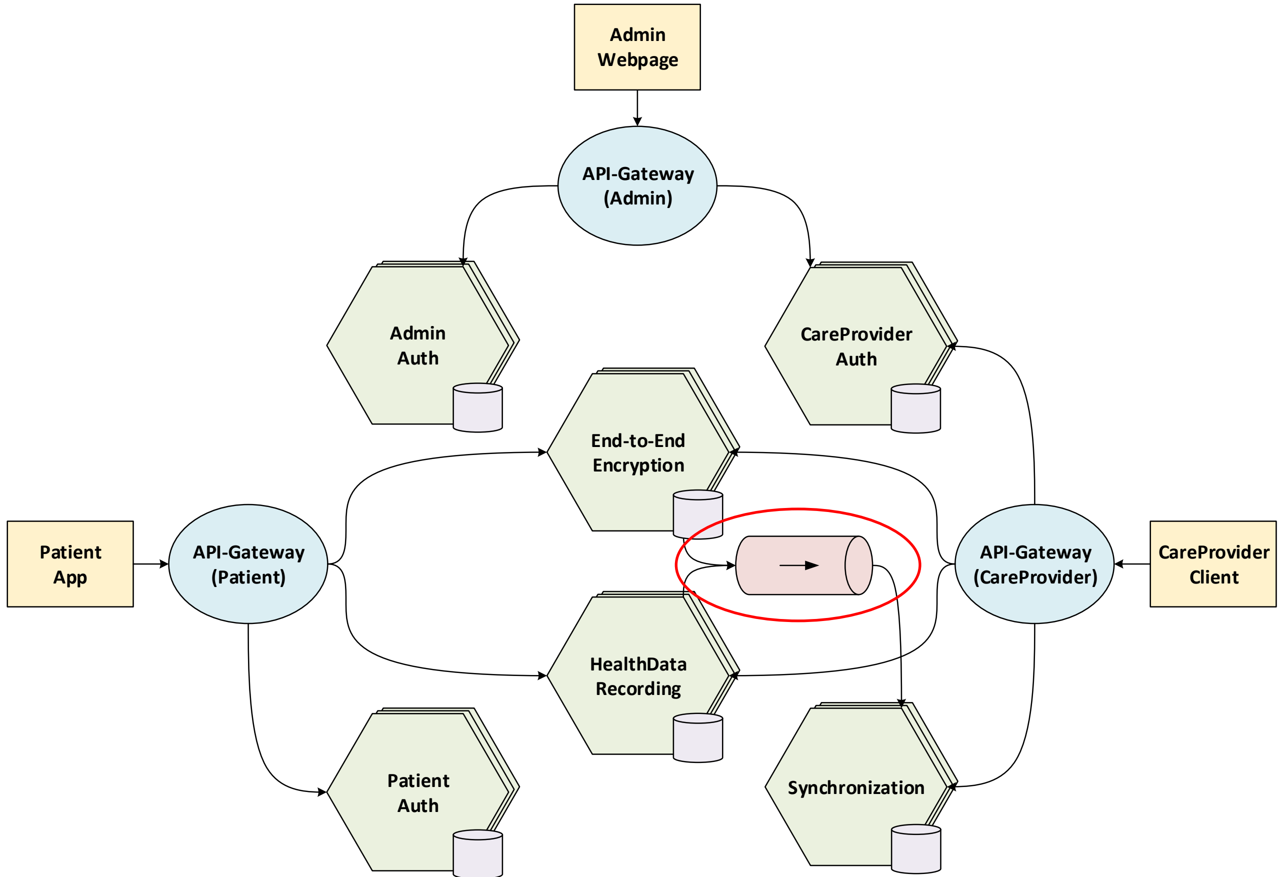


Abbildung 27 Microservice-Architekturvorschlag 2 (Entkoppelte Synchronisation)

Der Architekturvorschlag 2 ist eine Weiterentwicklung vom Architekturvorschlag 1.

In diesem Vorschlag wurde der Zugriff auf den Microservice *Synchronization* durch eine Message Queue entkoppelt. Ohne diese Queue wäre das Speichern von neuen Gesundheitsdaten nicht möglich, wenn der *Synchronization-Service* nicht verfügbar ist. Da die Übertragung der Gesundheitsdaten laut der nicht funktionalen Anforderung (NFA) 10 aus Tabelle 6 bis zu 20 Sekunden dauern darf, ist es kein Problem, wenn diese Informationen asynchron übertragen werden. Ebenfalls wird die NFA 8 adressiert, die verlangt, dass Benutzeraktionen nicht länger als 2 Sekunden dauern sollten. Wenn der Microservice *HealthDataRecording* den Microservice *Synchronization* synchron aufrufen würde, müsste dieser solange blockieren, bis die Anfrage abgearbeitet wurde. Dementsprechend würde das Absenden der Blutdruck-Daten länger dauern. In diesem Fall muss der Microservice *HealthDataRecording* nur die Information in die Message Queue schreiben und kann danach die Anfrage abschliessen, was deutlich effizienter ist.
Wichtig zu erwähnen ist, dass der Microservice *Synchronization* nur die Synchronisierungs-Informationen zum jeweiligen Gesundheitsdatensatz speichert. Das bedeutet, der Microservice *Synchronization* kann einem *CareProvider* Auskunft darüber geben, welche Daten zur Synchronisierung bereitstehen. Die effektiven Gesundheitsdaten werden vom *CareProvider* direkt beim *HealthDataRecording-Service* abgeholt.

Als Message Queue bietet sich RabbitMQ an, da mit dieser Nachrichten persistiert werden können und somit auch bei einem Neustart der Message Queue nichts verloren geht.

6.2.3 Microservice-Architekturvorschlag 3 (CareProvider Queues)

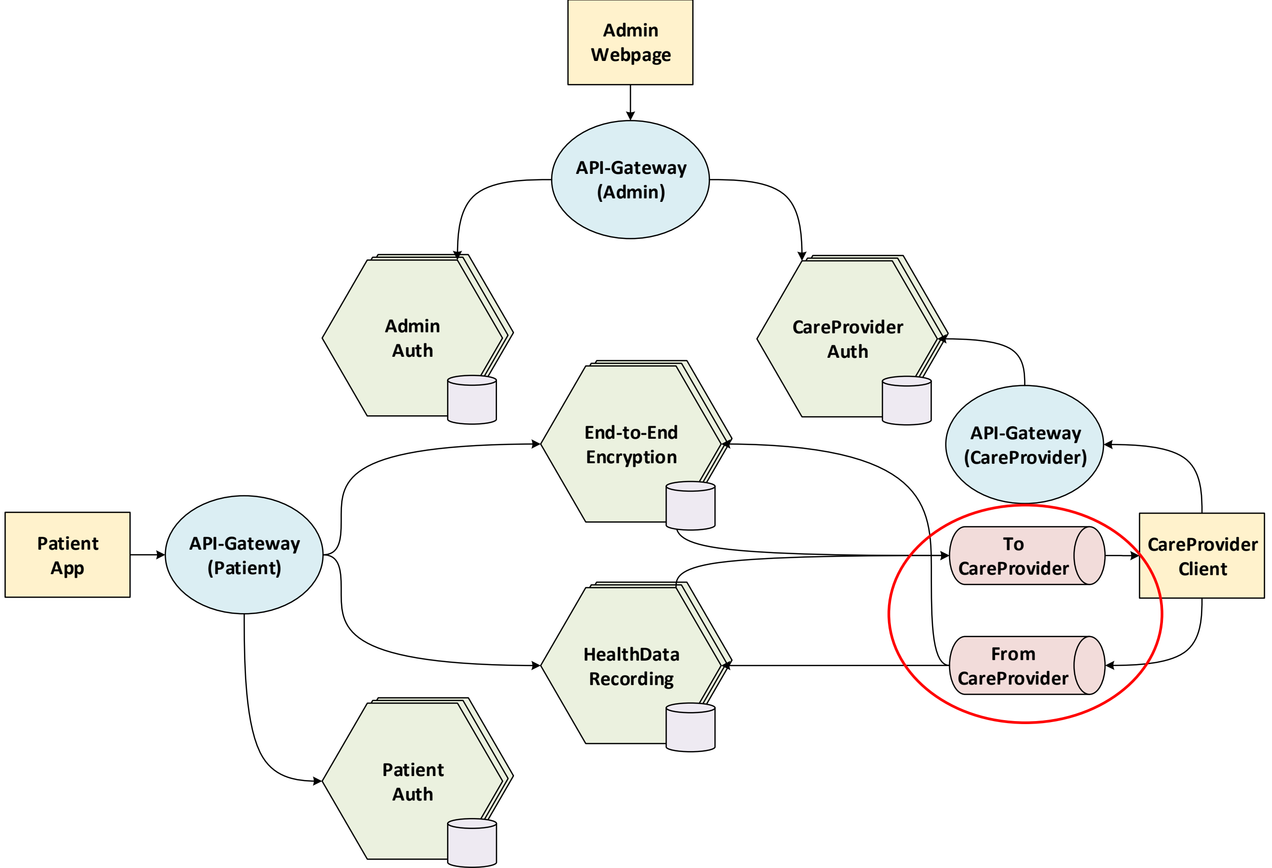


Abbildung 28 Microservice-Architekturvorschlag 3 (CareProvider Queues)

Dieser Architekturvorschlag entwickelt die Architekturvariante 7 *Gateway-Queue-Microservice* aus Abbildung 24 weiter.

Er basiert auf der Idee, dass die Daten nicht auf dem *Synchronization-Service* gespeichert, sondern in eine Message Queue geschrieben werden, der dann vom jeweiligen *CareProvider* abgearbeitet wird. Im Vergleich zum Architekturvorschlag 1 und 2 wird die gesamte Komplexität der Synchronisierung mithilfe zwei Message Queues ersetzt. Die Queues werden nur dafür benötigt Nachrichten zwischen den Microservices auszutauschen. Die gesamte Logik, was auf eine Nachricht getätigt werden muss, befindet sich in den Microservices, wodurch das Microservice-Pattern Smart Endpoints and Dumb Pipes [25] angewendet wird. Bei dieser Variante bleibt die API für die *Admin Webpage* und die *Patient App* unverändert, was wünschenswert ist. Ein grosser Nachteil ist aber, dass die Implementierung der Kommunikation mit dem *CareProvider* grösstenteils umgebaut werden muss. Was weiterhin bestehen bleibt, ist der *CareProviderAuth-Service*, der die Berechtigung und Authentifizierung der verschiedenen *CareProvider Client* verwaltet.

6.3 Vergleich der Architekturvorschläge

Folgende Architekturvorschläge wurden erstellt:

- **Architekturvorschlag 1: Ausgelagerte Synchronisation**
- **Architekturvorschlag 2: Entkoppelte Synchronisation**
- **Architekturvorschlag 3: CareProvider Queues**

Diese Architekturvorschläge werden unter Einbezug der nicht funktionalen Anforderungen (NFA), der sieben Microservice Tenets [26] [5] sowie nach Aufwand bewertet. Die sieben Microservice Tenets fassen eine Vielzahl von Definitionen und Charakteristiken von Microservices mit Stand 2016 der Community zusammen. Anhand dieser Vergleiche soll schlussendlich ein Architekturvorschlag evaluiert werden, der dann prototypisch umgesetzt wird.

6.3.1 Vergleich anhand NFA

Folgende Tabelle bewertet, wie gut die drei Architekturvorschläge im Vergleich zueinander die NFA erfüllen. Je mehr **+** Symbole vorhanden sind, desto besser wird die jeweilige NFA erfüllt.

NFA Nummer	1. Ausgelagerte Synchronisation	2. Entkoppelte Synchronisation	3. CareProvider Queues
1.	+++	+++	+++
2.	+++	+++	+++
3.	+++	+++	+++
4.	+	++	+++
5.	+	++	+++
6.	+	++	+++
7.	+++	+++	+++
8.	+++	+++	+++
9.	+++	+++	+++
10.	+++	+++	+++
11.	+++	++	++
12.	+++	+++	+++
13.	+++	+++	+++
14.	+++	+++	+++

Tabelle 10 Vergleich der Architekturvorschläge anhand NFA

Aus dem Vergleich wird ersichtlich, dass viele NFA gleich gut durch die Architekturvorschläge erfüllt werden. Dies lässt sich damit begründen, dass die drei Architekturvorschläge im Grundsatz die gleiche Aufteilung in Microservices vorsehen. Die Unterschiede entstehen vor allem durch den Einsatz von verschiedenen Technologien. Ein weiterer Grund dafür ist, dass nicht alle NFA auch Architecturally Significant Requirements (ASR) [27] sind und somit durch eine Veränderung der Architektur nicht beeinflusst werden. Ein gutes Beispiel dafür ist die NFA 12, die eine bestimmte Testabdeckung verlangt. Die Testabdeckung kann unabhängig von der Architektur erreicht werden.

Unterschiede weisen die im Vergleich markierten NFA 4 bis 6 und 11 auf. Da die Bewertung von der NFA 4 und 5 aber mithilfe von der NFA 6 umfasst werden, sind diese nicht weiter erklärt:

« NFA 6: Die Komponente beim Gesundheitsdienstleister und der App des Patienten sollen sich nicht gegenseitig blockieren können. »
Der Architekturvorschlag 1 hat zwar bereits eine Entkopplung der Synchronisation, besitzt jedoch immer noch direkte Abhängigkeiten zwischen dem *HealthDataRecording-Service* und dem *Synchronization-Service* und da beide dieser Services direkt vom Gesundheitsdienstleister aufgerufen werden können, kann solch ein Aufruf auf den *Synchronization-Service* nach wie vor den *HealthDataRecording-Service* blockieren. Mit der Einführung der Message Queue im Architekturvorschlag 2 zwischen den Microservices *HealthDataRecording* und *End-to-End-Encryption* und dem *Synchronization-Service,* wird diese Abhängigkeit gelöst und somit die Unabhängigkeit des *Synchronization-Services* erreicht. Am besten wird diese NFA durch den Architekturvorschlag 3 gelöst, da dieser mithilfe von Message Queues den *Synchronization-Service* ersetzt und somit verhindert, dass der CareProvider Client und die Patient App direkt mit denselben Microservices kommunizieren.

