

Frankfurt University of Applied Sciences  
Fachbereich 2: Informatik & Ingenieurwissenschaften

# Bachelorarbeit

im Studiengang Informatik

zur Erlangung des akademischen Grades  
Bachelor of Science

**Thema:** Microservices in der Integrationsdomäne: Evaluierung unterschiedlicher Zerlegungsmuster eines ESB-Monolithen

**Autor:** Sven Moj <sven\_moj@yahoo.de>  
MatNr. 1018567

**Version vom:** 21. August 2015

**1. Betreuer:** Prof. Dr. Christian Baun  
**2. Betreuer:** Prof. Dr. Thomas Gabel

## Eidesstattliche Erklärung

### Eidesstattliche Erklärung zur Bachelorarbeit

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

*Unterschrift :*

*Ort, Datum :*

## **Zusammenfassung**

Microservices sind ein aktuelles Thema in der Softwareentwicklung und sollen Probleme monolithischer Architekturen lösen. Ziel dieser Arbeit ist es, eine bestehende monolithische Anwendung in voneinander unabhängig bereitstellbare Services zu zerlegen. Dies wird anhand der Anwendungslandschaft des fiktiven Beispielunternehmens DFDFAK gezeigt, indem ein monolithischer Enterprise Service Bus in mehrere Microservices zerlegt wird. Die Ergebnisse dieser Untersuchung zeigen, dass eine Microservice-Architektur die auftretenden Probleme einer monolithischen Architektur adressieren kann.

## **Abstract**

Microservices are a current topic in software engineering and are supposed to address problems occurring with monolithic architectures. The goal of this thesis is to decompose a monolithic application into a set of independently deployable microservices. This is shown by decomposing the monolithic enterprise service bus of the fictional sample company DFDFAK. The result of this thesis shows that a microservice architecture can address the occurring problems of a monolithic architecture.

# Inhaltsverzeichnis

<b>Eidesstattliche Erklärung</b>	<b>2</b>
<b>Abbildungsverzeichnis</b>	<b>5</b>
<b>Tabellenverzeichnis</b>	<b>5</b>
<b>Listingverzeichnis</b>	<b>5</b>
<b>1 Einleitung</b>	<b>6</b>
1.1 Motivation . . . . .	6
1.2 Zielsetzung . . . . .	6
1.3 NovaTec Consulting GmbH . . . . .	6
1.4 Aufbau der Arbeit . . . . .	7
1.5 Danksagung . . . . .	7
<b>2 Eigenschaften einer monolithischen Architektur</b>	<b>8</b>
<b>3 Was sind Microservices?</b>	<b>10</b>
3.1 Vorteile einer Microservice-Architektur . . . . .	11
3.2 Nachteile einer Microservice-Architektur . . . . .	11
<b>4 Beschreibung des monolithischen Systems</b>	<b>13</b>
4.1 DFDFAK als akademisches Beispiels-Unternehmen zum Testen von In- tegrationstools . . . . .	13
4.1.1 DFDFAK Systeme . . . . .	15
4.1.2 Integrationsszenario . . . . .	17
4.1.3 Einführung in das Thema JMS . . . . .	18
4.2 Grundlagen zu Spring Integration . . . . .	19
4.3 Der Enterprise Service Bus (ESB) . . . . .	20
<b>5 Teilung des Monolithen in Microservices</b>	<b>23</b>
5.1 Der Skalierungswürfel . . . . .	23
5.1.1 X-Achsen Skalierung . . . . .	23
5.1.2 Y-Achsen Skalierung . . . . .	24
5.1.3 Z-Achsen Skalierung . . . . .	24
5.2 Teilungsmöglichkeiten der monolithischen Architektur . . . . .	24
<b>6 Praktischer Teil</b>	<b>31</b>
6.1 Umsetzung . . . . .	32
6.2 Testen der Architektur . . . . .	33
6.3 Vorteile der umgesetzten Architektur . . . . .	35
6.4 Nachteile der umgesetzten Architektur . . . . .	35
<b>7 Ausblick</b>	<b>37</b>
<b>8 Fazit</b>	<b>39</b>
<b>Literaturverzeichnis</b>	<b>40</b>

**Anhang****41****Abbildungsverzeichnis**

1	Anwendungslandschaft der DFDFAK . . . . .	13
2	FehlerFix Fehlerereignis Beispiel im fixed length Format . . . . .	16
3	ErrorPipe Fehlerereignis Beispiel im delimited Format . . . . .	16
4	DFDFAK Integrationsarchitektur . . . . .	18
5	JMS Queue (Quelle: URL: <a href="http://www.straub.as/java/jms/basic.html">http://www.straub.as/java/jms/basic.html</a> ) . . . . .	19
6	JMS Topic (Quelle: URL: <a href="http://www.straub.as/java/jms/basic.html">http://www.straub.as/java/jms/basic.html</a> ) . . . . .	19
7	Transportrouten der drei Anwendungen FehlerFix, XError und ErrorPipe . . . . .	22
8	Skalierungswürfel . . . . .	23
9	Monolithische Architektur . . . . .	25
10	Nachrichtenroute des ESB . . . . .	25
11	Zerlegung des ESB in Microservices . . . . .	26
12	Alternative Zerlegungsmöglichkeit des ESB . . . . .	26
13	Zerlegung nach Aufgabenbereichen . . . . .	27
14	Microservices mit bidirektionalen Message-Routen . . . . .	28
15	Microservices mit unidirektionalen Message-Routen . . . . .	29
16	Auf Fehlermonitor zugeschnittene Microservices . . . . .	29
17	Pipes and Filter . . . . .	31
18	Umgesetzte Architektur . . . . .	32
19	Weboberfläche von XError zum Senden der Test-Nachricht . . . . .	34
20	Graphische Oberfläche von FehlerFix mit der erhaltenen Fehlermeldung . . . . .	35

**Tabellenverzeichnis**

1	DFDFAK Altsysteme . . . . .	15
2	FehlerFix Datenstruktur . . . . .	15

**Listingverzeichnis**

1	XError-Fehlernachricht . . . . .	16
2	context.xml . . . . .	41
3	Properties-Datei . . . . .	43
4	App.java (Programmstart) . . . . .	44
5	Klasse für kanonisches Datenformat . . . . .	44
6	Klasse zum En-/Decodieren von Nachrichten . . . . .	45
7	Klasse für kanonisches Datenformat . . . . .	47
8	Klasse zum Auslesen von Inhalt aus JMS-Nachrichten . . . . .	49
9	Klasse zum umwandeln von kanonischem Datenformat in String . . . . .	49

# 1 Einleitung

In diesem Kapitel wird die Motivation der Arbeit, die Zielsetzung, die Firma NovaTec Consulting GmbH und der Aufbau der Arbeit vorgestellt.

## 1.1 Motivation

Der Begriff Microservices ist in den letzten Jahren aufgekommen und beschreibt einen Architekturstil der große, komplexe und langlebige Anwendungen aus einer Menge von zusammenhängenden Services ermöglichen soll. Dieser Architekturstil wurde unter Entwicklern stark diskutiert. Auch auf der Konferenz JAX 2015 [JAX15] waren Microservices ein viel diskutiertes Thema, welches aus verschiedensten Sichtweisen betrachtet wurde. Dabei wurden nicht nur die Vorteile einer Microservice-Architektur hervorgehoben, sondern auch die Komplexität, die mit der Einführung dieses Architekturstils in ein Enterprise-System aufkommt, deutlich gemacht. Die Fragestellung zu dieser Arbeit entstand bei der NovaTec Consulting GmbH, welche als Beratungsunternehmen ein großes Interesse an dem hochaktuellen Thema Microservices besitzt. Die Competence Group *Integration Architecture and Technology* beschäftigt sich hauptsächlich mit der Einsetzbarkeit verschiedenster Enterprise Integration Technologien. Für die Umstellung einer Unternehmens-Anwendungslandschaft auf einen Microservices-Architekturstil, ergeben sich prinzipiell zwei Möglichkeiten. Bestehende monolithische Anwendungen können entweder in mehrere Microservices zerlegt werden oder es wird von Anfang an eine neue Funktionalität erstellt. NovaTec ist in ihrem Beratungsauftrag daran interessiert, diesen neuen Architekturstil zu untersuchen und herauszufinden, ob dieser Stil einen guten Ersatz zu bereits bestehenden monolithischen Systemen darstellt.

## 1.2 Zielsetzung

Ziel der Arbeit ist es ein monolithisches System in eine Microservice-Architektur zu zerlegen. Hierfür wird die Anwendungslandschaft des akademischen Beispielunternehmens DFDFAK (Deutsche Firma die fast alles kann) verwendet, welche der NovaTec als fiktive Firmenstruktur zum Erwerb von Kenntnissen neuer Integrationsarchitekturen dient. Die Anwendungslandschaft der DFDFAK besteht aus drei Anwendungen, welche durch einen monolithischen Enterprise Service Bus miteinander verbunden sind. Ziel ist die Zerlegung des ESB und das Herausarbeiten der Vor- und Nachteile der Teilung eines monolithischen Systems.

## 1.3 NovaTec Consulting GmbH

Diese Thesis entstand bei und für die NovaTec Consulting GmbH. Die NovaTec Consulting GmbH ist ein Tochterunternehmen der NovaTec Holding GmbH und ist ein

IT-Beratungsunternehmen mit Hauptsitz in Leinfelden-Echterdingen nahe Stuttgart. Neben dem Hauptsitz ist NovaTec an vier weiteren Standorten vertreten: in Berlin, Frankfurt am Main, München und in Jeddah (Saudi Arabien). Mit ca. 125 Beratern werden Unternehmen verschiedenster Branchen bei der erfolgreichen Umsetzung von komplexen Software-Lösungen unterstützt. Das Resultat dieses Thesis ist die Zerlegung eines monolithischen Enterprise Service Bus in Microservices und soll für NovaTec die Vor- und Nachteile einer Microservice-Architektur erkenntlich machen. Zusätzlich soll diese Arbeit den Beratungsauftrag der NovaTec unterstützen.

## **1.4 Aufbau der Arbeit**

Im folgenden Kapitel wird der Begriff der monolithischen Architektur definiert und die Vor- und Nachteile einer solchen Architektur vermittelt. Im dritten Kapitel wird erläutert welche Eigenschaften eine Microservice-Architektur besitzt, welche Vor- und Nachteile sie mit sich bringt und es wird ein Vergleich zu der monolithischen Architektur gezogen. Danach wird der Aufbau des für die Arbeit zugrunde liegenden monolithischen Systems beschrieben. Dabei werden die benutzten Technologien näher erläutert. Im darauf folgenden Kapitel werden die Gedankengänge zur Teilung des Monolithen in eine Microservice-Architektur erläutert. Der praktische Teil, der eine konkrete Umsetzung einer Zerlegungsmöglichkeit der monolithischen Architektur umfasst und die Vor- und Nachteile dieser Zerlegungsmöglichkeit aufzeigt, wird in Kapitel 6 erläutert. Es folgt ein Ausblick auf die Möglichkeit, Microservices in Docker-Containern bereitzustellen. Dieser Themenbereich ist ebenfalls sehr komplex und konnte im Laufe der Arbeit nicht umgesetzt werden. Am Ende wird ein Fazit gezogen und die Ergebnisse der Arbeit werden noch einmal kurz zusammengefasst. Im Anhang befindet sich beispielhaft der Quellcode des FehlerFix-Microservice.

## **1.5 Danksagung**

Danken möchte ich vor allem Herrn Prof. Dr. Christian Baun für die Betreuung meiner Bachelorarbeit. Des Weiteren gilt mein Dank an meinen Betreuer von Seiten der NovaTec, Herrn Dr. Oliver Berger, für seine Unterstützung. Durch stetig kritisches Hinterfragen und konstruktiver Kritik verhalf er mir zu einer fundierten Arbeit.

## 2 Eigenschaften einer monolithischen Architektur

Im Allgemeinen wird durch eine Software-Architektur festgelegt, wie einzelne Komponenten einer Software miteinander in Verbindung stehen und welche Aufgabenbereiche sie jeweils abdecken. Monolithische Software-Architekturen verbinden ihre funktionalen Elemente in einem untrennbaren, homogenen Gebilde.

Eine monolithische Struktur folgt nicht dem Ansatz einer expliziten Gliederung in Teilsysteme oder Komponenten. So sind diese Systeme häufig stark an Ressourcen wie Hardware, bestimmte Datenformate und proprietäre Schnittstellen gebunden. Einen Gegensatz zu den streng gekoppelten monolithischen Architekturen bilden Client-Server-Architekturen oder allgemein verteilte Systeme. [Lip15]

Software-Architekturen unterliegen allgemein einer großen Freiheit hinsichtlich ihrer Gestaltung. Dies hat gerade in früheren Zeiten zu Software-Systemen geführt, die in der Regel nicht in Teilsysteme gegliedert oder auf Basis einzelner, zusammenwirkender Komponenten erstellt wurden. Diese monolithischen Strukturen, die Benutzerschnittstelle sowie die Datenzugriffs- und Logikschicht in einem einzelnen Artefakt vereinen, sind noch heute bei älteren Systemen vorzufinden und können mit einer Reihe von Nachteilen behaftet sein: [Lip15] [Ric15a]

- Monolithische Software-Systeme sind kaum wartbar oder erweiterbar, da die einzelnen Teile dieses Systems nur mit erheblichem Aufwand modifiziert und an neue Bedingungen angepasst werden können.
- Monolithische Software-Systeme sind ab einer bestimmten Größe aufgrund der mangelnden Modularisierung nicht mehr beherrschbar, es kommt bei Modifikationen zu nicht vorhersehbaren Nebeneffekten.
- Änderungen müssen meistens mehreren Teams bekanntgegeben werden, da es häufig Modul-Verantwortlichkeiten gibt und bei größeren Änderungen mehrere Module und damit Teams betroffen sind.
- Teile des monolithischen Software-Systems können nicht oder nur schlecht wiederverwendet werden.
- Aufgrund der engen Kopplung von Teilen der Software können diese nicht nebenläufig beispielsweise zur Lastverteilung auf verteilten Systemen laufen.
- Die Laufzeiten des Erstellungsprozesses können mehrere Stunden betragen

Monolithische Architekturen weisen allerdings auch eine Reihe von Vorteilen auf: [Lip15] [Ric15a]



- Sie sind einfach zu entwickeln, da integrierte Entwicklungsumgebungen (IDE) und andere Entwicklungswerkzeuge für die Entwicklung einer einzelnen Anwendung konstruiert wurden.
- Sie sind einfach zu testen, weil nur eine große Anwendung gestartet werden muss.
- Sie sind einfach bereitzustellen, denn die Bereitstellungseinheit muss nur auf eine einzige Maschine, welche als Server fungiert, kopiert werden.
- Sie sind einfach zu refaktorisieren, da sich der gesamte Quellcode meistens in einem Repository befindet.
- Es werden meistens wenige Remote Calls aufgerufen, welches zu einer guten Betriebsleistung führt.

Würde man die genannten Funktionalitäten in einer monolithischen Software-Architektur abbilden, so wären die einzelnen Funktionen eng gekoppelt und zentralisiert implementiert und hätten die oben genannten Nachteile zur Folge. Diesem stehen heute moderne Anwendungsarchitekturen entgegen, die als sogenannte Rich-Clients realisiert werden und die besagte GUI-Applikationen auf voneinander abgegrenzten Komponenten aufbauen, deren Zusammenwirken über sorgsam definierte Schnittstellen organisiert sind. Das sind beispielsweise die typischen Client-Server-Anwendungen. So hat sich insgesamt ein Paradigmenwechsel weg von monolithischen Architekturen hin zu Serviceorientierten Architekturen (SOA) vollzogen, die auf einem flexiblen Einsatz logisch in sich abgeschlossener und nur lose gekoppelter Dienste, den Services, basieren. SOA ist ein abstraktes Architekturmuster, welches die Grundlage für die verteilte Bereitstellung, die Suche nach und die Nutzung von Diensten bietet. [Lip15]

### 3 Was sind Microservices?

Der Begriff "Microservice Architektur" kam in den letzten Jahren auf, um Softwareanwendungen zu beschreiben, die aus voneinander unabhängig deploybaren Services bestehen. Auch wenn eine formale Definition bisher fehlt, gibt es gemeinsame Merkmale hinsichtlich der Organisation entlang von Business-Capabilities, der automatisierten Bereitstellung, der Logik in den Endpunkten und der dezentralen Steuerung von Programmiersprachen und Daten. [FL15]

Die Microservice-Architektur hat sich aus den Problemen anderer gängiger Software-Architekturen entwickelt. Sie wurde nicht als eigenständige Lösung erschaffen, die auf einem praktischen Anwendungsfall beruht. Der Microservice-Architekturstil entwickelte sich natürlich aus zwei Hauptquellen: Monolithischen Anwendungen, welche mit einem Schichtenarchitekturmuster entwickelt wurden und verteilten Anwendungen, die das Serviceorientierte Architekturmuster (SOA) benutzen.

Der evolutionäre Weg von einer monolithischen Anwendung hin zu einem Microservices-Architekturstil wurde vor allem durch die Entwicklung des Continuous Delivery Prinzips angetrieben. Der Hauptgedanke hierbei war, eine Bereitstellungspipeline von der Entwicklung zur Produktion zu schaffen, welche die Bereitstellung von Anwendungen rationalisiert. Wie bereits erwähnt, bestehen monolithische Anwendungen typischerweise aus eng gekoppelten Komponenten, welche Teil einer einzelnen Bereitstellungseinheit sind. Dies macht es aufwendig und schwierig eine Anwendung zu ändern, zu testen und bereitzustellen. Die genannten Faktoren führen meist zu einer Anwendung, welche bei jedem Bereitstellen einer neuen Funktion oder einer Aktualisierung nicht mehr ordnungsgemäß funktioniert. Das Microservice-Architekturmuster thematisiert diese Probleme indem es die Anwendung in viele bereitstellbare Einheiten (Servicekomponenten) zerlegt, welche unabhängig von anderen Servicekomponenten entwickelt, getestet und bereitgestellt werden können.