« NFA 11: Es sollen nicht mehr als 4 besser 3 verschiedene Frameworks/Technologien (z. B. Spring, Angular, PostgreSQL) in der ganzen Applikation eingesetzt werden, um ein möglichst grosses Wissen auf diesen Technologien zu haben. »
Da der Architekturvorschlag 1 als einziger keine Message Queue besitzt, besteht er aus einer Technologie weniger, wie die anderen beiden Architekturvorschläge und erfüllt dieses Kriterium somit besser. Damit weist dieser auch bei einer Umsetzung weniger technische Risiken auf.

6.3.1.1 *Resultat Vergleich mithilfe NFA*
Aus dem Vergleich anhand der NFA resultiert der Architekturvorschlag 3 *CareProvider Queues* als am besten geeignet.

6.3.2 Vergleich anhand der sieben Microservice Tenets
Folgende Tabelle bewertet, wie gut die drei Architekturvorschläge im Vergleich zueinander die sieben Microservice Tenets erfüllen. Je mehr **+** Symbole vorhanden sind, desto besser werden diese Microservice Tenets erfüllt.

Microservice Tenet	**1. Ausgelagerte Synchronisation**	**2. Entkoppelte Synchronisation**	**3. CareProvider Queues**
1. Independent-X (X = Deployment, Scaling, Change): Unabhängige Bereitstellung, Skalierbarkeit und Änderbarkeit.	++	+++	+
2. Business Alignment: Die Aufteilung der Microservices orientieren sich an den bestehenden Strukturen des Geschäfts. Dies kann zum Beispiel mit Domain-Driven Design (DDD) erreicht werden.	+++	+++	+++
3. IDEAL Cloud Architectures: **I**solated State, **D**istribution/Decomposition, **E**lasticity, **A**utomation und **L**oose Coupling der Cloud Architecture. Beispiel: Twelve-Factor-App [28]	++	+++	+++

4. Polyglot Programming and Persistence: Die Microservices können in unterschiedlichen Programmiersprachen implementiert sein und eine individuelle Persistierung der Daten verwenden.	+++	+++	+
5. Containerization and Clustering: Die Microservices können in Container wie zum Beispiel Docker [29] gestartet werden und mit Container-Orchestrierungs-Programmen wie Kubernetes [30] in Clustern bereitgestellt werden.	+++	+++	+++
6. Decentralization & Automation (CI/CD): Während des Entwickelns der Microservices besteht ein hoher Grad von Autonomie und Automatisierung.	+++	+++	+++
7. Service Monitoring (DevOps Way): Die Konfiguration-, Performance- und Fehlerüberwachung sind implementiert und ermöglichen das Überwachen der Microservices.	+++	++	++

Tabelle 11 Vergleich der Architekturvorschläge anhand der Seven Microservice Tenets

Die sieben Microservice Tenets werden von den Architekturvorschlägen unterschiedlich gut erfüllt. Unterschiede sind bei den im Vergleich markierten Microservice Tenets 1, 3, 4, und 7 vorhanden:

« 1. Independent-X (X = Deployment, Scaling, Change) »
Da der Architekturvorschlag 3 die Daten in Message Queues persistiert, ist die Änderbarkeit von den drei Architekturvorschlägen am wenigsten gewährleistet. Zusätzlich muss der *CareProvider Client* zur Verwendung der Message Queues umprogrammiert werden und beeinträchtigt die Änderbarkeit somit weiter. Der Architekturvorschlag 2 ist dank der Verwendung von internen Message Queues zeitlich besser entkoppelt und deshalb, mit Sicht auf den ersten Microservice Tenet, besser geeignet als der Architekturvorschlag 1.

« 3. IDEAL Cloud Architectures »
Durch die direkte Kopplung zwischen dem *HealthDataRecording-Service* und dem *Synchronization-Service* des Architekturvorschlags 1 erfüllt dieser den Punkt von Loose Coupling im Vergleich mit den anderen beiden Architekturvorschlägen weniger gut.

« 4. Polyglot Programming and Persistence »
Man kann argumentieren, dass die Verwendung der Message Queues, wie sie in Architekturvorschlag 3 vorgesehen ist, ein Microservice-Antipattern ist. Polyglot Persistence wird in diesem Vorschlag stark verletzt, da die Queues eine gemeinsame Datenbank darstellen. In diesem Fall wird sogar das externe System dazu gezwungen diese Technik einzusetzen. Das bedeutet jedoch nicht, dass dies eine falsche oder schlechte Lösung ist, aber der vierte Microservice Tenet wird damit klar verletzt.

« 7. Service Monitoring (DevOps Way) »
Das Monitoring der Daten im Architekturvorschlag 2 und 3 wird durch das Verwenden von Message Queues komplizierter, da eine zusätzliche Technologie verwendet wird, die überwacht werden muss.

6.3.2.1 *Resultat Vergleich mithilfe der sieben Microservice Tenets*

Aus dem Vergleich anhand der sieben Microservice Tenets resultiert der Architekturvorschlag 2 *Entkoppelte Synchronisation* als am besten geeignet.

6.3.3 Vergleich des Aufwandes

Folgende Tabelle vergleicht die drei Architekturvorschläge anhand des Aufwands. Je mehr **+** Symbole vorhanden sind, desto weniger Aufwand wird für den aktuellen Punkt benötigt.

Aufwand	1. Ausgelagerte Synchronisation	2. Entkoppelte Synchronisation	3. CareProvider Queues
Entwicklungsaufwand	+++	++	+
Bereitstellungsaufwand	+++	++	+
Unterhalt	+++	++	+

Tabelle 12 Vergleich des Aufwands der Architekturvorschläge

Aus obiger Tabelle ist klar ersichtlich, dass Architekturvorschlag 1 am wenigsten Aufwand bedeutet. Dies ist darauf zurückzuführen, dass dieser Vorschlag mit weniger Technologien auskommt (ohne Message Queue) und in diesem Fall am wenigsten entwickelt, verteilt und unterhalten werden muss. Architekturvorschlag 2 verursacht durch die zusätzlichen Queues Mehraufwand in Entwicklung, Verteilung und Unterhalt. Da bei Architekturvorschlag 3 nicht nur der *Synchronization Server* betroffen ist, sondern auch der *CareProvider Client* an die Message Queue angepasst werden und somit umgeschrieben werden muss, bedeutet dies am meisten Entwicklungs- sowie Bereitstellungs- und Unterhaltsaufwand.

6.3.3.1 *Resultat Vergleich des Aufwands*

Aus dem Vergleich des Aufwands resultiert der Architekturvorschlag 1 *Ausgelagerte Synchronisation* als am besten geeignet.

6.4 Architekturvorschlag Entscheid

Der Architekturvorschlag 3 löst, wie aus dem Vergleich anhand der NFA ersichtlich, die Aufgabenstellung am besten. Er entkoppelt die Microservices am besten und besitzt somit am wenigsten Abhängigkeit. Der Architekturvorschlag 3 verursacht jedoch durch die zusätzliche Abhängigkeit zwischen den Software-Systemen *Synchronization-Service* und *CareProvider Client* ein erhöhter Mehraufwand und wird deshalb nicht umgesetzt, zumal der Code für das externe System *CareProvider Client* für diese Arbeit nicht vorliegt.

Als umzusetzende Architektur für den Prototyp des *Synchronization Servers* wird Architekturvorschlag 1 gewählt, da er am wenigsten Aufwand bedeutet, mit dem Ziel zu untersuchen, ob und wie die ursprüngliche Applikation in ihre Domänen in Microservices unterteilt werden kann. Ausserdem kommt Architekturvorschlag 1 mit am wenigsten technischen Risiken aus, da er eine Technologie (Message Queue) weniger implementiert wie die beiden anderen Architekturvorschläge. Dies bedeutet, dass mit diesem Entscheid in Kauf genommen wird, dass die NFA 6 nicht erfüllt werden wird, da sich die Microservices weiterhin noch blockieren können.

Als Ausblick kann der Architekturvorschlag 1 nach dem Umbau mit einer Message Queue erweitert werden, um zum Architekturvorschlag 2 zu gelangen, der die Microservices dann voneinander entkoppelt.

7 Umsetzung

Nach der kritischen Evaluation des Architekturvorschlags wird der Architekturvorschlag 1 *Ausgelagerte Synchronisation* umgesetzt, getestet und mit einem Review der nicht funktionalen Anforderungen auf seine Qualität geprüft.

7.1 Implementierungsdetails

Die Implementierungsdetails zeigen einen schrittweisen Umbau der monolithischen Architektur in eine Microservice-Architektur.

7.1.1 Schritt 1: Authentifizierung zu JWT umbauen

Dieser Umbau ist vonnöten, da aktuell jedes Spital ein Schlüssel besitzt, mit dem es jede Anfrage an den *Synchronization Server* signiert und sich somit authentisiert. Der Monolith liest den Key des jeweiligen Spitals aus seiner Datenbank und verifiziert damit die Anfrage. Diese Implementierung würde also bei einem Umbau zu Microservices von jedem Microservice verlangen, dass er auf den Key des Spitals Zugriff hat, um eine Anfrage eines Spitals selbstständig zu verifizieren. Das würde das Microservice-Pattern Seperation of Concerns (SoC) [31] verletzen, das mit der Einführung von Microservices angestrebt wird. Jeder Microservice müsste sich darum kümmern, wie er ein Spital authentifizieren kann. Um dieses Problem zu adressieren, schlägt die Community [32] die Einführung von JSON Web Tokens (JWT) vor, um eine stateless Authentication zu erreichen. Konkret bedeutet dies, dass der *CareProviderAuth-Service* (siehe Abbildung 26) nach der erfolgreichen Authentifizierung des Spitales, ein JWT ausstellt, das er mit seinem Private-Key signiert. Ab diesem Zeitpunkt sendet das Spital dieses Token bei jeder Anfrage an den *Synchronization-Service* mit. Alle Microservices, die Anfragen des Spitales bearbeiten, sind im Besitz des Public-Keys des *CareProviderAuth-Services* und können damit die Gültigkeit des Tokens überprüfen und somit auf die Integrität des Anfragestellers vertrauen. Somit ist der Concern der Authentifizierung eines Spitals dem *CareProviderAuth-Service* anvertraut.

Learning 1: Jede Applikation, die potentiell zu einer Microservice-Architektur umgebaut werden könnte, sollte von Anfang an mit JSON Web Tokens (JWT) absichert werden, um einen aufwändigen Umbau zu vermeiden.

7.1.2 Schritt 2: API-Gateway vorschalten

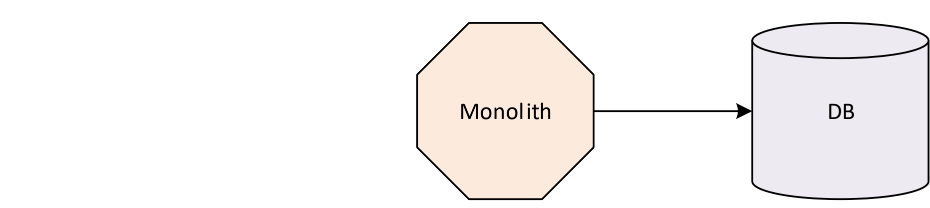


Abbildung 29 Refactoring: Ausgangslage

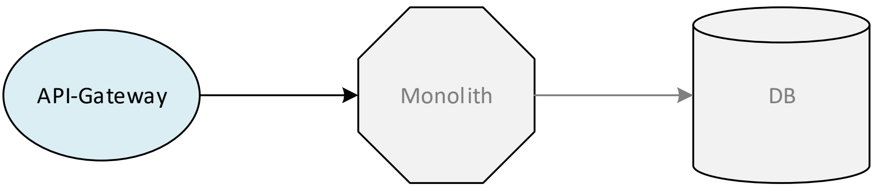


Abbildung 30 Refactoring: API-Gateway hinzufügen

Vor dem Monolithen wird ein API-Gateway vorgeschaltet, das als Endpunkt für die Anfragen fungiert und diese danach an den Monolithen weiterleitet. Da die in dieser Arbeit behandelte Applikation mit dem Spring-Framework [33] arbeitet, wurde das Spring Cloud Gateway [34] eingesetzt.

Das Spring Cloud Gateway routet HTTP-Anfragen. Speziell ist, dass dieses eine gute Integration mit anderen Projekten des Spring-Frameworks hat. Für die drei Applikationen die mit dem Monolithen kommunizieren, der *Admin Webpage*, dem *CareProvider Client* und der *Patient App*, wurde jeweils ein solches API-Gateway erstellt.

Learning 2: Als Vorbereitung, um Anfragen in Zukunft an den verantwortlichen Microservice weiterzuleiten, ist es sinnvoll, ein API-Gateway beim Monolithen vorzuschalten.

7.1.3 Schritt 3: Duplikat des Monolithen für den *CareProviderAuth-Service* erstellen

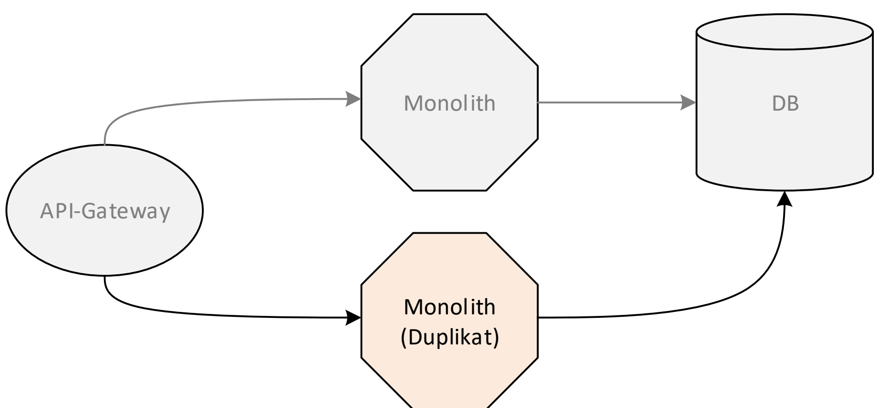


Abbildung 31 Refactoring: Monolith duplizieren

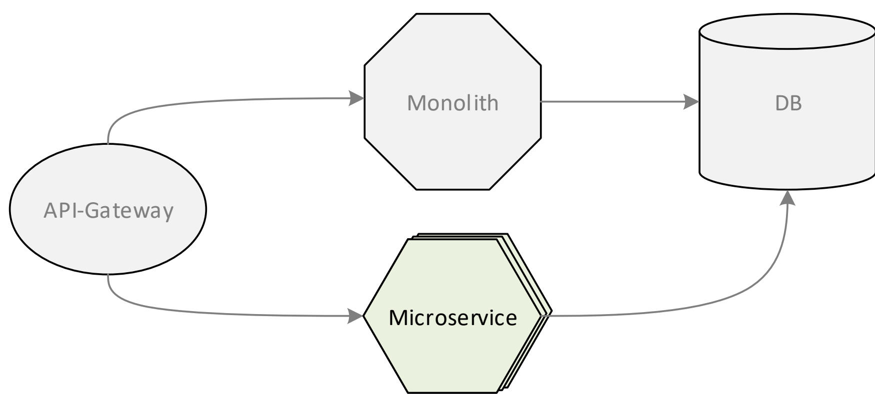


Abbildung 32 Refactoring: Monolith-Duplikat auf Microservice reduzieren

In diesem Schritt wird der erste Microservice erstellt. Für diesen Umbau gibt es zwei Varianten: Den Microservice von Grund auf neu aufzubauen oder den Monolithen zu kopieren und den nicht benötigten Code löschen und den Monolithen um die Microservice-Funktionalität zu reduzieren. In diesem Projekt wurde Zweiteres angewendet. Das hat den Vorteil, dass die Unit-Tests des bestehenden Codes übernommen werden können und die weiterhin benötigten REST-Endpunkte gleichbleiben. Weiter kann schnell festgestellt werden, ob auch tatsächlich keine wichtigen Abhängigkeiten für die Aufteilung aus dem Event Storming vergessen wurden. Der neu entstandene Microservice greift auf dieselbe Datenbank zu wie der Monolith. Das hat den Vorteil, dass auch bei der Anbindung der Datenbank vorerst keine Änderungen eingeführt werden müssen. Dies wird bewusst so umgesetzt, um so schnell wie möglich festzustellen, ob sich die Business-Logik tatsächlich in einem isolierten Microservice abbilden lässt.

Learning 3: Duplizieren des Monolithen und weglöschen der nicht benötigten Funktionalität ist eine effiziente Vorgehensweise, die schnell aufzeigt, ob die geplanten Bounded Contexts aus dem Event Storming in der Implementierung des Microservices umgesetzt werden können.

Learning 4: Durch das Anbinden der Datenbank des Monolithen an den Microservice hat man von Anfang an eine lauffähige Version des Microservices und kann diesen schrittweise auf die geplante Funktionalität reduzieren.

7.1.4 Schritt 4: Duplikat des Monolithen für den *AdminAuth-Service* erstellen

Dieselbe Vorgehensweise wie in Schritt 3 wird für den *AdminAuth-Service* angewendet.

7.1.5 Schritt 5: Duplikat des Monolithen für den *Synchronization-Service* erstellen

Die Aufgabe des *Synchronization-Service* besteht darin, den *CareProvider* darüber zu informieren, welche Daten bereits abgeholt wurden und welche noch synchronisiert werden müssen. Diese Information wird in der Datenbank des Monolithen in einer Tabelle gespeichert. Auch für den *Synchronization-Service* wird ein Duplikat des Monolithen erstellt und somit kann dieser die Tabelle wiederverwenden. Der wichtigste Umbau bei diesem Schritt ist, dass der Monolith die Synchronisierungs-Information nicht selbst in die Datenbank schreibt, sondern eine HTTP-Anfrage an den *Synchronization-Service* macht.

Learning 5: Ein direkter HTTP-Aufruf zwischen zwei Microservices verschlechtert die Stabilität des ganzen Systems. Ein solcher Aufruf sollte also unbedingt vermieden werden oder mit einem asynchronen Aufruf über eine Message Queue implementiert werden.

7.1.6 Schritt 6: Auftrennen der Datenbank

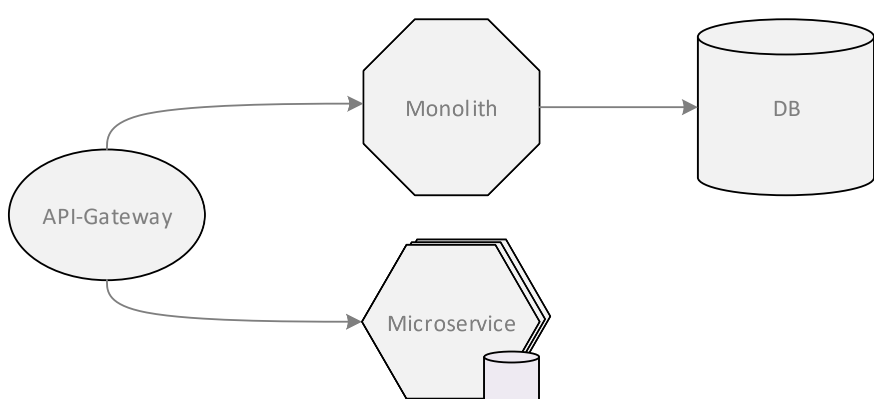


Abbildung 33 Refactoring: Auftrennen der Datenbank

Für den Microservice wird eine eigene Datenbank erstellt und die benötigten Daten werden in diese portiert. Danach wird der Microservice an diese Datenbank angeschlossen und somit wird ein komplett eigenständiger Microservice erreicht. Dies wird für alle drei Microservices angewendet.

Learning 6: Wenn eine Applikation in der Zukunft zu einer Microservice-Architektur umgebaut werden könnte, sollten Datenbank-Funktionalitäten die eine starke Kopplung verursachen, vermieden werden. Dazu gehören automatisch inkrementierte IDs, Fremdschlüssel oder Zeitstempel, die essentiell für gewisse Abfragen sind.

Ab diesem Punkt wird mit der Implementierung des Architekturvorschlags 1 aufgehört, da ein weiterer Umbau als Grundlage für die Ergebnisdiskussion keinen nennenswerten Mehrwert liefern würde. Dafür wird die eingeplante Zeit eingesetzt, um eine Entkopplung der Microservices durch eine Message Queue zu erreichen, wie es im Architekturvorschlag 2 ersichtlich ist.

7.1.7 Schritt 7: Asynchrone Microservice-Kommunikation einführen

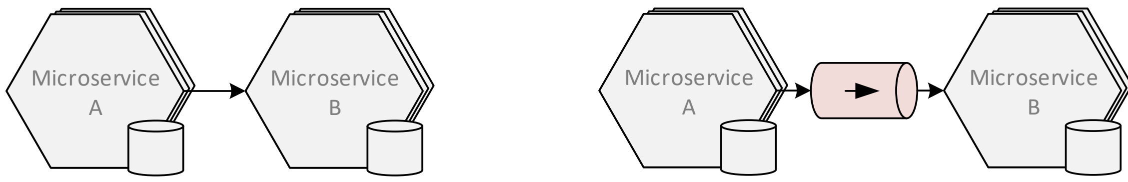


Abbildung 35 Refactoring: Synchrone Microservices

Abbildung 34 Refactoring: Asynchrone Microservices

Im Schritt 5 wird erwähnt, dass ein synchroner Aufruf zwischen Monolith und *Synchronization-Service* stattfindet. Im Architektur-Entscheid ist dies für den Prototyp so vorgesehen. Die Stabilität dieser Implementierung ist jedoch nicht zufriedenstellend, da der synchrone Aufruf eine Verfügbarkeit beider Instanzen voraussetzt. Durch die Einführung einer Message Queue, wie dieser im Architekturvorschlag 2 vorgesehen ist, wird diese starke Kopplung gelöst.

Aus Interesse, ob sich diese Annahme bestätigt, wird dieser Schritt zusätzlich noch implementiert.

7.1.8 Fazit

Die zuvor beschriebene Vorgehensweise erwies sich als effizient, da sie sehr früh aufzeigt, wo ein geplanter Umbau aufgrund von nicht vorhergesehenen Abhängigkeiten nicht möglich ist. Die geplante Aufteilung in Microservices konnte, wie im Architekturvorschlag 1 beschrieben, soweit umgesetzt erreicht werden. Einen grossen Umbau und somit einen Breaking-Change an der API, hat der notwendige Umbau zur JWT-Authentifizierung verursacht. Die Stabilität des gesamten *Synchronization Servers* wurde durch die direkte Abhängigkeit zwischen dem verbleibenden Monolithen und *Synchronization-Service* schlechter. Diese konnte aber unter Einsatz von einer Message Queue wieder erreicht werden, wodurch der Unterschied von Architekturvorschlag 1 und 2 implementiert ist.

Die Unit-Tests konnten auch nach der Aufteilung übernommen werden. Bei den Integrationstests, die mit Spring REST Docs [35] realisiert wurden, war dies nicht möglich. Das Konzept der Spring REST Docs Integrationstests ist es, dass der Monolith inklusive Datenbank gestartet wird und Szenarien anhand einer Abfolge von API-Aufrufen getestet werden. Spring REST Docs Integrationstests stellen nicht nur sicher, dass die API richtig funktioniert, sondern generieren zusätzlich noch eine API-Dokumentation in AsciiDoc [36]. Der Einsatz dieses Frameworks ist nach dem Umbau zu Microservices nicht mehr möglich, weshalb Postman [37], ein externes API-Testing Tool, eingesetzt wird.

7.2 Testing

Als Continuous Integration (CI)-System wird GitLab CI [38] verwendet, da der gesamte Code in einem GitLab Repository verwaltet wird und die GitLab eigene CI bereits integriert ist und aktiviert werden kann. Sobald also ein Entwickler Code in dieses Repository eincheckt, triggert dies die Pipeline zum Builden, Testen und Validieren des neuen Codes. Somit kann fehlerhafter Code automatisiert entdeckt werden und das Zusammenführen von solchem Code verhindert werden.

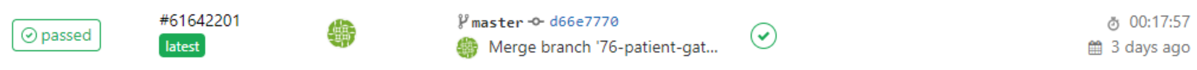


Abbildung 36 GitLab CI Pipeline Ergebnis

GitLab bietet neben CI auch Continuous Delivery (CD) [39] an, eine automatisierte Auslieferung von CI validiertem Code, der diesen in einer Testumgebung oder Produktivumgebung verteilt. In diesem Projekt wird dies jedoch nicht eingesetzt, da es keine Erkenntnisse zu den in dieser Arbeit behandelten Fragen bringen würde.

Da eine Microservice-Architektur aus verschiedenen unabhängigen Microservices besteht, bekommen Integrationstest eine hohe Relevanz, um das korrekte Zusammenspiel festzustellen. In der vorgegebenen Implementierung wurden Spring REST Docs Integrationstests eingesetzt, um den Monolithen mit der Datenbank zu testen. Diese Tests sind jedoch nicht geeignet um externe Abhängigkeiten, wie andere Microservices, im Verbund zu testen. Um aussagekräftige Integrationstests zu erstellen, wird Postman eingesetzt, mit welchem HTTP-Anfragen erstellt werden können, um diese an die API zu senden und den mit erwarteten Resultaten zu testen.

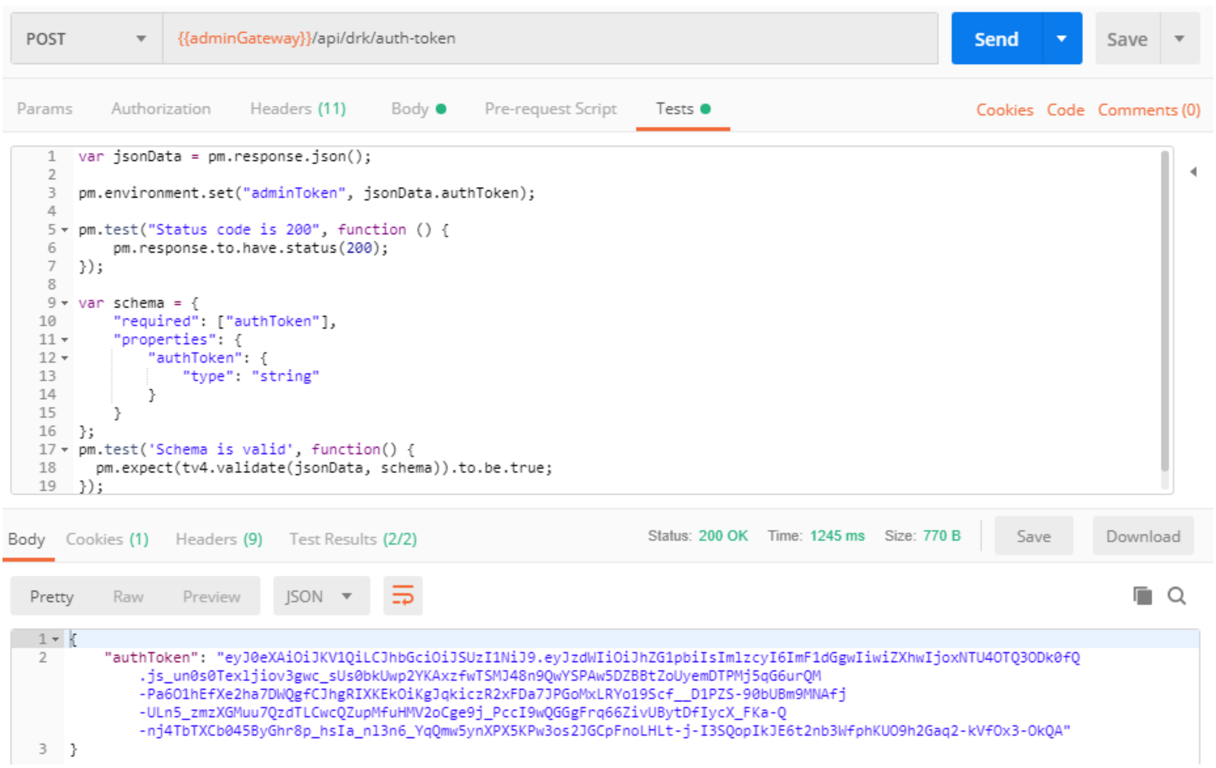


Abbildung 37 Integrationstest mit Postman

Postman bietet die Funktionalität, dass die Resultate der HTTP-Anfragen in Variablen gespeichert werden und bei den folgenden HTTP-Anfragen wiederverwendet werden können. In der obigen Abbildung wird dies beispielsweise erreicht, indem die Umgebungsvariable namens *adminToken* auf den Wert vom *authToken* gesetzt wird. Gerade bei einem solchen Authentifizierungs-Token, das bei

allen geschützten Endpunkten mitgeschickt werden muss, ist es nützlich, dieses mit Variablen-Referenzen wiederverwenden zu können.