Eine andere Entwicklung, die zu dem Microservice-Architekturmuster geführt hat, stammt von Problemen, die bei der Implementierung von Anwendungen mit dem Serviceorientierten Architekturmuster aufgetreten sind. Während SOA ein sehr mächtiges Architekturmuster ist und unerreichte Ebenen der Abstraktion, heterogene Verbindungsfähigkeit, Serviceorchestrierung und das Versprechen die Geschäftsziele mit der IT-Leistungsfähigkeit zu verbinden bereitstellt, ist es für viele Anwendungen dennoch oft sehr komplex, teuer, schwierig zu verstehen und aufwendig zu implementieren. Der Microservice-Architekturstil thematisiert diese Komplexität indem er die Orchestrierungsbedürfnisse eines Services eliminiert und die Verbindungsfähigkeit und den Zugriff auf Servicekomponenten vereinfacht. [Ric15a]

### 3.1 Vorteile einer Microservice-Architektur

Die Separierung von Komponenten erschafft eine effektive Umgebung für das Entwickeln und Warten höchst skalierbarer Anwendungen. Kleinere Services, welche individuell entwickelt und bereitgestellt werden, sind einfacher zu pflegen, zu reparieren und zu aktualisieren. Dies führt zu einer agileren Leistungsfähigkeit, welche auf ein sich schnell änderndes Umfeld reagieren kann. (vgl. [Dwy15])

- **Eliminierung eines Single-Points-of-Failure:** Das Separieren von Komponenten einer Anwendung macht es weniger wahrscheinlich, dass das Versagen einzelner Komponenten zum Versagen aller Komponenten führt oder das Versagen von Hardware das ganze System außer Betrieb nimmt. Abgestürzte Prozesse können isoliert werden und die Kommunikation mit nicht erreichbaren Endpunkten kann wiederholt werden, bis sie erfolgreich ist.
- **Schnellere Iterationen:** Entwickler können sich mehr auf eine spezifische Aufgabe konzentrieren und mit Programmiersprachen arbeiten, die ihnen vertraut sind. Um ein Update bereitzustellen müssen Entwickler nur den Bereitstellungsprozess des einzelnen Services durchlaufen, welches zu einem agileren Leistungsvermögen führt.
- **Effektive Skalierbarkeit:** Das Skalieren einer einzelnen Serviceebene wird kosteneffizienter und ist auf Anfrage sehr elastisch. Um Elastizität zu gewähren kann zum Beispiel ein Mechanismus eingefügt werden, der bei einem Lastenanstieg die Anzahl der bereitgestellten Services erhöht. Dies führt dazu, dass einzelne Aufgaben gleichzeitig bearbeitet werden können ohne das restliche Verhalten der Anwendung zu beeinflussen.
- **Versionierung:** Programmierschnittstellen können effektiver versioniert werden, weil einzelne Services ihrem eigenen Schema folgen. Große Veröffentlichungen können auf der Anwendungsebene durchgeführt werden, während man Services auf Anfrage aktualisieren kann.
- **Sprachflexibilität:** Jeder Service kann in Abhängigkeit von den Präferenzen des Programmierers, der Aufgabentauglichkeit oder der Zuordnung zu einer spezifischen Bibliothek in einer unterschiedlichen Programmiersprache geschrieben werden.

### 3.2 Nachteile einer Microservice-Architektur

Jede Anwendungsarchitektur, die versucht die Probleme der Skalierbarkeit zu lösen, hat eine Reihe von Herausforderungen zu bewältigen, die durch die Komplexität eines verteilten Systems entstehen. Eine Anwendung in einzelne Services zu zerlegen

bedeutet, dass nun mehr bewegliche Teile gepflegt werden müssen. Dies ist durch den Architekturstil beabsichtigt, aber neue Faktoren müssen dabei berücksichtigt werden. (vgl. [Dwy15])

- **Inter-Service Kommunikation:** Entkoppelte Services brauchen einen zuverlässigen, effektiven Weg um miteinander zu kommunizieren. Dabei darf die Gesamtanwendung nicht verlangsamt werden. Das Liefern von Daten über das Netzwerk bringt Wartezeiten und potentielle Ausfälle mit sich, welche die Benutzererfahrung beeinflussen können. Es ist ein gängiger Ansatz eine Nachrichtenwarteschlange als eine zuverlässige Transportschicht einzusetzen.
- **Datenkonsistenz:** In jeder verteilten Architektur ist das Sicherstellen von Datenkonsistenz eine Herausforderung. Mehrere replizierte Datenbanken und konstante Datenzustellung können ohne die entsprechenden Vorgänge sehr einfach zu Inkonsistenzen führen.
- **Aufrechterhaltung einer hohen Verfügbarkeit:** Eine hohe Verfügbarkeit zu gewährleisten ist eine Voraussetzung in jedem Produktionssystem. Microservices stellen effektivere Isolation und Skalierbarkeit bereit, allerdings trägt die Betriebszeit eines jeden Services zu der Gesamtverfügbarkeit und somit auch zu der Benutzererfahrung einer Anwendung bei. Jeder Service muss seine eigenen Maßnahmen zur verteilten Kommunikation implementiert haben um eine anwendungsweite Verfügbarkeit gewährleisten zu können.
- **Testen:** Die enge Kopplung von Code und Abhängigkeiten innerhalb der einzelnen Services führt zwar zu einer einfacheren Entwicklungsumgebung für bestimmte Services, allerdings auch zu größeren Herausforderungen beim Testen. Services kommunizieren häufiger miteinander oder sind auf eine Datenquelle oder Programmierschnittstelle angewiesen. Einen einzelnen Service zu testen setzt eine komplette Testumgebung für diesen Service voraus.

## 4 Beschreibung des monolithischen Systems

In diesem Kapitel wird das existierende Monolithische System beschrieben.

### 4.1 DFDFAK als akademisches Beispiels-Unternehmen zum Testen von Integrationstools

In der IT-Landschaft eines Unternehmens gibt es verschiedene Anwendungen, die miteinander kommunizieren müssen. Diese Kommunikation wird vermittelt durch eine Integrationsarchitektur (Enterprise Application Integration = EAI). Eine Form einer Integrationsarchitektur stellt ein Enterprise Service Bus (ESB) dar. Meist werden zur Implementierung einer EAI entweder Rahmenwerke oder Kaufsoftware von Softwareherstellern benutzt. Diese Implementierungen müssen getestet werden. Zu diesem Zweck nutzt die NoveTec das fiktive Unternehmen DFDFAK (Deutsche Firma die fast alles kann). Die Anwendungslandschaft der DFDFAK besteht repräsentativ aus drei fachlichen Domänen (hier: A, B und C), welche aus verschiedenen Applikationen und Datenbanken bestehen könnten. Die fachlichen Details dieser Domänen sind für die Arbeit mit der Anwendungslandschaft uninteressant.

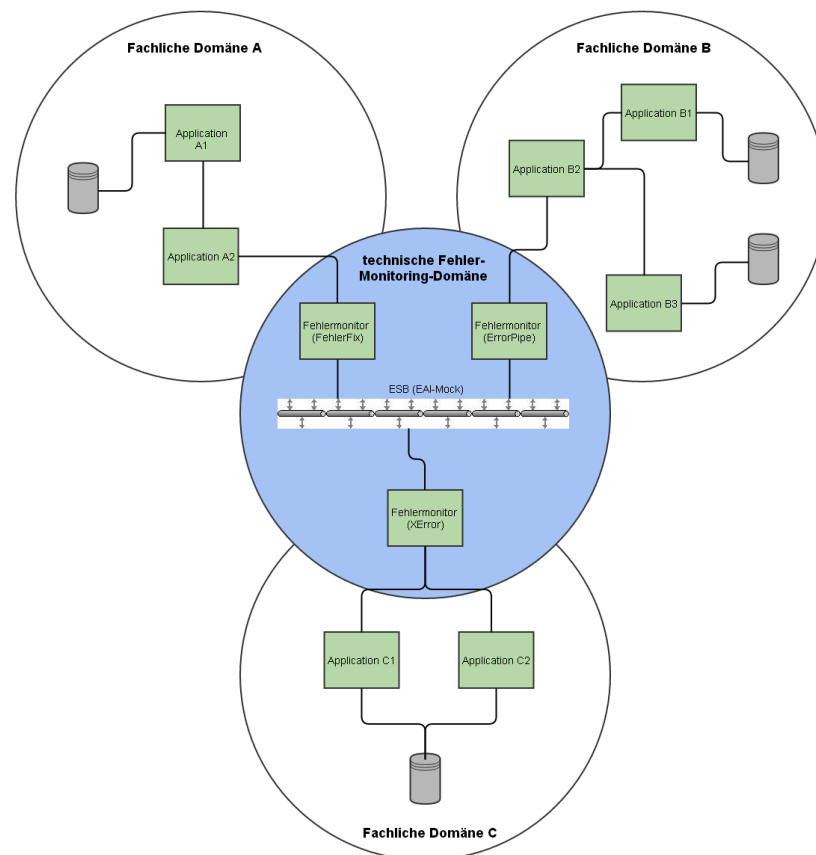


Abbildung 1: Anwendungslandschaft der DFDFAK

Aus jeder der drei Domänen wird eine Anwendung abstrahiert, deren Aufgabe es ist, Fehler, die in IT-Prozessen auftreten, entgegenzunehmen und darzustellen. Diese drei Anwendungen werden zu einer künstlichen Fehler-Monitoring-Domäne zusammengefasst und kommunizieren über einen Enterprise Service Bus (ESB) miteinander (siehe Abbildung 1).