Mehrere HTTP-Anfragen können in Postman in einer Collection zusammengefasst werden. Postman bietet die Möglichkeit, eine solche Collection sequentiell auszuführen, um Integrationstests zu realisieren.

Um das cloud-native Refactoring zu den Microservices zu testen, wurde ein Szenario ausgewählt, welches das Zusammenspiel aller Microservices testet.

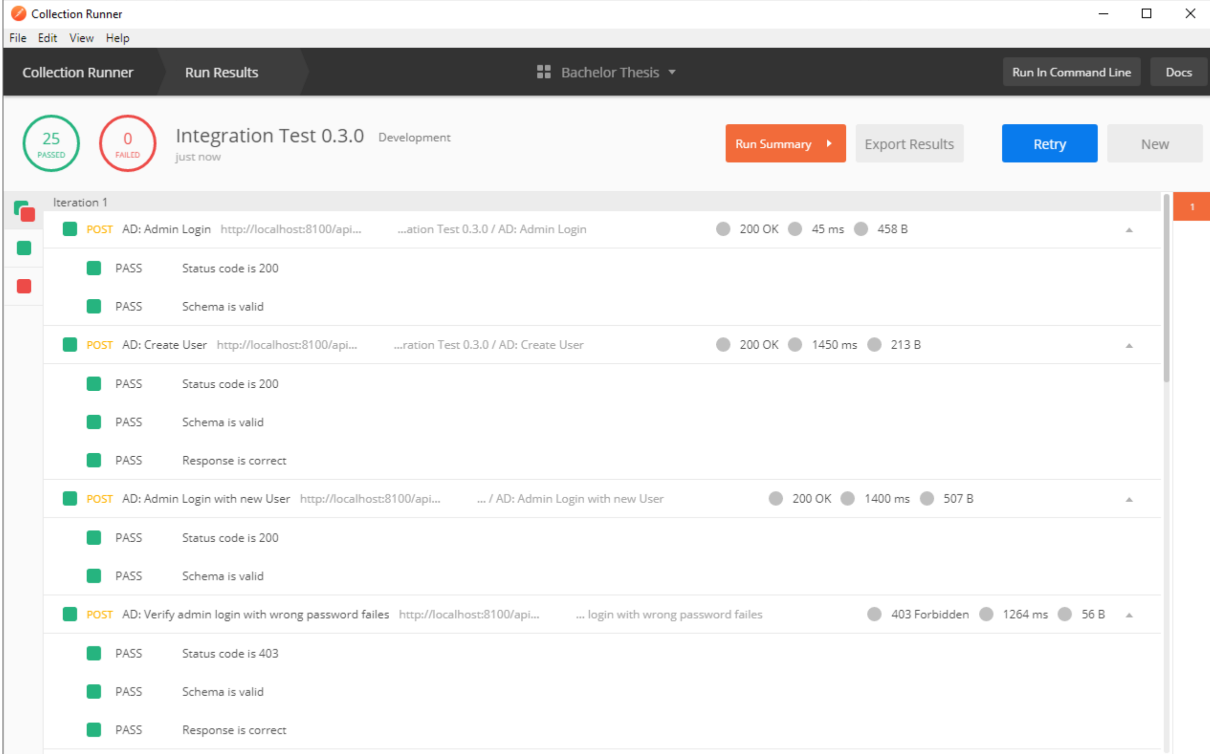


Abbildung 38 Postman Collection Runner Integration Test Resultat

Ausserdem können diese Integrationstests exportiert werden und mit Newman [40], der Kommandozeilenintegration von Postman, in der Konsole ausgeführt werden:

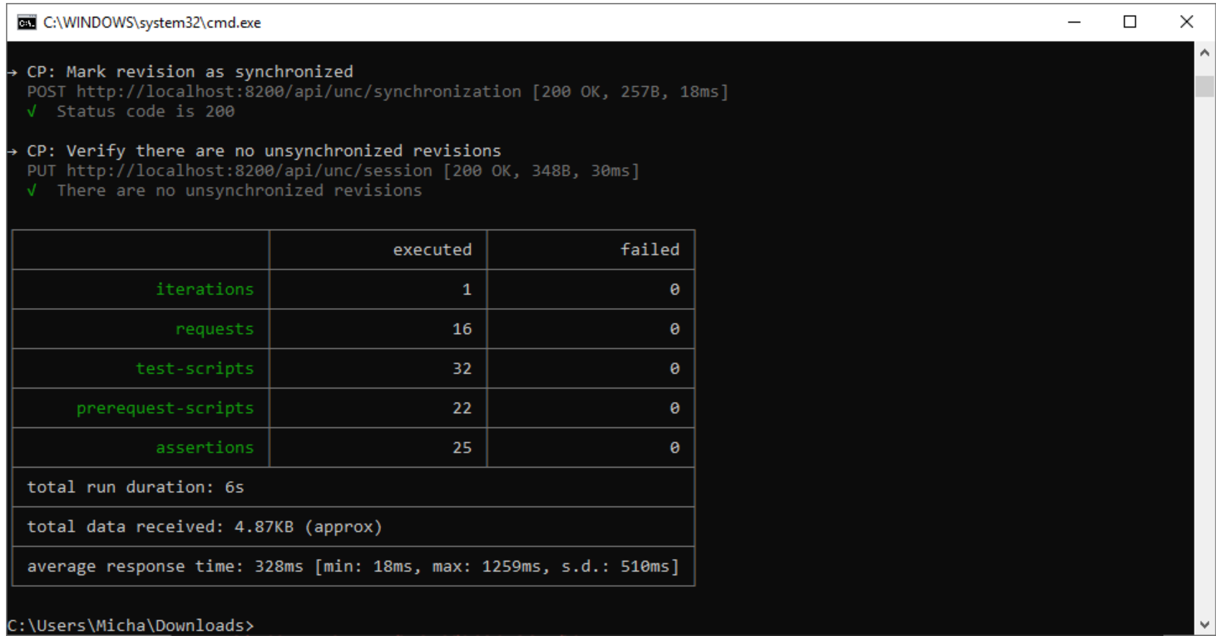


Abbildung 39 Integrationstests in der Konsole mit Newman

Das Durchführen der Integrationstests per Konsole ermöglicht es, die Integrationstests in einem weiteren Schritt in die CI-Pipeline einzubauen, um diese automatisiert auszuführen.

Erwähnenswert an dieser Stelle ist die Korrelation der Integrationstests auf die Events des Event Storming:

Nr.	Integrationstest	Event Storming Event
1.	AD: Admin Login	(weggelassen)
2.	AD: Create User	(weggelassen)
3.	AD: Admin Login with new User	(weggelassen)
4.	AD: Verify admin login with wrong password fails	(umgekehrter Test)
5.	AD: Create CareProvider	Erfasse Spital
6.	AD: Verify creating CareProvider without JWT fails	(umgekehrter Test)
7.	CP: Request CareProvider Token	Authentifiziere Spital
8.	CP: Verify there are no unsynchronized revisions	Sende Blutdruckdaten (Prozess)
9.	CP: Create Consumer	Erfasse Patient
10.	CP: Create HealthDataRequest	Gebe Blutdruckerfassung frei
11.	CP: Verify there are no unsynchronized revisions	Sende Blutdruckdaten (Prozess)
12.	PA: Submit HealthData	Sende Blutdruckdaten an Spital
13.	CP: Verify there is an unsynchronized revision	Sende Blutdruckdaten (Prozess)
14.	CP: Get unsynchronized submissions	Sende Blutdruckdaten (Prozess)
15.	CP: Mark revision as synchronized	Sende Blutdruckdaten (Prozess)
16.	CP: Verify there are no unsynchronized revisions	Sende Blutdruckdaten (Prozess)

Tabelle 13 Vergleich Integrationstests mit Events aus dem Event Storming

Es können 11 der 14 Integrationstests (ohne den zwei umgekehrten Tests) im Event Storming wiedergefunden werden. Die drei fehlenden Events wurden während des Event Stormings bewusst weggelassen, da entschieden wurde, dass die Admin-Interaktionen in dieser Analyse-Phase keine hohe Priorität hatten.

Es scheint also eine Abhängigkeit zwischen den Events aus dem Event Storming und den verschiedenen Schritten der Integrationstests zu geben. Dass so viele Events für die Tests verwendet wurden, lässt darauf schliessen, dass aus den Events direkt Integrationstests geschrieben werden können, was einen weiteren Nutzen des Event Storming aufzeigt. Nach dem vollständigen Umbau in den evaluierten Architekturvorschlag ist deshalb zu erwarten, dass jeder Event, der tatsächlich mit dem *Synchronization Server* interagiert, sich in einem Integrationstest wiederfinden lässt.

7.3 Review der nicht funktionalen Anforderungen

Nach dem prototypischen Umsetzen des evaluierten Architekturvorschlags wurde die Microservice-Architektur im Verbund getestet. Dieses Review bewertet, wie gut die Qualitätsmerkmale der nicht funktionalen Anforderungen (NFA), unter Einbezug der Resultate der Testprotokolle im Anhang, mit der umgesetzten Architektur erfüllt werden.

Qualitätsmerkmal	Beurteilung	Bemerkung
1. Ein QR-Code muss auch ausgedruckt und erfolgreich gescannt werden können, wenn im Spital keine Internet-Verbindung vorhanden ist.	unverändert	Ausserhalb des behandelten Source Codes (*CareProvider Client*)
2. Die Komponente beim Gesundheitsdienstleister kann auch benutzt werden, wenn keine Internet-Verbindung bei diesem vorhanden ist. Die getätigten Aktionen werden synchronisiert, sobald wieder eine Internet-Verbindung vorhanden ist.	unverändert	Ausserhalb des behandelten Source Codes (*CareProvider Client*)
3. Der Patient wird per Push-Benachrichtigung über neue Daten informiert.	unverändert	Ausserhalb des behandelten Source Codes (*CareProvider Client*)
4. Der Patient kann in 99.5 % der Zeit Daten an den Server senden und lesen.	erfüllt	Siehe 12.1.1Testprotokoll NFA 4 (100 %, 110000 von 110000 Anfragen) und Testprotokoll NFA 4 & 5 Loadtest (99.946 %, 180595 von 180692 Anfragen) im Anhang
5. Das System soll in den Spitzenzeiten von bis zu 10'000 Anfragen in der Sekunde in der Grösse von 50 KB 99.5 % dieser Anfragen abarbeiten können.	nicht erfüllt	Siehe Testprotokoll NFA 4 & 5 Loadtest im Anhang 99.946 %, 180595 von 180692 Anfragen mit ~180 Anfragen pro Sekunde
6. Die Komponente beim Gesundheitsdienstleister und der App des Patienten sollen sich nicht gegenseitig blockieren können.	nicht erfüllt	Wenn der *CareProvider* die zu Synchronisierenden Daten beim Microservice abholt und dies den Microservice blockiert, kann der Patient nicht mehr Gesundheitsdaten abliefern. Somit kann der Gesundheitsdienstleister den Patienten blockieren.
7. Die Applikation kann bei einem Gesundheitsdienstleister bereitgestellt und betrieben werden, ohne dass eine Konfiguration an der Firewall vorgenommen werden muss.	unverändert	Ausserhalb des behandelten Source Codes (*CareProvider Client*)

8.	Überall, wo ein Mensch mit der Applikation interagiert, darf die Reaktionszeit in 99.5 % der Fälle nicht grösser als 2 Sekunden sein.	erfüllt	Siehe Testprotokoll NFA 8 & 9 im Anhang 100 %, 9100 von 9100 Anfragen
9.	Überall, wo eine Maschine mit einer Maschine kommuniziert, darf die Reaktionszeit in 99.5 % der Fälle nicht grösser als 4 Sekunden sein.	erfüllt	Siehe Testprotokoll NFA 8 & 9 im Anhang 100 %, 9100 von 9100 Anfragen
10.	Das Senden einer Anfrage von einem Patienten zu einem Gesundheitsdienstleister oder umgekehrt, darf in 99.5 % der Fälle nicht länger als 20 Sekunden dauern, wenn beide Parteien eine aktive Internet-Verbindung haben.	erfüllt	Siehe Testprotokoll NFA 10 im Anhang 100 %, 480 von 480 Iterationen (4320 Anfragen)
11.	Es sollen nicht mehr als 4 besser 3 verschiedene Frameworks/Technologien (z. B. Spring, Angular, PostgreSQL) in der ganzen Applikation eingesetzt werden, um ein möglichst grosses Wissen auf diesen Technologien zu haben.	erfüllt	Spring, Angular, PostgreSQL, RabbitMQ
12.	Die Applikation weist eine Test-Abdeckung von mindestens 75 % vor, um neue Änderungen schnell einführen zu können.	unbekannt	Durch die Ablösung von Spring REST Docs Integrationstests zu Postman Integrationstests kann die Abdeckung nicht mehr gemessen werden.
13.	Die Dokumentation der API der Web-Server soll bis spätestens zwei Wochen nach dem Commit in den Masterbranch und erfolgreichem Testdurchlauf konsistent mit der effektiven Implementierung sein.	erfüllt	Die Postman Integrationstests können als API-Dokumentation veröffentlicht werden [41] oder sogar automatisiert werden. [42]
14.	Die ausgetauschten Daten zwischen einem Patienten und dem Gesundheitsdienstleister können nur auf der Komponente beim Gesundheitsdienstleister und auf dem Smartphone des Patienten gelesen werden. Die Daten sind mit einer kryptographischen Stärke von mindestens 128 Bit Ende-zu-Ende verschlüsselt.	unverändert	Ausserhalb des behandelten Source Codes (*Patient App* & *CareProvider Client*)

Tabelle 14 Review nicht funktionale Anforderungen

Das Review der nicht funktionalen Anforderungen (NFA) hat aufgezeigt, dass gewisse NFA ausserhalb des behandelten Codes dieser Arbeit sind, weshalb sie als Resultat mit unverändert bewertet sind. Für die übrigen NFA wurde, falls möglich, jeweils ein Testprotokoll erstellt und mit Postman Integrationstests oder Locust [43] Loadtests über die Dauer von 1h getestet.

Der Loadtest der NFA 5 hat ergeben, dass die erwarteten 10'000 Anfragen in der Sekunde vom Prototyp ohne Skalierung nicht erreicht werden. Beim Loadtest wurde im RabbitMQ-Dashboard festgestellt, dass die Anzahl Nachrichten in der Message Queue über die Zeit zunimmt, was darauf hindeutet, dass ein *Synchronization-Service* die Nachrichten nicht genügend schnell abarbeitet. Das horizontale Skalieren durch drei weitere *Synchronization-Service*-Instanzen liess die Queue abbauen, jedoch wurden schlussendlich nur ~180 Anfragen pro Sekunde erreicht. Ob diese 10'000 Anfragen pro Sekunden überhaupt erreicht werden können ist unklar, zumal aus dem Test nicht klar

hervorgeht, an welcher Stelle das Performanceproblem liegt. Mit vertikaler und horizontaler Skalierung kann die Kapazität der Microservices sicherlich erhöht werden.

Die NFA 6 wurde nicht erfüllt, da die Komponente beim Gesundheitsdienstleister und die App sich theoretisch noch gegenseitig blockieren können, da beide direkte Anfragen auf denselben Microservice durchführen. Durch das cloud-native Refactoring der ursprünglichen Applikation in Microservices mit Message Queues, hat für tiefere Kopplung zwischen *CareProvider Client* und *Patient App* geführt, jedoch wäre noch ein weiterer Umbau nötig, um diese NFA vollständig zu erfüllen.

Durch die Einführung der externen Postman Integrationstests und Ablösung der Spring REST Docs Integrationstests, kann die Testabdeckung nicht mehr gemessen werden, weshalb das NFA 12 als unbekannt beurteilt wird.

Im Vergleich zur vorgegebenen Applikation, welche die NFA 5 und 6 bereits nicht erfüllt, wurde eine Verbesserung der NFA 6 erreicht. Mit einer Weiterentwicklung der aktuellen Architektur ist es theoretisch möglich, diese NFA vollständig zu erfüllen. Zusammenfassend erfüllt die umgebaute Architektur die NFA befriedigend.

8 Ergebnisdiskussion

Das Resultat dieser Arbeit ist zum einen das dokumentierte Vorgehen, das auch für cloud-native Refactorings in anderen Branchen angewendet werden kann, zum anderen das cloud-native Refactoring der vorgegebenen Applikation.

8.1 Vorgehensweise

Die C4-Modellierung bewährte sich als pragmatisches Hilfsmittel bei der Dokumentation der bestehenden Applikation. Dies vor allem dadurch, dass auf jeder der vier Ebenen entschieden werden kann, in welchem Detaillierungsgrad die jeweiligen Komponenten oder Systeme dokumentiert werden sollen.

Die Workshop-Technik Event Storming eignete sich für die Modellierung der Domänen. Interessant ist die Kombination aus Requirement Engineering und Domänen Analyse, da beide Aufgaben in einem Workshop angegangen werden. Das volle Potential dieser Technik konnte jedoch nicht ausgeschöpft werden, da keine Domänen-Experten oder Domänen-Expertinnen am Workshop teilnahmen und die Informationen somit aus zweiter Hand kamen. Obwohl die Domänen-Grenzen nicht offensichtlich waren, war die Komplexität des behandelten Geschäftsprozesses etwas klein für diese Workshop-Technik.

Der Einsatz des Context Mapper und Service Cutter führte nicht zu einem hilfreichen Resultat. Könnte die CML-Modellierung aus dem Source Code generiert werden, könnte sich deren Einsatz lohnen.

Das entwickelte Vorgehen zur iterativen Abspaltung von Microservices aus dem Monolithen, hat sich als zielführend und effizient erwiesen. Es war im Fall der behandelten Applikation deutlich effizienter einen Microservice aus dem Monolithen herauszulösen, als diesen von Grund auf neu zu programmieren. So konnten zum Beispiel auch die Tests des Codes übernommen werden und mussten nur punktuell angepasst werden.

8.2 Umgesetzter Prototyp

Der Prototyp zeigt, dass es möglich ist, die behandelte Applikation zu einer Microservice-Architektur, aufgeteilt nach Bounded Contexts, umzubauen. Dieses Fallbeispiel zeigt jedoch auch, dass die Entscheidung für einen Microservice nicht zwingend mit einem Bounded Context begründet werden muss. Der *Synchronization-Service* wurde beispielsweise aus Performance-Gründen ausgelagert.

Der effektiv umgesetzte Prototyp ist in folgender Grafik visualisiert:

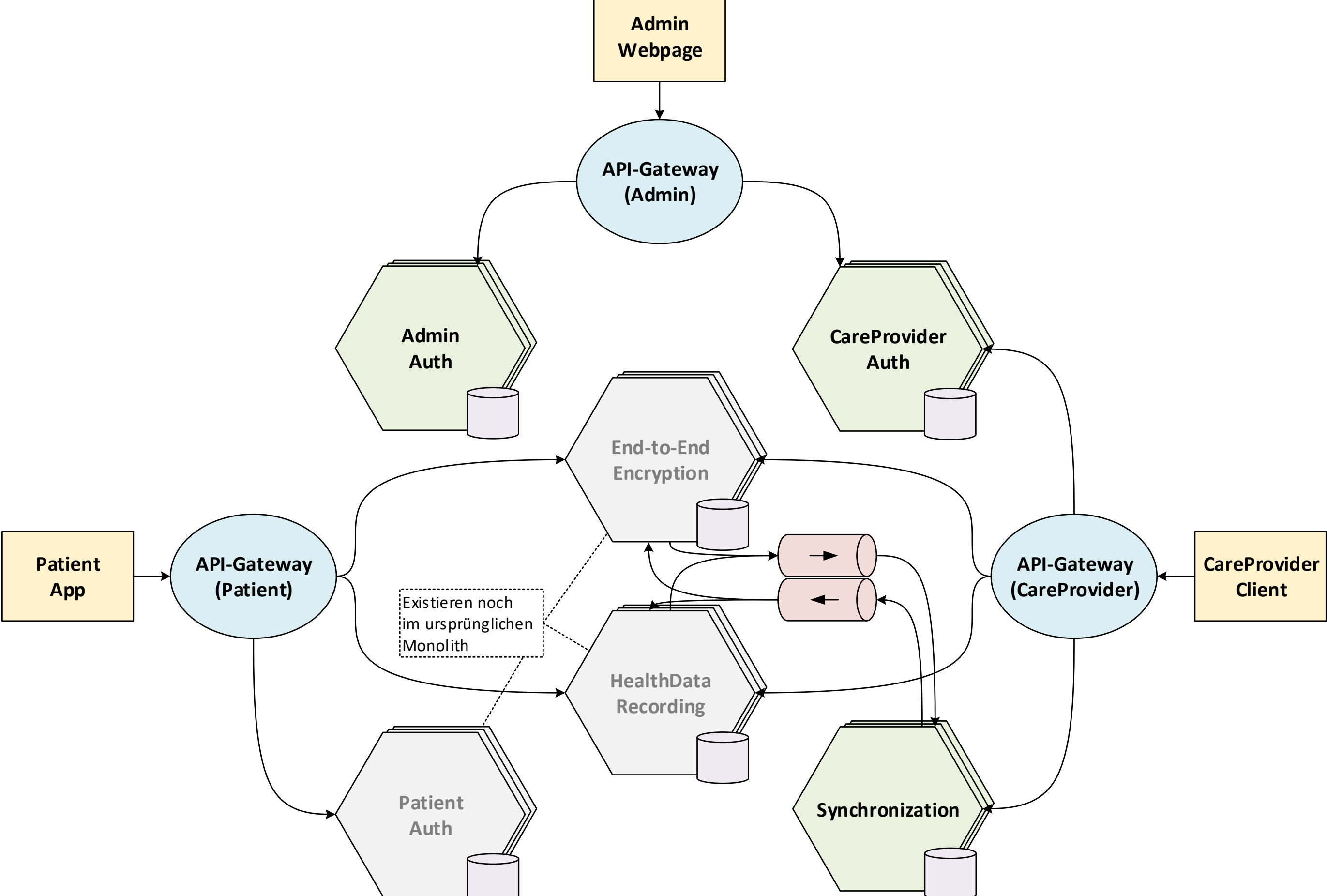


Abbildung 40 Architektur des umgesetzten Prototyps

In drei Iterationen wurde der *AdminAuth*, *CareProviderAuth* und *Synchronization-Service* ausgelagert. Da eine weitere Unterteilung des Monolithen in Microservices für die Diskussion keinen essentiellen Mehrwert geliefert hätte, wurde der Monolith nicht vollständig in den evaluierten Architekturvorschlag umgebaut. Dafür wurde die eingeplante Zeit für die Entkopplung der Microservices durch eine Message Queue eingesetzt.

8.2.1 Wirtschaftlichkeit

Beim entstandenen Prototyp ist mit einem höheren Betriebs- und Konfigurationsaufwand zu rechnen, da eine deutlich grössere Anzahl an Instanzen betrieben werden muss. Die Kompatibilität der Schnittstellen zwischen den Microservices kann nicht wie beim Monolithen zur Kompilierzeit festgestellt werden und muss mit aufwändigen Integrationstests sichergestellt werden. Folglich ist auch mit Mehraufwand bei der Entwicklungsumgebung für die Entwickler zu rechnen. Wenn jedoch mehrere Teams an einer solchen Applikation arbeiten, können sich die oben genannten Abstriche lohnen, da die Teams unabhängiger und somit effizienter arbeiten können. Für die behandelte Applikation ist dies jedoch nicht der Fall und eine Microservice-Architektur verschlechtert die Wirtschaftlichkeit, da der Umfang dieser Applikation gut von einem einzelnen Team verwaltet werden kann.

8.2.2 Stabilität

Ob die Stabilität der entstandenen Microservice-Architektur verbessert ist, kann nicht abschliessend beurteilt werden. Je unabhängiger die Microservices sind, desto autonomer können diese arbeiten. Mit der Aufteilung der Microservices nach den Bounded Context des Event Stormings wurden im

Prototyp mehrheitlich autonome Microservices erreicht. Diese erlauben somit den Ausfall eines Microservices, ohne dass die restlichen Microservices betroffen sind, was die Stabilität erhöht. Andererseits führt die verteilte Bereitstellung und zusätzlichen Technologien zu einer komplexeren Bereitstellung, die fehleranfälliger ist und somit die Stabilität negativ beeinflussen kann. Ein Team, das Erfahrung im Betrieb von Microservices hat, ist jedoch vermutlich in der Lage eine höhere Stabilität mit dieser Architektur zu erreichen. Falls die Ressourcen und Erfahrung für diese Aufgabe nicht vorhanden sind, wird die Stabilität nicht erhöht oder sogar negativ beeinflusst.

8.2.3 Flexibilität

Auch die Frage, ob die entstandene Architektur flexibler ist, kann nicht eindeutig beantwortet werden. Eine Microservice-Architektur, die schlechte Bounded Context für die Microservices gewählt hat, ist weniger flexibel als ein Monolith. In einer solchen Applikation betrifft eine neue Funktionalität mehrere Microservices, was deutlich mehr Aufwand bedeutet. Wenn die Bounded Context für die Microservices jedoch gut gewählt sind, unabhängig von verschiedenen Teams weiterentwickelt werden können und ihre Funktionalität hinter einer soliden API gekapselt sind, dann wird durchaus an Flexibilität gewonnen.

Die geplanten Microservices konnten im Prototyp mit einem hohen Grad an Autonomie implementiert werden, was darauf hinweist, dass die Bounded Contexts im Event Storming gut gewählt wurden und somit vermutlich auch für Erweiterungen eine flexible Basis bilden.

8.2.4 Adressierung der NFA 6

Bei der Betrachtung vom Review der NFA fällt die Nichterfüllung der architekturrelevanten NFA 6 auf. Diese verlangt, dass sich Gesundheitsdienstleister und Patienten gegenseitig nicht blockieren können. Obschon die Aufteilung der Microservices anhand der Bounded Context gut funktioniert hat und der Schaden bei einer Blockierung auf einen Microservice beschränkt wird, erfüllt die entstandene Architektur die NFA 6 nicht vollständig. Um diese NFA komplett zu erfüllen, müssten die Microservices, die mit der *Patient App* kommunizieren, vollständig von den Microservices, die mit dem Gesundheitsdienstleister kommunizieren, getrennt werden. Das kann durch den Einsatz einer Message Queue zwischen den entsprechenden Microservices erreicht werden. An dieser Stelle sollte jedoch abgeklärt werden, wie wichtig NFA 6 tatsächlich ist und ob es die Kosten und die verlangte Komplexität der Architektur wert ist.

8.3 Empfehlung

Aufgrund der oben aufgeführten Punkte wird für die behandelte Applikation von einem Umbau zu einer Microservice-Architektur abgeraten. Ein solcher wird sich erst lohnen, wenn mehrere Entwickler-Teams an dieser Applikation arbeiten. Deshalb empfiehlt es sich bereits jetzt, Vorkehrungen zu treffen, die ein cloud-native Refactoring in der Zukunft erleichtern. Die Learnings in den Implementierungsdetails können solche Vorkehrungen sein.

9 Ausblick

Das erarbeitete Vorgehen gilt es auf andere monolithische Applikationen in unterschiedlichen Branchen anzuwenden, um herauszufinden, ob dieses cloud-native Refactoring-Vorgehen dort ebenfalls erfolgreich eingesetzt werden kann.