Die Anwendungen, die für die Fehler-Monitoring-Domäne (die Integrationsdomäne) benutzt werden, tragen die Namen FehlerFix, ErrorPipe und XError und sind über einen Enterprise Service Bus miteinander verbunden. Für eine realitätsnähere Implementierung kommunizieren die Fehlermonitoranwendungen in verschiedenen Formaten über den ESB. Die Anwendung XError benutzt beispielsweise das XML-Format für die Codierung seiner Nachrichten, während die Anwendung FehlerFix Nachrichten im FixedLength-Format verwendet. Der ESB hat somit die Aufgabe, erhaltene Nachrichten in das für die Zielanwendungen richtige Format umzuwandeln und die Fehlernachrichten an die richtigen Zielanwendungen weiterzuleiten.

Von den drei Anwendungen werden Fehlernachrichten mit folgendem Inhalt erwartet:

- Fehlercode
- Fehlertext
- Timestamp

Alle Anwendungen besitzen eine eigene Schnittstelle. Ein Enterprise Service Bus soll die Kommunikation zwischen den Systemen ermöglichen. Die Anforderungen an den ESB sind:

- Entgegennahme von Fehlerdaten in unterschiedlichen Formaten auf unterschiedlichen Eingangsschnittstellen:
  - Java Message Service (JMS)
  - WebService
  - Socket
  - File Transfer
- Umwandlung des Eingangsformates in das Zielformat
- Routing zum gewünschten Zielsystem
- Versenden der Fehlerdaten über unterschiedliche Ausgangsschnittstellen:
  - JMS
  - WebService

- Socket
- FileTransfer

#### 4.1.1 DFDFAK Systeme

Ein Altsystem bezeichnet in der Informatik eine etablierte, historisch gewachsene Anwendung im Bereich Unternehmenssoftware.[vgl. [Wik15a]] Um die in der Anforderung beschriebene Anwendung zu simulieren wurden in der Anwendungslandschaft der DFDFAK drei Altsysteme gebaut, die jeweils sowohl eine File-Schnittstelle, als auch eine der oben genannten Übertragungsarten implementieren. Dabei simulieren die Ausgänge der jeweiligen Altsysteme die unterschiedlichen Fehlersituationen. Die Eingangs- sowie die Ausgangsschnittstelle sind dabei jeweils identisch. Tabelle 1 zeigt die Formate und Schnittstellen, die realisiert werden.<sup>1</sup>

Tabelle 1: DFDFAK Altsysteme

Altsystem	FehlerFix	ErrorPipe	XError
Format	fixed length	delimited	XML
Schnittstelle	JMS	Socket	WebService

#### FehlerFix

Das älteste der drei fehlerverarbeitenden Systeme heißt FehlerFix und verfügt über eine JMS-Schnittstelle, über die Fehlerereignisse importiert bzw. exportiert werden können. Die Fehlerdatenfelder sind an festen Stellen eines Strings zu finden. Tabelle 2 zeigt die Datenstruktur der FehlerFix-Anwendung und Abbildung 2 die vollständige Fehlernachricht als String.

Tabelle 2: FehlerFix Datenstruktur

Nummer	Name	Länge
1	code	5
2	text	6
3	timestamp	7

<sup>1</sup>Jede der Anwendungen implementiert jeweils auch eine File-Ein- und Ausgangs-Schnittstelle



Abbildung 2: FehlerFix Fehlerereignis Beispiel im fixed length Format

## ErrorPipe

Ein weiteres fehlerverarbeitendes System heißt ErrorPipe. Die externe Repräsentation der Fehler ist hier ebenfalls ein String, jedoch sind die einzelnen Felder durch Delimiter getrennt. Die Anwendung kann Fehlerergebnisse über Sockets importieren und exportieren. Eine Fehlernachricht im Delimited-Format ist in Abbildung 3 zu finden.

**E|12345| error| 345rtz**

Abbildung 3: ErrorPipe Fehlerereignis Beispiel im delimited Format

## X-Error

Das dritte System ist X-Error. Ein Fehlerereignis wird im XML-Format ausgegeben, das einer vorgegebenen XSD (XML Schema Definition) entspricht. Diese Anwendung benutzt einen Webservice zur Kommunikation. In Listing 1 ist eine Fehlernachricht im XML-Format zu finden. Der wesentliche Inhalt der Fehlernachricht befindet sich in Zeile sechs bis Zeile neun. Der verbleibende Teil der Nachricht ist für die spätere Weiterverarbeitung nicht relevant.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <etdErrorMsg xmlns:xsi=
3 "http://www.w3.org/2001/XMLSchema-instance">
4 <Error>
5     <Reason>
6         <Code>12345</Code>
7         <Text> error </Text>
8     </Reason>
9     <TimeStamp> 345rtz </TimeStamp>
10    <Source>
11        <Schema></Schema>
12        <Module></Module>
13        <Collab></Collab>
14    </Source>
15    <ErrorInfo>

```