In der prototypischen Umsetzung der Microservice-Architektur wurden nicht alle evaluierten Microservices aus dem Monolithen herausgelöst. Dies betrifft die Microservices *Patient Auth*, *HealthDataRecording* und *End-to-End-Encryption*, welche mit weiteren Iterationen herausgelöst werden können. Ebenfalls kann mit einer Erweiterung der Architektur mit zusätzlichen Message Queues eine Architektur erreicht werden, bei dem sich der *CareProvider* und der *Patient* nicht mehr gegenseitig blockieren können. Die Aufteilung der Microservices orientiert sich bei einer solchen Architektur nicht nur nach Domänen, sondern auch nach technischen Anforderungen.

Beim Einsatz der umgesetzten Architektur müsste die Continuous Integration (CI) erweitert werden, damit die Microservices im Verbund, mit den automatisierten Postman Integrationstests, getestet werden können. Auch müsste sorgfältig abgeklärt werden, wie API-Changes eingeführt werden oder Datenbank-Backups stattfinden. Denn im laufenden Betrieb der Microservice-Architektur stellen diese Aufgaben eine Herausforderung dar, da die Microservices nicht alle gleichzeitig abgeschaltet werden können, ohne dafür eine Ausfallzeit in Kauf zu nehmen.

In dieser Arbeit wurde ein Event Storming eingesetzt, um die Bounded Context zu identifizieren. Vernon [9] erwähnt neben dieser Workshop-Technik Event Sourcing als weitere Methode im Kontext von Domain-Driven Design. Interessant wäre ein Vergleich dieser beiden Methoden in einem cloud-native Refactoring.

Der Einsatz des Context Mapper, unter Einbezug des Service Cutter, zur automatischen Identifizierung von Microservices, wäre gerade für das Refactoring von grösseren Monolithen ein vielversprechender Ansatz. Damit diese Tools jedoch effizienter eingesetzt werden können, müsste das CML-Modell, das als Input für den Service Cutter vorausgesetzt wird, aus dem bestehenden Source Code generiert werden. Ein solches Programm könnte ähnlich aufgebaut sein wie Structurizer for Java [44], das automatisch ein C4-Modell aus dem Java Source Code generiert. In diesem Zusammenhang könnte auch untersucht werden, ob eine Konvertierung vom Output von Structurizer for Java zu CML realisierbar wäre.

10 Glossar

Begriffe, die mit einem Stern (*) markiert sind, wurden aus dem Glossar der vorhergegangen Studienarbeit [45] übernommen.

Begriff	**Beschreibung**
Aggregate	Wird im Kontext von DDD als Konsistenzgrenze von mehreren Entitäten verwendet. [12]
Angular	Ist ein Front-End-Webapplikationsframework. [7]
API	Abkürzung für Programmierschnittstelle, ermöglicht die Interaktion zwischen Systemen.
API-Gateway	Zentraler Einstiegspunkt, welcher die Microservices mit einem anderen Netzwerk, zum Beispiel dem Internet, verbindet, Anfragen weiterleitet, als Loadbalancer fungiert und Protokoll-Übersetzungen vornimmt. Auch Edge Service genannt. [17]
Architecturally Significant Requirements (ASR)	Anforderungen, die einen messbaren Einfluss auf die Architektur haben. [27]
AsciiDoc	Auszeichnungssprache die in verschiedene Dokumentformate konvertiert werden kann. [36]
autonomer Microservice	Microservice, der keine Abhängigkeiten zu anderen Microservices hat und Anfragen selbstständig beantworten kann.
Bereitstellung	Installation einer Software zum Beispiel auf einem Server.
Bounded Context	Im Kontext von DDD wird die Applikation in mehrere Bereiche, sogenannte Bounded Context, unterteilt. Bounded Context können in Microservices abgebildet werden.
C4-Modellierung	Eine Modellierungstechnik, die die Dokumentation eines Systems auf vier Abstraktionsstufen ermöglicht. [2]
CAP-Theorem	Besagt, dass Konsistenz (C), Verfügbarkeit (A) und Partitionstoleranz (P) von einem verteilten System nicht gleichzeitig erreicht werden können. [23]
CI/CD-Pipeline	Eine automatisch ausgeführte Abfolge von Aktionen, zum Beispiel das Testen von Source Code oder die Bereitstellung einer Applikation in einer Testumgebung, meistens ausgelöst durch eine Änderung im Source-Code-Verwaltungssystem.
Class-Responsibility-Collaboration-Karten (CRC-Karte)	Kommt ursprünglich aus dem objektorientierten Design weshalb *Class* im Namen vorkommt. [3] Wird in diesem Bericht verwendet, um die Zuständigkeiten und Zusammenarbeit von Komponenten zu beschreiben.
cloud-native Refactoring	Umbau einer Applikation zu einer Architektur, die in einer Cloud-Infrastruktur betrieben und skaliert werden kann.
Command	Im Kontext von Event Storming die auslösende Aktion, die zu einem Ereignis führt.
Content Based Router	Ein Enterprise Integration Pattern, das Nachrichten anhand von ihrem Inhalt an genau einen Empfänger weiterleitet. [18]
Context Mapper	Eine Open-Source-Software mit einer DSL für Context Mapping und Service Decomposition. [10]
Context Mapper Domain Specific Language (CML)	Eine DSL, für die Modellierung von strategic DDD-Patterns. [11]
Coupling Criterias (CC)	Sind Bestandteil der Open-Source-Software Service Cutter und beschreiben warum zwei Entitäten vom selben Service verwaltet werden sollen oder nicht. [14]
Datenbankcluster	Ein Zusammenschluss von mehreren Datenbanken, um zum Beispiel eine höhere Verfügbarkeit oder Kapazität zu erreichen.

DevOps*	DevOps (Development und IT Operations) beschreibt die Tätigkeit Softwareentwicklung, Konfigurieren und Unterhalten von Systemen.
Docker*	Mit Docker können virtuelle Umgebungen erstellt, geteilt und betrieben werden.
Domain Event	Im Kontext von Event Storming ein Ereignis, das innerhalb von einem Geschäftsbereich der Applikation auftritt.
Domain-Driven Design (DDD)	Propagiert die Modellierung von komplexer Software nach den fachlichen Domänen. [12]
Domäne	Ein autonomer Bereich oder ein Konzept innerhalb einer Applikation.
Endpunkt	Ein Teil einer API, der für eine spezifische Aufgabe verantwortlich ist.
Enterprise Integration Patterns (EIP)	Patterns zur Integration von heterogenen Systemen. [46]
Entity	Im Kontext von DDD eine Entität, die einen eigenen Lebenszyklus hat.
Event Storming	Workshop-Technik zur Identifizierung von Anforderungen und Bounded Contexts. [9]
Gesundheitsdienstleister	Eine Person, der es aufgrund der gesetzlichen Grundlage erlaubt ist, Gesundheitsdienstleitungen zu erbringen.
GitLab	Source-Code-Verwaltungssystem basierend auf Git. [47]
HTTP-Anfragen	Anfragen mit dem Hypertext Transfer Protocol.
Integrationstest	Testet den Verbund und das korrekte Zusammenspiel von mehreren Systemen.
JSON	Abkürzung für JavaScript Object Notation, ist ein kompaktes Format, das vielfach in HTTP-Anfragen verwendet wird.
JSON Web Token (JWT)	Ein Zugriffs-Token, das in RFC-7519 genormt ist. [24]
Kubernetes	Kubernetes ist ein Container-Orchestrierungs-Programm, das auf Docker-Containern basiert. [30]
Lastenverteilung	Verteilung von Anfragen auf verschiedene Instanzen einer Applikation.
Locust	Eine Open-Source-Software das Loadtests (Lastentests) ausführt. [43]
Loose Coupling	Diese Eigenschaft besagt, dass ein System wenig Abhängigkeiten zu anderen Systemen hat.
Message Queue	Eine Nachrichten-Warteschlagen, die sequenziell abgearbeitet wird.
mHealth	Der Einsatz von mobilen Geräten zur Unterstützung der Gesundheitsfürsorge. [48]
Microservice Tenet	Beschreiben Best Practices der Community in Bezug auf Microservices. [5]
Microservice*	Ein Microservice ist ein unabhängiger Teil einer komplexen Anwendungssoftware und erledigt eine kleine Aufgabe. Dies ermöglich modularen Aufbau einer Anwendungssoftware.
Monolith	Eine Applikation, welche die komplette Applikationslogik in einer einzigen Instanz beinhaltet und somit ein homogenes Gebilde aus funktionalen Elementen darstellt.
Newman	Konsolen-Programm zur Ausführung von HTTP-Anfragen, die mit Postman geschrieben wurden. [40]
Patientengut	Die Patienten und Patientinnen, die von einem Gesundheitsdienstleister betreut werden.

Polyglot Persistence	Beschreibt eine Eigenschaft von Microservices, die es erlaubt in den verschiedenen Microservices eine andere Technologie zur Persistenz von Daten einzusetzen.
Polyglot Programming	Beschreibt eine Eigenschaft von Microservices, die es erlaubt die verschiedenen Microservices in unterschiedlichen Programmiersprachen zu programmieren.
PostgreSQL	Eine Open-Source-Software, die eine relationale Datenbank zur Verfügung stellt, die sich am SQL-Standard orientiert. [8]
Postman	Eine Software, mit der unter anderem HTTP-Anfragen simulieren werden können. [37]
RabbitMQ	Eine Open-Source-Software die eine Message Queue implementiert. [20]
Requirement Engineering	Der Prozess zur Erfassung und Verwaltung von Anforderungen.
Seperation of Concerns (SoC)	Ein Pattern, das die Trennung von Anliegen innerhalb einer Software verlangt. [31]
Serverless	Ein Bereitstellungs-Modell für Software, bei dem keine fixen Server-Ressourcen reserviert werden, sondern die Kapazität automatisch der Nachfrage angepasst wird. [49]
Service	Im Kontext dieses Berichts ein Synonym für Microservice.
Service Cutter	Eine Open-Source-Software, die einen Monolithen anhand von Algorithmen in mögliche Microservices aufteilt. [13]
Single Point of Failure	Ein Teil in einem System, dessen Ausfall den Ausfall des ganzen Systems nach sich zieht. [50]
Smart Endpoints and Dumb Pipes	Message Queues, welche die Nachrichten nur transportieren und nicht transformieren oder routen. [25]
Spring Boot	Ermöglicht eine vereinfachte Konfiguration von einem Projekt mit dem Spring-Framework. [1]
Spring Cloud Gateway	Implementierung von einem API-Gateway im Ökosystem des Spring-Framework. [34]
Spring REST Docs	Generiert eine API-Dokumentation in AsciiDoc anhand von Integrationstests. [35]
Spring-Framework	Ein Java-Framework das häufig zur Programmierung von Webservern eingesetzt wird. [33]
stateless Authentication	Ein Ansatz, bei dem die Benutzer-Sitzung auf der Client-Seite verwaltet wird und der Server über diese Sitzung keine Kontrolle hat, deren Gültigkeit jedoch anhand einer elektronischen Signatur überprüfen kann.
Synchronization Server	Im Kontext dieses Berichtes die behandelte monolithische Applikation.
Unit-Test	Überprüft die korrekte Funktionsweise eines Moduls einer Applikation.
Web API	Die Programmschnittstelle eines Webservers, die die Interaktion mit einem Webbrowser ermöglicht.
Web Applikation	Eine Applikation, die in einem Web-Browser betrieben wird.
Webserver	Ein Server, der Inhalte für Webseiten oder Web Applikationen zur Verfügung stellt.

Tabelle 15 Glossar

11 Quellen/Referenzverzeichnis

11.1 Tabellenverzeichnis

Tabelle 1 Vorgehen cloud-native Refactoring 9
Tabelle 2 Class-Responsibility-Collaboration-Karte (CRC-Karte) - Synchronization Server 14
Tabelle 3 CRC-Karte - DataRequest Component 16
Tabelle 4 CRC-Karte - CareProvider Component 17
Tabelle 5 CRC-Karte - Synchronization Component 17
Tabelle 6 Nicht funktionale Anforderungen 18
Tabelle 7 Event Storming Klebezettel Notation 19
Tabelle 8 Resultat Longlist-Architekturvarianten 35
Tabelle 9 Vergleich Event Storming mit Microservice-Architekturvorschlag 1 (Ausgelagerte Synchronisation) 37
Tabelle 10 Vergleich der Architekturvorschläge anhand NFA 40
Tabelle 11 Vergleich der Architekturvorschläge anhand der Seven Microservice Tenets 42
Tabelle 12 Vergleich des Aufwands der Architekturvorschläge 43
Tabelle 13 Vergleich Integrationstests mit Events aus dem Event Storming 50
Tabelle 14 Review nicht funktionale Anforderungen 52
Tabelle 15 Glossar 60

11.2 Abbildungsverzeichnis

Abbildung 1 Umgesetzte Architektur des Prototyps 3
Abbildung 2 Legende: Vorgehen in einem cloud-native Refactoring 8
Abbildung 3 Vorgehen in einem cloud-native Refactoring 8
Abbildung 4 System Context Diagram (structurizr) - Synchronization Server 11
Abbildung 5 Container Diagram (structurizr) - Synchronization Server 13
Abbildung 6 Component Diagram (structurizr) - Synchronization Server 15
Abbildung 7 Event Storming Schritt 1 - Erfassung der Events 19
Abbildung 8 Event Storming Schritt 2 - Erfassung der Commands Teil 1 20
Abbildung 9 Event Storming Schritt 2 - Erfassung der Commands Teil 2 20
Abbildung 10 Event Storming Schritt 3 - Erfassung der Entitäten Teil 1 21
Abbildung 11 Event Storming Schritt 3 - Erfassung der Entitäten Teil 2 21
Abbildung 12 Event Storming Schritt 4 - Erstellen der Bounded Context 22
Abbildung 13 Use Case Diagramm funktionale Anforderungen 24
Abbildung 14 Beispiel Use Case RegisterCareProvider, modelliert mit der CML des Context Mapper 25
Abbildung 15 Service Cutter Resultat 26
Abbildung 16 Architekturvorschläge Legende 28
Abbildung 17 Architekturvariante Monolith 29
Abbildung 18 Architekturvariante Single-Gateway-Monolith 29
Abbildung 19 Architekturvariante Multi-Gateway-Monolith 30
Abbildung 20 Architekturvariante Monolith-Per-Careprovider 30
Abbildung 21 Architekturvariante Microservice-Direct-Invocation 31
Abbildung 22 Architekturvariante Single-Gateway-Multi-Microservice 32
Abbildung 23 Architekturvariante Multi-Gateway-Multi-Microservice 32
Abbildung 24 Architekturvariante Gateway-Queue-Microservice 33
Abbildung 25 Architekturvariante Serverless-Direct-Database 34
Abbildung 26 Microservice-Architekturvorschlag 1 (Ausgelagerte Synchronisation) 36
Abbildung 27 Microservice-Architekturvorschlag 2 (Entkoppelte Synchronisation) 38

Abbildung 28 Microservice-Architekturvorschlag 3 (CareProvider Queues) 39
Abbildung 29 Refactoring: Ausgangslage 44
Abbildung 30 Refactoring: API-Gateway hinzufügen 44
Abbildung 31 Refactoring: Monolith duplizieren 45
Abbildung 32 Refactoring: Monolith-Duplikat auf Microservice reduzieren 45
Abbildung 33 Refactoring: Auftrennen der Datenbank 46
Abbildung 34 Refactoring: Asynchrone Microservices 47
Abbildung 35 Refactoring: Synchrone Microservices 47
Abbildung 36 GitLab CI Pipeline Ergebnis 48
Abbildung 37 Integrationstest mit Postman 48
Abbildung 38 Postman Collection Runner Integration Test Resultat 49
Abbildung 39 Integrationstests in der Konsole mit Newman 50
Abbildung 40 Architektur des umgesetzten Prototyps 55
Abbildung 41 Testergebnis Newman NFA 4 66
Abbildung 42 LOCUST Loadtest Resultat 67
Abbildung 43 RabbitMQ-Dashboard: steigende Anzahl zwischengespeicherter Anfragen während LOCUST Loadtest 68
Abbildung 44 RabbitMQ-Dashboard: abnehmende Anzahl zwischengespeicherter Anfragen während LOCUST Loadtest 69
Abbildung 45 Testergebnis Newman NFA 8 & 9 70
Abbildung 46 Testergebnis Newman NFA 10 71

11.3 Literaturverzeichnis

[1] Spring Boot - Pivotal, https://spring.io/projects/spring-boot, letzter Zugriff am 10.06.2019

[2] The C4 model for visualising software architecture, https://c4model.com/, letzter Zugriff am 10.06.2019

[3] Class-Responsibility-Collaboration-Karten, https://de.wikipedia.org/w/index.php?oldid=129246975, letzter Zugriff am 10.06.2019

[4] Event storming - Wikipedia, https://en.wikipedia.org/w/index.php?oldid=899320952, letzter Zugriff am 10.06.2019

[5] O. Zimmermann, "Microservices tenets", Agile approach to service development and deployment, 2016, http://rdcu.be/mJPz, letzter Zugriff am 10.06.2019

[6] Structurizr, https://www.structurizr.com/, letzter Zugriff am 10.06.2019

[7] Angular - Wikipedia, https://de.wikipedia.org/w/index.php?oldid=189352105, letzter Zugriff am 10.06.2019

[8] PostgreSQL - The world's most advanced open source database, https://www.postgresql.org/, letzter Zugriff am 10.06.2019

[9] V. Vernon, "Domain-Driven Design kompakt", 2017, Übersetzt von C. Lilienthal, H. Schwentner

[10] Context Mapper, https://contextmapper.github.io/, letzter Zugriff am 10.06.2019

[11] CML Reference, https://contextmapper.github.io/docs/language-reference/, letzter Zugriff am 10.06.2019

[12] E. Evans, "Domain-Driven Design Reference", Definitions and Pattern Summaries, 2015, http://domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf, letzter Zugriff am 10.06.2019

[13] Service Cutter - Github, https://servicecutter.github.io/, letzter Zugriff am 10.06.2019

[14] Coupling Criteria - Service Cutter, https://github.com/ServiceCutter/ServiceCutter/wiki/Coupling-Criteria, letzter Zugriff am 10.06.2019

[15] I. X. Y. Leung, P. Hui, P. Lio, J. Crowcroft, "Towards real-time community detection in large networks", 2009, https://arxiv.org/pdf/0808.2633.pdf, letzter Zugriff am 10.06.2019

[16] M. Girvan, M. E. J. Newman, "Community structure in social and biological networks", 2001, https://arxiv.org/pdf/cond-mat/0112110.pdf, letzter Zugriff am 10.06.2019

[17] Backends For Frontends - Sam Newman, https://samnewman.io/patterns/architectural/bff/, letzter Zugriff am 10.06.2019

[18] Content-Based Router - Enterprise Integration Patterns, https://www.enterpriseintegrationpatterns.com/patterns/messaging/ContentBasedRouter.html, letzter Zugriff am 10.06.2019

[19] Guaranteed Delivery - Enterprise Integration Patterns, https://www.enterpriseintegrationpatterns.com/patterns/messaging/GuaranteedMessaging.html, letzter Zugriff am 10.06.2019

[20] Messaging that just works - RabbitMQ, https://www.rabbitmq.com/, letzter Zugriff am 10.06.2019

[21] Microservice Principles: Smart Endpoints and Dumb Pipes, https://medium.com/@nathankpeck/microservice-principles-smart-endpoints-and-dumb-pipes-5691d410700f, letzter Zugriff am 10.06.2019

[22] Function as a Service, https://de.wikipedia.org/w/index.php?oldid=184592394, letzter Zugriff am 10.06.2019

[23] CAP-Theorem - Wikipedia, https://de.wikipedia.org/w/index.php?oldid=186842318, letzter Zugriff am 10.06.2019

[24] JSON Web Tokens, https://jwt.io/, letzter Zugriff am 10.06.2019

[25] Smart Endpoints and Dumb Pipes, https://martinfowler.com/articles/microservices.html, letzter Zugriff am 10.06.2019

[26] O. Zimmermann, "Domain-Secific Service Decomposition with Microservice API Patterns", 2019, https://www.conf-micro.services/2019/slides/keynotes/Zimmerman.pdf, letzter Zugriff am 10.06.2019

[27] Architecturally significant requirements - Wikipedia, https://en.wikipedia.org/w/index.php?oldid=898730411, letzter Zugriff am 10.06.2019

[28] Twelve-Factor App methodology - Wikipedia, https://en.wikipedia.org/w/index.php?oldid=896298129, letzter Zugriff am 10.06.2019

[29] Docker - Enterprise Container Platform, https://www.docker.com/, letzter Zugriff am 10.06.2019

[30] Production-Grade Container Orchestration - Kubernetes, https://kubernetes.io/, letzter Zugriff am 10.06.2019

[31] Separation of concerns - Wikipedia, https://en.wikipedia.org/w/index.php?oldid=900724679, letzter Zugriff am 10.06.2019

[32] Microservices Authentication and Authorization Solutions, https://medium.com/tech-tajawal/microservice-authentication-and-authorization-solutions-e0e5e74b248a, letzter Zugriff am 10.06.2019

[33] Spring - Pivotal, https://spring.io/, letzter Zugriff am 10.06.2019

[34] Spring Cloud Gateway - Pivotal, https://spring.io/projects/spring-cloud-gateway, letzter Zugriff am 10.06.2019

[35] Spring REST Docs - Pivotal, https://spring.io/projects/spring-restdocs, letzter Zugriff am 10.06.2019

[36] AsciiDoc, https://de.wikipedia.org/w/index.php?oldid=185969574, letzter Zugriff am 10.06.2019

[37] API Development Environment - Postman, https://www.getpostman.com/, letzter Zugriff am 10.06.2019

[38] GitLab Continuous Integration & Delivery, https://about.gitlab.com/product/continuous-integration/, letzter Zugriff am 10.06.2019

[39] Introduction to CD with GitLab, https://docs.gitlab.com/ee/ci/introduction/, letzter Zugriff am 10.06.2019

[40] Command line integration with Newman - Postman Learning Center, https://learning.getpostman.com/docs/postman/collection_runs/command_line_integration_with_newman/, letzter Zugriff am 10.06.2019

[41] Intro to API documentation - Postman Learning Center, https://learning.getpostman.com/docs/postman/api_documentation/intro_to_api_documentation/, letzter Zugriff am 10.06.2019

[42] Creating html doc from postman collection - Stack Overflow, https://stackoverflow.com/questions/29238120/creating-html-doc-from-postman-collection, letzter Zugriff am 10.06.2019

[43] Locust - A modern load testing framework, https://locust.io/, letzter Zugriff am 10.06.2019

[44] Structurizr for Java - Github, https://github.com/structurizr/java, letzter Zugriff am 10.06.2019

[45] M. Habegger, M. Schena, "Architektur Prototyp einer Spring Cloud Applikation", Studienarbeit, 2018, https://eprints.hsr.ch/755/1/HS%202018%202019-SA-EP-Habegger-Schena-Architektur-Prototyp%20einer%20Spring%20Cloud%20Applikation.pdf, letzter Zugriff am 10.06.2019

[46] Enterprise Integration Patterns, https://www.enterpriseintegrationpatterns.com/, letzter Zugriff am 10.06.2019

[47] GitLab, https://about.gitlab.com/, letzter Zugriff am 10.06.2019

[48] mHealth - Wikipedia, https://de.wikipedia.org/w/index.php?oldid=188022580, letzter Zugriff am 10.06.2019

[49] Serverless computing - Wikipedia, https://en.wikipedia.org/w/index.php?oldid=899701772, letzter Zugriff am 10.06.2019

[50] Single point of failure - Wikipedia, https://en.wikipedia.org/w/index.php?oldid=898184625, letzter Zugriff am 10.06.2019

12 Anhänge

12.1 Testprotokolle NFA

12.1.1 Testprotokoll NFA 4

Zu testen:
NFA 4: Der Patient kann in 99.5 % der Zeit Daten an den Server senden und lesen.

Test:
Über eine Dauer von 1h werden Anfragen, wie vom Patienten gesendet, auf den Prototyp gesendet und 99.5 % dieser Anfragen sollen erfolgreich sein.

Testumgebung:
Datum: 26.05.19
Tester: Micha Schena
Synchronization Prototyp Commit: 985d6c14
Newman Version: 4.4.1
Testmaschine: Windows 10 Pro Version 1809 64-Bit, 16GB RAM, i7-8750H

Vorbereitung:
Für 10 000 Iteration des Tests braucht Newman 5 min 33.6 s. Damit der Test etwa eine Stunde dauert, braucht es deshalb 11 * 10 000 = 110 000 Iterationen.

Testlauf:
Command: newman run patient_test.postman_collection.json -n 110000

	executed	failed
iterations	110000	0
requests	110000	0
test-scripts	110000	0
prerequest-scripts	110000	0
assertions	110000	0
total run duration: 1h 40m 52s		
total data received: 0B (approx)		
average response time: 14ms [min: 10ms, max: 781ms, s.d.: 7ms]		

Abbildung 41 Testergebnis Newman NFA 4

Resultat:
Erfüllt, da 100 % (110000 von 110000) der Anfragen erfolgreich waren.

12.1.2 Testprotokoll NFA 4 & 5 Loadtest

Zu testen:
NFA 4: Der Patient kann in 99.5 % der Zeit Daten an den Server senden und lesen.
NFA 5: Das System soll in den Spitzenzeiten von bis zu 10'000 Anfragen in der Sekunde in der Grösse von 50 KB 99.5 % dieser Anfragen abarbeiten können.

Test:
Das System wird mit dem Programm Locust, einem Load-Testing-Tool das mehrere gleichzeitige Benutzer simulieren kann, getestet. Dies wurde gewählt, da es schnell aufgesetzt ist und eine gute Dokumentation besitzt, mit der die benötigten Python-Dateien zügig erstellt sind.

Beim Test werden Gesundheitsdaten vom Patienten an den *Synchronization Server* gesendet. Es werden so viele Anfragen wie möglich gesendet. Diese Anfrage wird an den Monolithen des Prototyps gesendet und veröffentlicht eine Message auf der Message Queue. Diese Messages werden vom *Synchronization-Service* abgearbeitet.

Testumgebung:
Datum: 26.05.19
Tester: Moritz Habegger
Synchronization Prototyp Commit: 985d6c14
Locust Version: 0.11.0
Testmaschine: macOS Mojave 10.14.4 (18E226) 64-Bit, 16GB RAM, i7-4870HQ

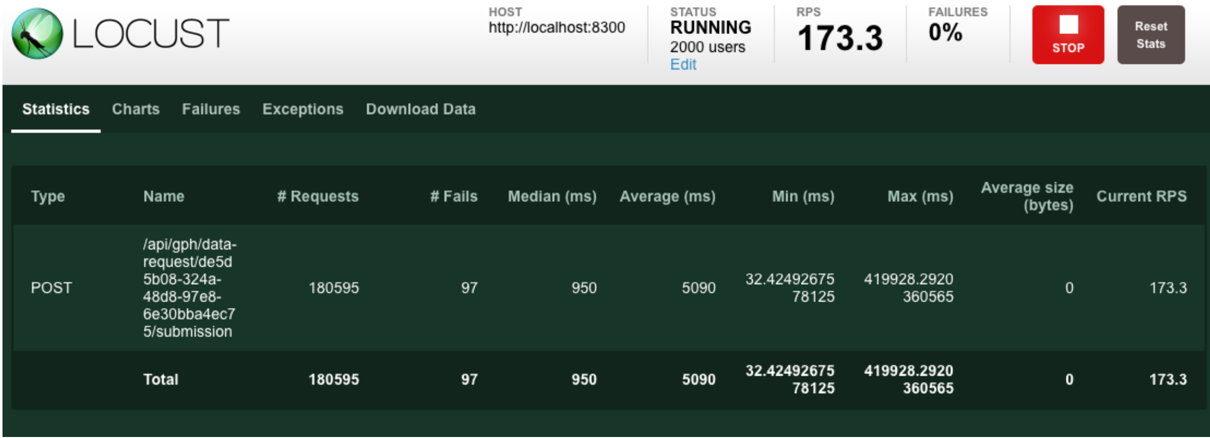
Vorbereitung:
Locust wurde auf 2000 User konfiguriert, was in ~180 Anfragen pro Sekunde resultiert. Eine höhere Konfiguration hat die Anzahl der Anfragen nicht erhöht. Die Microservices des *Synchronization Servers* wurden mit Docker bereitgestellt. Von jedem Microservice wurde eine Instanz bereitgestellt.