```

16         <Category></Category>
17         <Severity></Severity>
18     </ErrorInfo>
19     <Resubmit>
20         <Host></Host>
21         <Port></Port>
22         <Queue></Queue>
23         <Mode></Mode>
24     </Resubmit>
25     <Republish>
26         <Host></Host>
27         <Port></Port>
28         <Queue></Queue>
29         <Mode></Mode>
30     </Republish>
31     <InputData>MTIzNDUgZXJyb3IgMzQ1cnR6</InputData>
32 </Error>
33 </etdErrorMsg>

```

Listing 1: XError-Fehlernachricht

#### 4.1.2 Integrationsszenario

Abbildung 4 zeigt eine schematische Darstellung der Integrationsarchitektur. Die drei Anwendungen können jeweils auch über die zusätzlich zur spezifischen Schnittstelle existierenden File-Schnittstelle Daten austauschen. So ist es möglich, die Integrationsarchitektur zu testen, ohne dass die Anwendungen gestartet werden müssen. Dateien, welche die Fehlernachrichten enthalten, werden hierzu in das Input Verzeichnis der Plattform kopiert. Das Routing erfolgt über den Dateinamen:

$$< Sourceformat > \_to\_ < TargetFormat > -n. < Endung >$$

Dabei können **SourceFormat** und **TargetFormat** xml, delim und fix entsprechen. Die Dateiendung entspricht dem Sourceformat: xml = \*.xml, delim = \*.dat und fix = \*.txt. Anhand dieser Endungen kann das Fehlerereignis übersetzt werden.

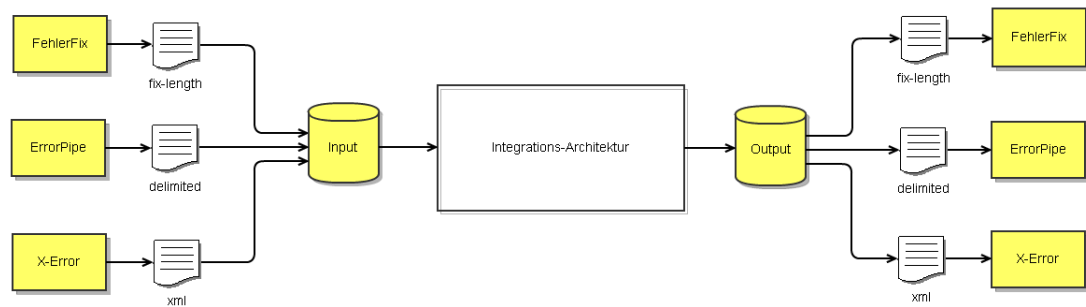


Abbildung 4: DFDFAK Integrationsarchitektur

### 4.1.3 Einführung in das Thema JMS

JMS steht für Java Message Service und ist eine Programmierschnittstelle (API) für die Ansteuerung einer Message Oriented Middleware (MOM) zum Senden und Empfangen von Nachrichten aus einem Client heraus, der in der Programmiersprache Java geschrieben ist. JMS hat das Ziel, lose gekoppelte, verlässliche und asynchrone Kommunikation zwischen den Komponenten einer verteilten Anwendung zu ermöglichen. (vgl. [Wik15c])

Messaging ist eine Möglichkeit der lose gekoppelten und verteilten Kommunikation in Form von zwischen den Softwarekomponenten verschickten Nachrichten. Messaging versucht, die sonst enge Kopplung anderer Kommunikationsmöglichkeit wie TCP Kommunikation über Sockets, CORBA oder RMI durch die Einführung einer zwischen den Clients gelegenen Komponente aufzubrechen. Damit wird sichergestellt, dass den Clients keine genaueren Informationen über die Gegenstellen ihrer Kommunikation zur Verfügung gestellt werden müssen, was sowohl den Einsatzbereich als auch die Wartung und Wiederverwendung der Komponenten erhöht.

JMS und der hiermit angesteuerte Dienst unterstützen zwei unterschiedliche Ansätze zum Versenden von Nachrichten, zum einen die Nachrichtenwarteschlange (engl. Queue) für sogenannte Point-to-Point Verbindungen und zum anderen ein Anmelde-Versendesystem (engl. Topic) für Publish-Subscribe-Kommunikation. [Wik15c]

- Im ersten Verfahren (siehe Abbildung 5) verschickt der Sender Nachrichten an eine Queue, an der ein oder mehrere Empfänger hängen. Ist kein Empfänger verfügbar, kann die Nachricht optional gespeichert werden und potentielle Empfänger können sie jederzeit später abholen. Für den Fall nur eines Empfängers, kann man dies am besten mit einem Paketdienst vergleichen. Jede Sendung hat genau einen Empfänger. Ist dieser nicht zu Hause, kann er sich die Sendung zu einem beliebigen Zeitpunkt später abholen. Bei mehreren Empfängern wird bei

der Zustellung der Nachrichten sichergestellt, dass jede eingereichte Nachricht exakt einmal zugeteilt wird. Hierdurch lässt sich Lastenverteilung realisieren, bei der Empfänger beliebig hinzugefügt und entfernt werden können.[Wik15c]

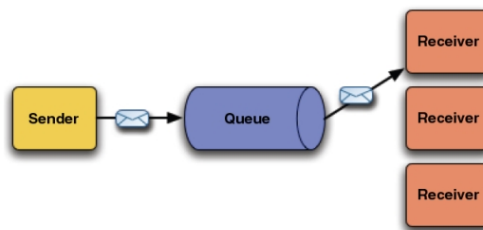


Abbildung 5: JMS Queue (Quelle: URL: <http://www.straub.as/java/jms/basic.html>)

- Bei dem Anmelde-Versendesystem (siehe Abbildung 6) werden die Nachrichten vom Publisher an ein Topic geschickt, auf das eine beliebige Anzahl von Empfängern hört. Wird die Nachricht nicht konsumiert, weil kein Empfänger sich an das Topic angemeldet hat, geht die Nachricht verloren. Das Ergebnis hierbei ist analog zum Broadcasting im TCP/IP. Die Nachricht steht nur zum Zeitpunkt des Sendens zur Verfügung. Wahlweise können die Nachrichten auch zwischengespeichert werden (durable-subscription).[Wik15c]

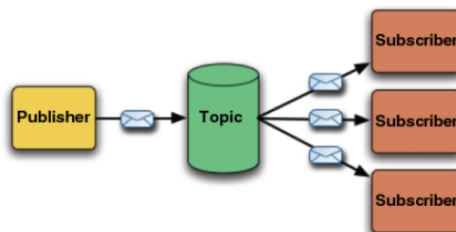


Abbildung 6: JMS Topic (Quelle: URL: <http://www.straub.as/java/jms/basic.html>)

Um JMS nutzen zu können wird ein JMS-Provider benötigt, der die Topics, Queues und Sessions verwaltet.(vgl. [Wik15c]) Für die Implementierung wurde ActiveMQ von Apache verwendet und das Sendeverfahren per Queue implementiert, weil der Typ der Nachrichtenübertragung der FehlerFix-Anwendung JMS ist. ActiveMQ läuft als eigenständiger Provider neben dem Enterprise Service Bus, was bedeutet dass der Provider in seinem eigenen Prozess läuft (stand alone) und damit separat von den JMS-Client-Prozessen.

## 4.2 Grundlagen zu Spring Integration

Spring Integration ist ein freies Rahmenwerk für Enterprise Application Integration. Es ist eine Erweiterung des Spring Rahmenwerkes, um die Implementation von

Entwurfsmustern zu unterstützen. Spring ist ebenfalls ein quelloffenes Rahmenwerk, welches die Entwicklung mit Java/Java EE vereinfachen soll und gute Programmierpraktiken fördern soll. Das primäre Ziel von Spring Integration ist es, ein simples Modell für die Erschaffung von Enterprise Integration Lösungen zu liefern, während das Separation-of-Concerns-Prinzip beibehalten wird, um wart- und testbaren Code zu erschaffen. (vgl. [FPBF12])