Es wurde manuell ein Patient und ein *HealthDataRequest* auf dem *Synchronization Server* angelegt. Die IDs dieser beiden Entitäten wurden manuell im Locust-Python-Script eingetragen.

Testlauf:
Command: locust -f locust.py --host=http://localhost:8300
(der Port 8300 entspricht dem *API-Gateway (Patient)*)

Resultat:

Der Loadtest ergab 180595 erfolgreiche und 97 fehlgeschlagene Anfragen. Das entspricht 0.0537 % fehlgeschlagener Anfragen. Das NFA 4 erlaubt eine Fehlerquote von 0.5 %, was somit erfüllt wurde.



Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
POST	/api/gph/data-request/de5d5b08-324a-48d8-97e8-6e30bba4ec75/submission	180595	97	950	5090	32.4249267578125	419928.2920360565	0	173.3
	Total	**180595**	**97**	**950**	**5090**	**32.4249267578125**	**419928.2920360565**	**0**	**173.3**

Abbildung 42 LOCUST Loadtest Resultat

Während des Tests wurde festgestellt, dass die veröffentlichten Events auf der Message Queue von einem *Synchronization-Service* nicht genügend schnell abgearbeitet werden konnten.

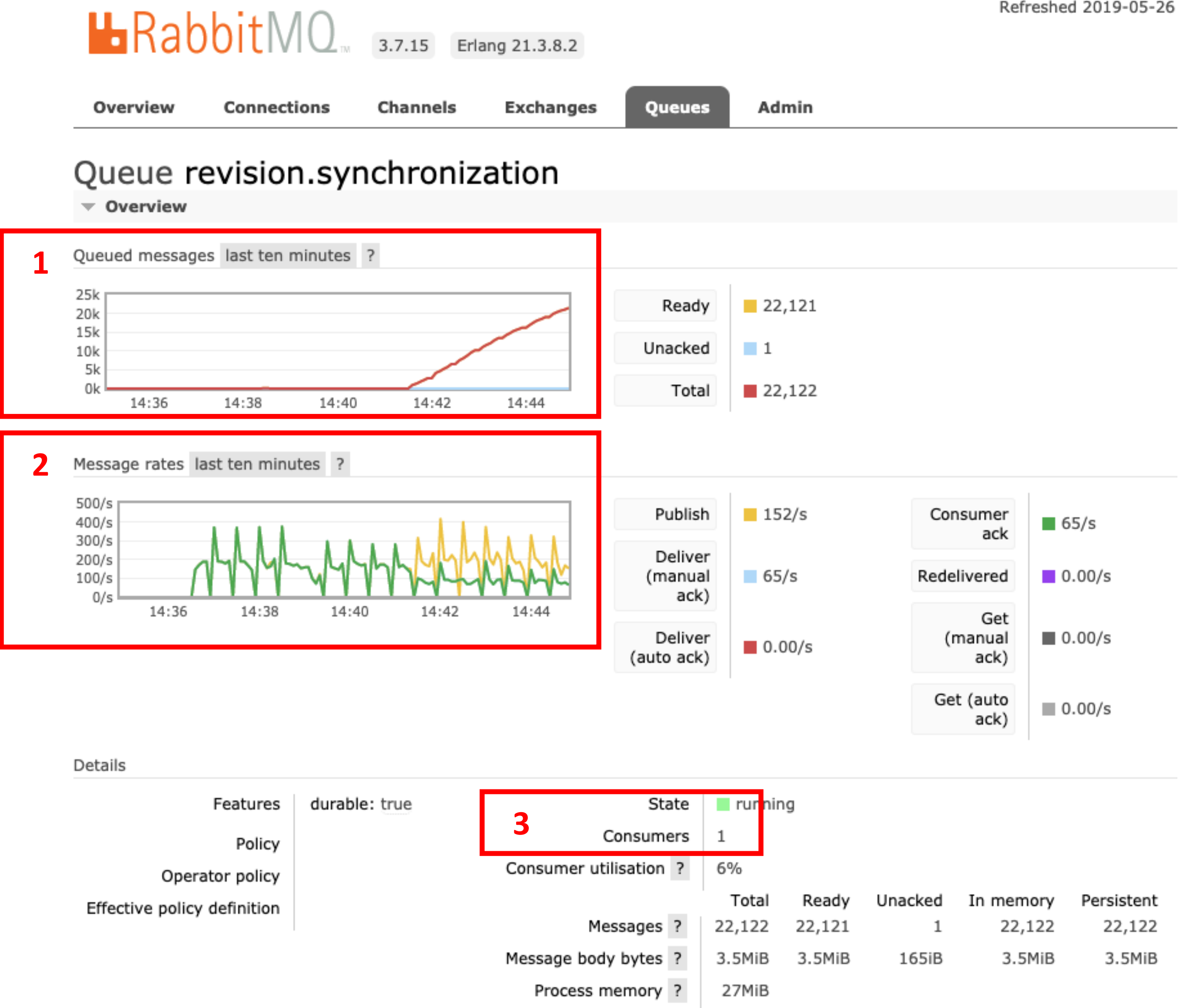


Abbildung 43 RabbitMQ-Dashboard: steigende Anzahl zwischengespeicherter Anfragen während LOCUST Loadtest

Auf dem RabbitMQ-Dashboard ist zu sehen, dass die Anzahl nicht abgearbeiteter Anfragen über die Zeit zunimmt (Markierung 1). Das hängt unter anderem damit zusammen, dass der *Synchronization-Service* die Nachrichten nicht genug schnell abarbeiten konnte. Das wird in der Markierung 2 ersichtlich, in der die gelbe Linie die neuen und die grüne die abgearbeiteten Nachrichten darstellt. Bei der Markierung 3 wird ersichtlich, dass nur ein Consumer, der *Synchronization-Service*, die Nachrichten abarbeitet.

Um diese Message Queue schneller abzuarbeiten, wurden 3 weitere Instanzen des *Synchronization-Service* gestartet.

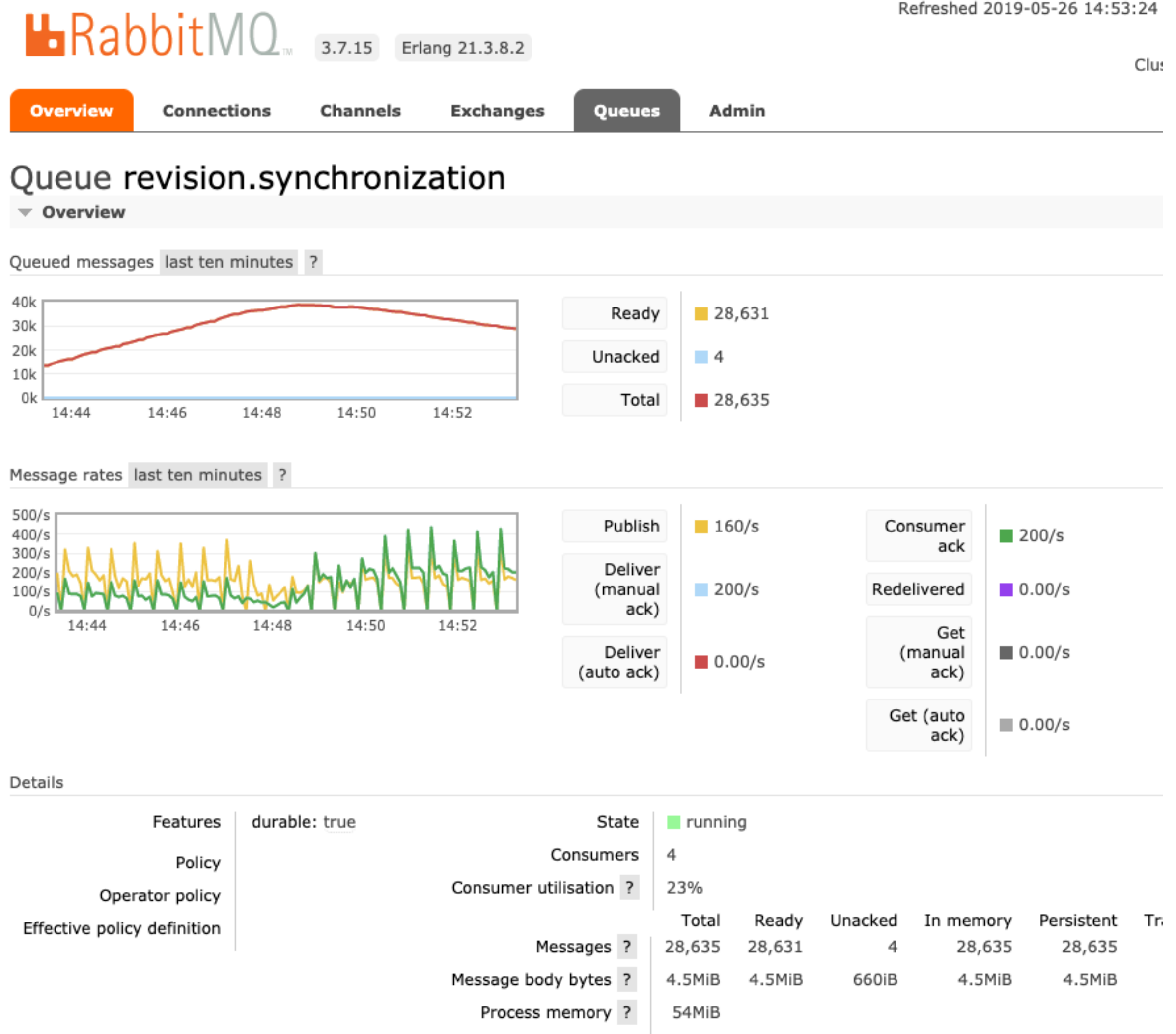


Abbildung 44 RabbitMQ-Dashboard: abnehmende Anzahl zwischengespeicherter Anfragen während LOCUST Loadtest

Durch das Hinzufügen der drei *Synchronization-Services* konnten die Nachrichten wieder abgebaut werden.

Die Anfragen wurden nicht weiter optimiert und somit ist die Erfüllung der NFA 5 mit 10'000 Anfragen pro Sekunde nicht erreicht. Erreicht wurden ~180 Anfragen pro Sekunde. Um NFA 5 zu erreichen, müsste der Test auf einem leistungsfähigen Cluster durchgeführt werden und verschiedene Optimierungen vorgenommen werden.

12.1.3 Testprotokoll NFA 8 & 9

Zu testen:
NFA 8: Überall, wo ein Mensch mit der Applikation interagiert, darf die Reaktionszeit in 99.5 % der Fälle nicht grösser als 2 Sekunden sein.
NFA 9: Überall, wo eine Maschine mit einer Maschine kommuniziert, darf die Reaktionszeit in 99.5 % der Fälle nicht grösser als 4 Sekunden sein.

Test:
Über eine Dauer von 1h werden Anfragen auf den Prototyp gesendet. Ein Teil dieser Anfragen sind Interaktionen vom Menschen mit der Applikation, die mit Testfall nicht länger als 2 Sekunden dauern dürfen und der andere Teil der Anfragen ist die Kommunikation zwischen Maschine und Maschine, die mit Testfall nicht länger als 4 Sekunden dauern darf.

Testumgebung:
Datum: 26.05.19
Tester: Micha Schena
Synchronization Prototyp Commit: 985d6c14
Newman Version: 4.4.1
Testmaschine: Windows 10 Pro Version 1809 64-Bit, 16GB RAM, i7-8750H

Vorbereitung:
Für 100 Iteration des Tests braucht Newman 8 min 56 s. Damit der Test etwas länger als eine Stunde dauert, braucht es deshalb 7 * 100 = 700 Iterationen.

Testlauf:
Command: newman run nfa_8_9_test.postman_collection.json -e development.postman_environment.json -n 700

	executed	failed
iterations	700	0
requests	9100	0
test-scripts	18200	0
prerequest-scripts	14000	0
assertions	18900	0
total run duration: 1h 3m 28.7s		
total data received: 2.72MB (approx)		
average response time: 81ms [min: 7ms, max: 4.1s, s.d.: 231ms]		

Abbildung 45 Testergebnis Newman NFA 8 & 9

Resultat:

Erfüllt, da 100 % (9100 von 9100) der Anfragen erfolgreich waren.

12.1.4 Testprotokoll NFA 10

Zu testen:
NFA 10: Das Senden einer Anfrage von einem Patienten zu einem Gesundheitsdienstleister oder umgekehrt, darf in 99.5 % der Fälle nicht länger als 20 Sekunden dauern, wenn beide Parteien eine aktive Internet-Verbindung haben.

Test:
Getestet wird nur die Richtung vom Patienten zum Gesundheitsdienstleister, da die andere Richtung ausserhalb des bearbeiteten Codes liegt.

Über eine Dauer von 1h werden Anfragen auf den Prototyp gesendet. Beim Senden von Gesundheitsdaten wird ein Zeitstempel gespeichert, nach dem Synchronisieren als Testfall verglichen und soll dabei weniger als 20 Sekunden alt sein.

Testumgebung:
Datum: 26.05.19
Tester: Micha Schena
Synchronization Prototyp Commit: 985d6c14
Newman Version: 4.4.1
Testmaschine: Windows 10 Pro Version 1809 64-Bit, 16GB RAM, i7-8750H

Vorbereitung:
Für 40 Iteration des Tests braucht Newman 5 min 27.5 s. Damit der Test etwas mehr als eine Stunde dauert, braucht es deshalb 12 * 40 = 480 Iterationen.

Testlauf:
Command: newman run nfa_10_test.postman_collection.json -e development.postman_environment.json -n 480

	executed	failed
iterations	480	0
requests	4320	0
test-scripts	8640	0
prerequest-scripts	7200	0
assertions	5760	0
total run duration: 1h 5m 6.9s		
total data received: 1.81MB (approx)		
average response time: 98ms [min: 8ms, max: 3.5s, s.d.: 262ms]		

Abbildung 46 Testergebnis Newman NFA 10

Resultat:
Erfüllt, da 100 % (480 von 480) der Iterationen erfolgreich waren.