Beim Verwenden von Spring Integration ist es vor allem wichtig das Prinzip der Dependency Injection zu verstehen. Dies ist eines der Grundprinzipien von Spring. Mit Dependency Injection ist es möglich, die Verantwortlichkeit für den Aufbau des Abhängigkeitsnetzes zwischen den Objekten eines Programmes aus den einzelnen Klassen in eine zentrale Komponente zu überführen. In einem objektorientierten programmierten System ist dagegen jedes Objekt selbst dafür zuständig, seine Abhängigkeiten, also benötigte Objekte und Ressourcen, zu verwalten. Dafür muss jedes Objekt einige Kenntnisse seiner Umgebung mitbringen, die es zur Erfüllung seiner eigentlichen Aufgabe normalerweise nicht benötigen würde. Dependency Injection überträgt die Verantwortung für das Erzeugen und die Verknüpfung von Objekten an eine eigenständige Komponente. Bei Spring ist dies der Spring Container. Dadurch wird der Code des Objektes unabhängiger von seiner Umgebung. Das kann Abhängigkeiten von konkreten Klassen beim Kompilieren vermeiden und erleichtert besonders die Erstellung von Unit-Tests. [Wik15b]

Bei Spring Integration wird die Verwaltung der Abhängigkeiten in ein XML-Dokument geschrieben, welches den Ablauf des Programmes beschreibt. Der Vorteil hierbei ist, dass bei Änderungen im Programmablauf, das Programm selbst nicht neu kompiliert werden muss. Es müssen lediglich die Einträge in der XML-Datei geändert werden.

### 4.3 Der Enterprise Service Bus (ESB)

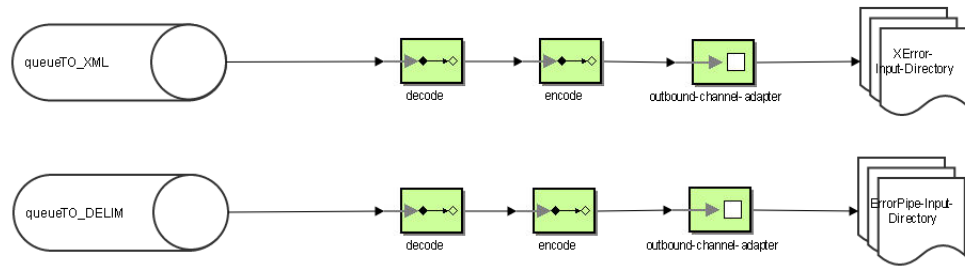
Der im DFDFAK implementierte monolithische ESB ist mit Spring Integration geschrieben und soll die Kommunikation zwischen den drei Legacy-Systemen FehlerFix, XError und Errorpipe über Fehlernachrichten vermitteln. Dabei nutzt der ESB die File-Schnittstellen der Anwendungen ErrorPipe und XError, sowie die JMS-Schnittstelle der FehlerFix-Anwendung. Die Anwendung ErrorPipe verfügt des Weiteren noch über eine Socket- und die Anwendung XError über eine WebService-Schnittstelle, diese wurden jedoch nicht im ESB implementiert. Für die Untersuchung im Rahmen dieser Arbeit ist die Betrachtung der File- und JMS-Schnittstellen der Anwendungen ausreichend.

In Abbildung 7 sind die einzelnen Transportrouten des Enterprise Service Bus abgebildet. Jede Anwendung kommuniziert mit den beiden anderen Anwendungen, was zu einer Gesamtmenge von sechs Routen führt. Alle Routen durchlaufen eine Pipeline, deren Ablauf in jeder Route gleich ist. Eine Fehlernachricht wird versendet und von einem *Service Activator* entgegengenommen. Dieser leitet die Nachricht an einen Funktionsaufruf weiter, der die Nachricht aus dem Ausgangsformat aus in ein kanonisches Datenformat umwandelt. Die Nachricht im kanonischen Format wird an eine weitere Funktion weitergeleitet, die das Format ins Zielformat umwandelt. Danach wird die umgewandelte Nachricht an einen *Outbound Channel Adapter* übergeben, welcher die Nachricht zur Zielanwendung weiterleitet.

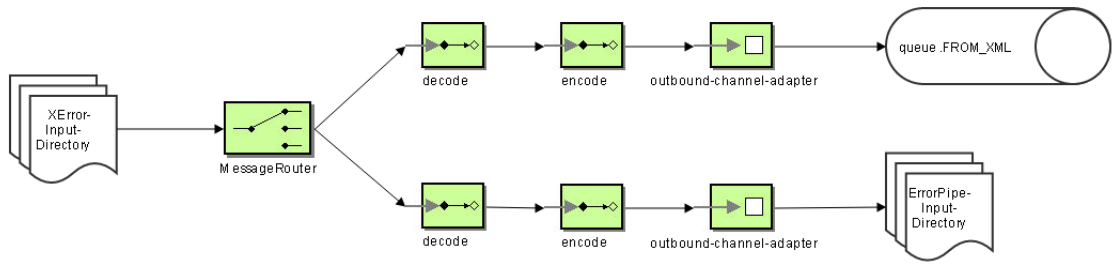
Im Fall der FehlerFix-Anwendung werden die Daten aus einer JMS-Message-Queue ausgelesen bzw. übersetzte Nachrichten an eine Queue gesendet, damit FehlerFix die Nachrichten aus dieser Queue empfangen kann. Die beiden anderen Anwendungen ErrorPipe und XError verfügen über ein Verzeichnis "data/out" in welches zu versendende Dateien gespeichert bzw. ein Verzeichnis "data/in" in welchem Dateien empfangen werden können.

Die Anwendung FehlerFix benutzt beim Versenden von Fehlernachrichten bereits zwei Queues, welche für jeweils eine der Zielanwendungen gedacht sind. Aus diesem Grund müssen die Fehlernachrichten, die von FehlerFix versendet werden, nicht mehr geroutet werden.

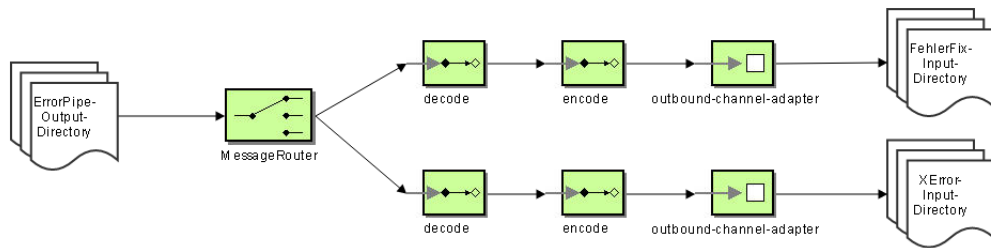
Bei den Anwendungen ErrorPipe und XError nimmt der ESB die Nachrichten über das Dateiformat auf. Aus dem Dateinamen der Fehlernachricht bestimmt ein *Message Router* die richtige Transportroute. Ein *File-to-String-Transformer* wandelt die Dateien bei Erhalt in Strings um und sendet sie dann an die Verarbeitungspipeline weiter.



(a) Nachrichtenübertragung FehlerFix



(b) Nachrichtenübertragung XError



(c) Nachrichtenübertragung ErrorPipe

Abbildung 7: Transportrouten der drei Anwendungen FehlerFix, XError und ErrorPipe

Anhand der Message-Routen in Abbildung 7 ist erkennbar, dass die monolithische Architektur in einzelne Microservices zerlegt werden kann, indem man jede Route in einem einzelnen Service laufen lässt. Durch die klare Trennung der einzelnen Routen ist wird deutlich, dass der ESB in verschiedene Microservices geteilt werden kann. Im nächsten Kapitel werden verschiedene Teilungsmöglichkeiten der monolithischen Architektur näher erläutert.

## 5 Teilung des Monolithen in Microservices

Im letztem Kapitel wurde die Struktur der monolithischen Architektur erläutert. Dabei wurden die Datenstrukturen und Kommunikationsmöglichkeiten der einzelnen Anwendungen erklärt. Außerdem wurde die Funktion der Programmierschnittstelle JMS erläutert und die grundlegenden Eigenschaften von Spring Integration näher erklärt. Am Ende des Kapitels wurde die Funktion des Enterprise Service Bus beschrieben.

### 5.1 Der Skalierungswürfel

In [AF10] wird ein sehr nützliches dreidimensionales Skalierbarkeitsmodell beschrieben: der Skalierungswürfel (siehe Abbildung 8). In diesem Modell wird das Skalieren einer Anwendung durch das Ausführen von Duplikaten hinter einem Lastenverteiler als X-Achsen-Skalierung beschrieben. Die anderen beiden Arten der Skalierung sind die Y-Achsen-Skalierung und die Z-Achsen-Skalierung. Eine Microservice-Architektur kann im Gegensatz zu einer monolithischen Architektur, welche nur an der X- und der Z-Achse skalieren kann, an der Y-Achse skaliert werden. Im folgenden werden die Unterschiede deutlich.

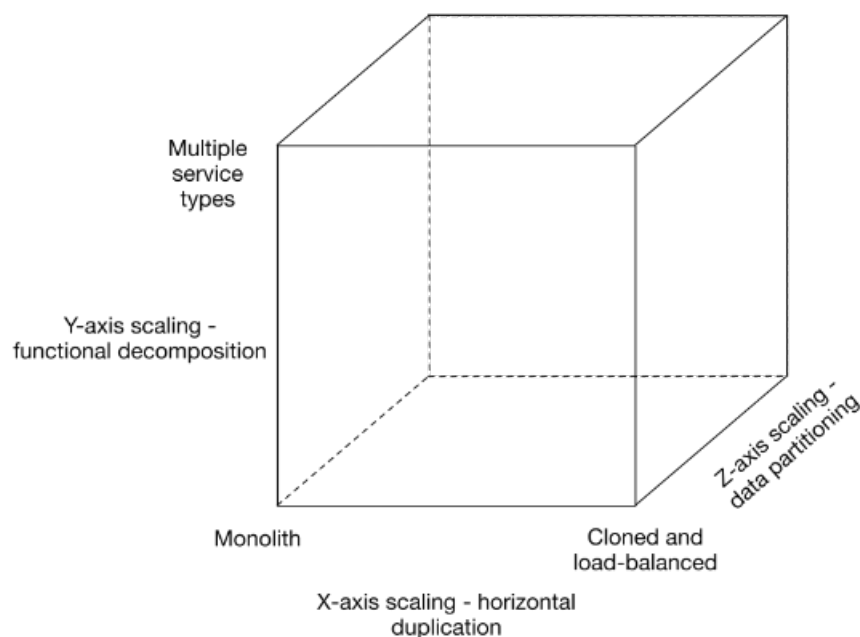


Abbildung 8: Skalierungswürfel

#### 5.1.1 X-Achsen Skalierung

Bei der X-Achsen-Skalierung werden mehrere Kopien einer Anwendung hinter einem Lastenverteiler ausgeführt. Wenn es  $N$  Kopien gibt, dann verarbeitet jede Kopie  $1/N$  der Last. Dies ist die am weitesten verbreitete Form eine Anwendung zu skalieren. Ein Nachteil dieses Ansatzes ist, dass jede Kopie der Anwendung potentiell Zugriff

auf alle Daten hat. Caches benötigen daher mehr Speicher um effektiv zu sein. Ein weiterer Nachteil dieses Ansatzes ist, dass er nicht die Probleme einer ansteigenden Entwicklungs- und Anwendungskomplexität berücksichtigt. [Ric15b]

### 5.1.2 Y-Achsen Skalierung

Anders als die X- und Z-Achsen-Skalierung, welche daraus besteht, identische Kopien einer Anwendung ausführen zu lassen, teilt die Y-Achsen-Skalierung die Anwendung in mehrere verschiedene Services auf. Jeder Service ist verantwortlich für eine oder mehrere verwandte Funktionen. Es gibt einige Möglichkeiten eine Anwendung in Services zu unterteilen. Ein Ansatz ist es eine Verb-basierte Zerteilung zu benutzen und einen Service zu definieren, der einen einzelnen Anwendungsfall behandelt. Die andere Option ist die Zerlegung der Anwendung nach Nomen, um damit Serviceverantwortungen für Operationen zu schaffen, die mit einer bestimmten Entität verwandt sind, z.B. Kundenmanagement. Eine Anwendung kann auch aus einer Kombination aus Verb-basierter und Nomen-basierter Zerlegung bestehen. [Ric15b]

### 5.1.3 Z-Achsen Skalierung

Bei der Z-Achsen-Skalierung läuft auf jedem Server eine identische Kopie des Codes. In diesem Aspekt ist sie verwandt mit der X-Achsen-Skalierung. Der wesentliche Unterschied besteht jedoch darin, dass jeder Server nur für eine Teilmenge der Daten verantwortlich ist. Einige Komponenten des Systems sind dafür verantwortlich jede Anfrage an den passenden Server weiterzuleiten. Oft wird als Weiterleitungskriterium ein Attribut einer Anfrage, z.B. der Primärschlüssel der Entität, auf die zugegriffen wird, verwendet. Ein anderes mögliches Kriterium ist der Kundentyp. Zum Beispiel könnte eine Anwendung zahlenden Kunden eine höhere Leistung zuweisen, als Kunden, welche die Anwendung kostenlos nutzen, indem sie die zahlenden Kunden an Server weiterleitet, die eine höhere Leistungsfähigkeit haben. [Ric15b]

## 5.2 Teilungsmöglichkeiten der monolithischen Architektur

Wie bereits erläutert, skaliert die Microservice-Architektur an der Y-Achse des Skalierungswürfels, also durch das mehrfache Bereitstellen der erforderlichen Funktionalitäten. Die monolithische Architektur ist in Abbildung 9 gut zu erkennen. Die drei Anwendungen FehlerFix, XError und ErrorPipe kommunizieren über einen ESB miteinander. Hierbei ist das Problem, dass bei einer Skalierung der Architektur, der ESB mehrfach bereitgestellt und die Lasten mit einem Lastenverteiler verwaltet werden müssen. Ebenso muss bei jeder einzelnen Änderung der gesamte ESB neu bereitgestellt werden.



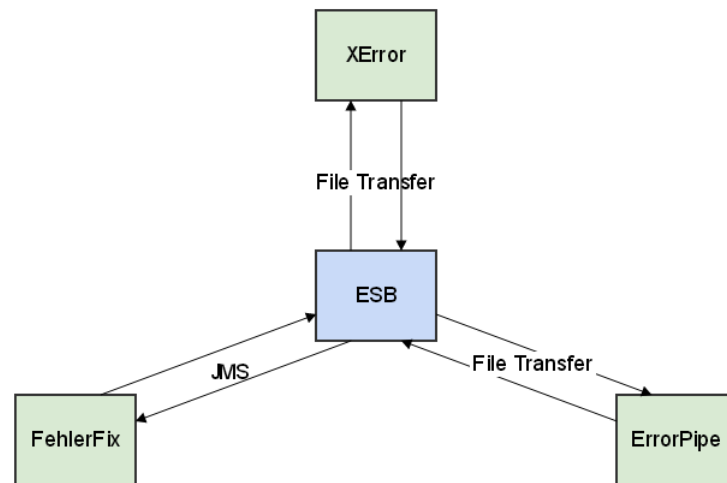


Abbildung 9: Monolithische Architektur

Abbildung 10 zeigt die Nachrichtenpipeline des Monolithen. Von den Anwendungen gesendete Nachrichten werden über einen Connector weitergeleitet an einen Decoder. Dieser wandelt die Nachricht aus dem Anwendungsformat in ein kanonisches Datenformat. Im nächsten Schritt weist ein Router die Nachricht der richtigen Zielanwendung zu. Danach wird die Nachricht aus dem kanonischen Datenformat in das Zielformat umgewandelt und anschließend von einem weiteren Connector an die Zielanwendung geschickt.

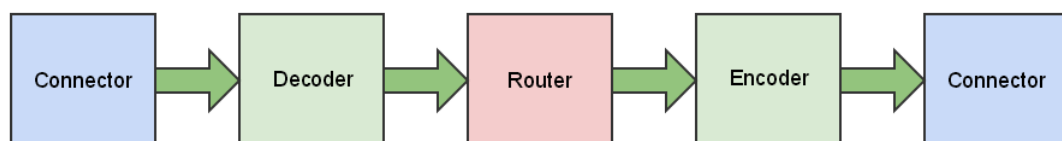


Abbildung 10: Nachrichtenroute des ESB

Bei der Zerlegung des Monolithen in verschiedene Microservices erscheint es am sinnvollsten die Services zuerst nach Aufgabenbereichen zu zerlegen. In Abbildung 7 wurden bereits alle Transportrouten einmal aufgelistet. Dabei ist zu erkennen, dass die einzelnen Routen Komponenten mit gleicher Funktion aufweisen. In Abbildung 11 ist eine Möglichkeit gezeigt, welcher Teil der monolithischen Architektur in einen Microservice zerlegt werden kann. Dabei umfasst der Service die rot eingerahmten Elemente. Die Aufgabe des Service ist es eine Nachricht zu empfangen, sie aus dem Ausgangsformat in das kanonische Datenformat zu bringen und an die richtige Anwendung zu senden.

Aus Abbildung 11 geht allerdings noch nicht hervor, was mit der Nachricht passiert, nachdem der Service die Nachricht im kanonischen Datenformat verschickt hat.

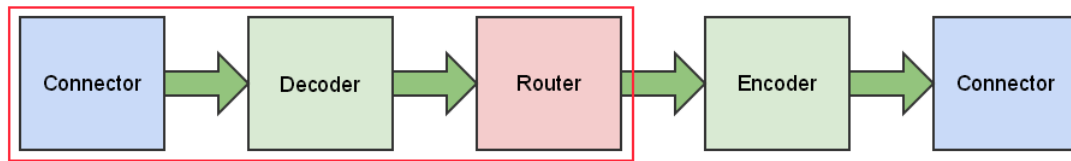


Abbildung 11: Zerlegung des ESB in Microservices

In Abbildung 12 wird, anders als oben beschrieben, ein kleinerer Teil des ESB in einen Microservice verpackt. Diese Möglichkeit verwendet nur den Connector und den Decoder und soll schnelle Änderungen möglich machen, wenn davon ausgegangen werden kann, dass sich während des Betriebes der Architektur das Datenformat häufig ändert. Der Nachteil dieses Ansatzes ist, dass noch ein eigener Routing-service in Betrieb genommen werden muss, der die Nachrichten im kanonischen Datenformat entgegen nimmt und weiterleitet.

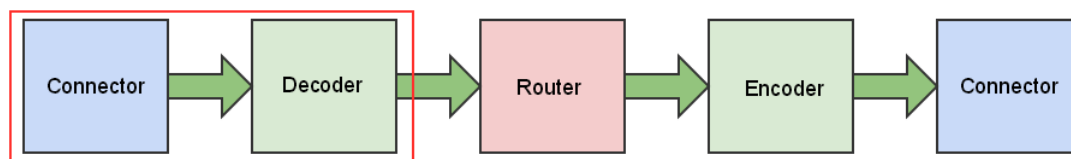


Abbildung 12: Alternative Zerlegungsmöglichkeit des ESB

In Abbildung 13 wurde der Monolith nach einzelnen Aufgabenbereichen anstatt nach Abschnitten der Übertragungs-pipeline geteilt. Diese Variante erzeugt jeweils einen Service für die Verwaltung der Connectoren, das Decodieren bzw. Encodieren aller Datenformate und für das Routing. Aus programmiertechnischer Sicht scheint dieser Ansatz sehr praktisch zu sein, da bei Änderungen die Aufgabenbereiche der Services klar definiert und einfach zu verstehen sind. Im laufenden Betrieb bringt eine solche Teilung allerdings keinen Vorteil bezüglich der Skalierbarkeit, da bei einem Lastenanstieg jeder Service skaliert werden muss.

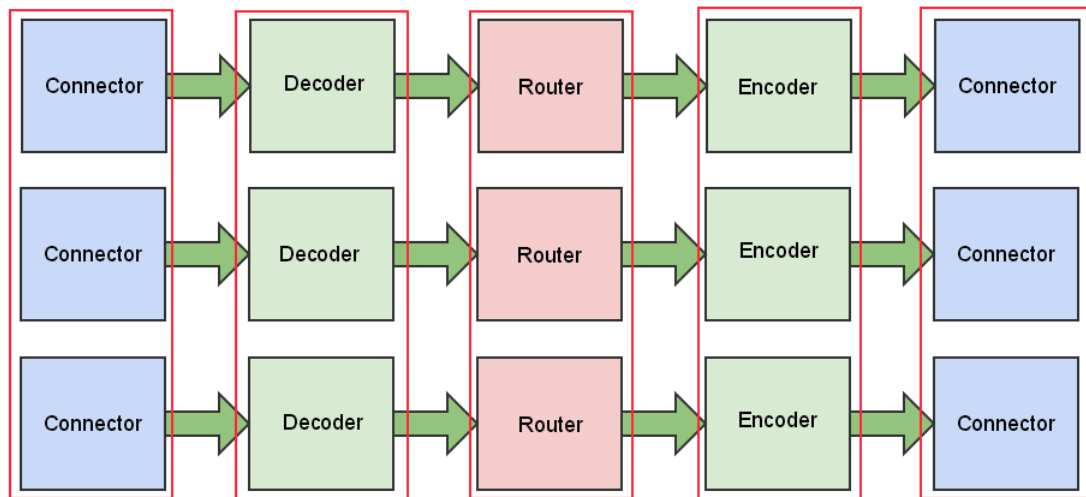


Abbildung 13: Zerlegung nach Aufgabenbereichen

In Abbildung 7 konnte man bereits erkennen, dass sich die monolithische Architektur anhand von Servicerrouten zerlegen lässt. Für jede Anwendung fallen dabei  $n$  Nachrichtenrouten an, wobei  $n$  für die Anzahl der Zielanwendungen steht, an die Nachrichten verschickt werden sollen. Daraus ergibt sich, dass es drei mal zwei also sechs Nachrichtenrouten gibt, denn die monolithische Architektur verfügt über drei Anwendungen, welche jeweils mit zwei anderen Anwendungen kommunizieren sollen. Die sechs zu betrachtende Nachrichtenrouten sind folgende:

- FehlerFix  $\rightarrow$  XError
- FehlerFix  $\rightarrow$  ErrorPipe
- XError  $\rightarrow$  FehlerFix
- XError  $\rightarrow$  ErrorPipe
- ErrorPipe  $\rightarrow$  FehlerFix
- ErrorPipe  $\rightarrow$  XError

Da nur drei verschiedene Nachrichtenformate existieren können sich je zwei der Routen einen gemeinsamen De- und Encoder teilen. Daraus folgt, dass es insgesamt drei Microservices geben muss, die als De- und Encoder funktionieren, um alle Routen abzudecken. Die daraus entstehende Microservice-Architektur ist in Abbildung 14 gezeigt.

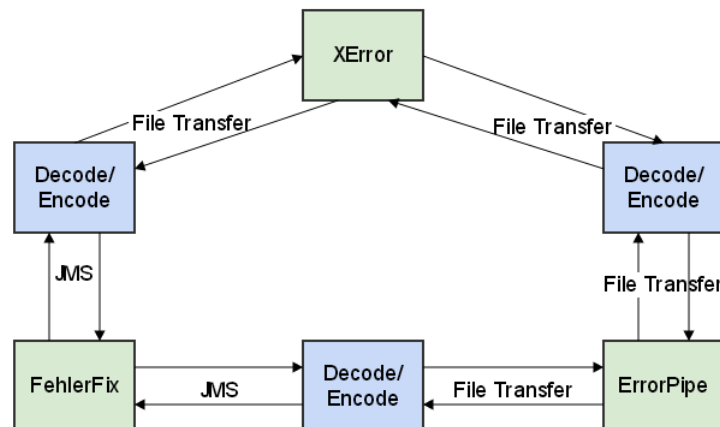


Abbildung 14: Microservices mit bidirektionalen Message-Routen

Die monolithische Architektur anhand von Nachrichtenrouten zu zerlegen bringt den Vorteil mit sich, dass nur der Service skaliert werden muss, welcher für die Kommunikation zwischen den beiden Anwendungen zuständig ist. Auch hier kann dies anhand eines Lastenverteilers geschehen. Wenn sich während der Laufzeit herausstellt, dass die entstandene Last nur Ursache einer unidirektionalen Nachrichtenkommunikation ist, besteht die Möglichkeit den ESB in mehrere unidirektionale Services zu zerlegen. Ein Beispiel dafür wäre ein Microservice, dessen Aufgabe es ist, Fehlernachrichten von der Anwendung FehlerFix an die Anwendung XError zu übertragen. Dies bietet den Vorteil, dass die Gesamtarchitektur sehr individuell skalierbar wird. Eine solche Architektur ist in Abbildung 15 zu sehen.

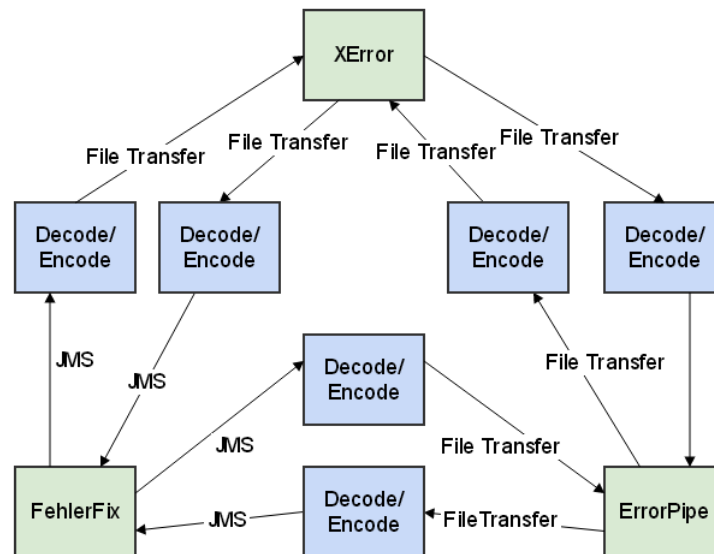


Abbildung 15: Microservices mit unidirektionalen Message-Routen

Nachteile dieser Teilungsmöglichkeit findet man vor allem dann, wenn sich die Anwendungslandschaft verändern sollte. Zum Beispiel könnte eine neue Anwendung hinzukommen. Bei vier Anwendungen steigt die Anzahl der benötigten unidirektionalen Services von sechs auf zwölf. Bei fünf Anwendungen sind es bereits 20. Dies führt zu einem sehr hohen Leistungsverlust. Die dazu gehörige Formel ist bei bidirektionalen Schnittstellen  $n * (n-1)$ .

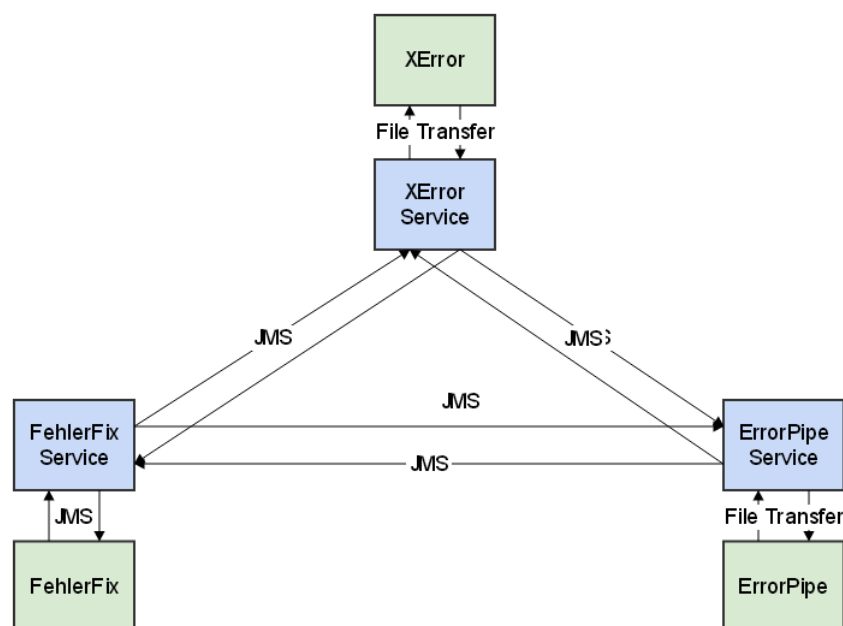


Abbildung 16: Auf Fehlermonitor zugeschnittene Microservices

Eine Alternative zu der Zerteilung der Architektur anhand von Nachrichtenrouten ist in Abbildung 16 abgebildet. Bei dieser Architektur schicken die Anwendungen ihre Fehlernachrichten an einen Service, welcher Nachrichten, die von den Anwendungen kommen, in ein kanonisches Datenmodell umwandelt und diese dann an den gewünschten Service weiterleitet. Beim Empfang der Nachricht vom entsprechenden Anwendungsservice wird das kanonische Datenmodell dann in das Format der Anwendung gebracht. Diese Methode bringt den Vorteil, dass bei Änderung, zum Beispiel an der Programmierschnittstelle einer Anwendung, nur der entsprechende Service angepasst werden muss, da die Services untereinander über das kanonische Datenformat kommunizieren. Ebenso muss bei einem Lastenanstieg nur der Service skaliert werden, dessen Anwendung die zusätzliche Last erzeugt.

## 6 Praktischer Teil

Für diese Arbeit wurde die in Abbildung 16 gezeigte Architektur umgesetzt. Die Architektur besteht aus drei Services, die jeweils ein kanonisches Datenformat encode und decodieren und die im kanonischen Datenformat vorliegenden Nachrichten an den richtigen Zielservice weiterleiten. Dabei wurden die einzelnen Services so gestaltet, dass sie dem Pipes-und-Filter-Architekturmuster entsprechen. Dieser Architekturstil ist in [HW04] beschrieben und wird benutzt, um größere Verarbeitungsaufgaben in eine Sequenz von kleineren, unabhängigen Verarbeitungsschritten (Filter), die durch Nachrichtenkanäle (Pipes) miteinander verbunden sind, aufzuteilen. Jeder Filter stellt dabei ein sehr einfaches Interface bereit. Er erhält eine Nachricht von der eingehenden Pipe, verarbeitet die Nachricht und publiziert die Ergebnisse an die ausgehende Pipe. Eine Pipe verbindet einen Filter mit einem anderen, indem sie Nachrichten von einem Filter zum nächsten schickt. Weil alle Komponenten das gleiche externe Interface benutzen, können die Komponenten zu verschiedenen Lösungen zusammengefügt werden, indem sie durch unterschiedliche Pipes verbunden werden.

In den einzelnen Microservices erfolgt dies auf gleiche Weise. Die Services selbst sind in Java mit Spring Integration geschrieben. Spring Integration stellt eine Ereignisgesteuerte Architektur bereit, wobei die einzelnen Komponenten beim Erhalt einer Nachricht eine Aufgabe ausführen. In Abbildung 17 wird die Kernarchitektur eines Services gezeigt. Hierbei wird die Pipes-und-Filter-Architektur des Services deutlich. Wenn eine Fehlermeldung von einer der Anwendungen (FehlerFix, XError, ErrorPipe) beim Service ankommt, wird diese Fehlermeldung aus ihrem normalen Format in ein kanonisches Datenformat decodiert. Im Prinzip könnte der Service die Nachricht in diesem Zustand bereits weiterleiten, jedoch wird die Nachricht vorher in einen String umgewandelt, da die Übertragung von Objekten per JMS nicht ohne das Erschaffen von Abhängigkeiten funktioniert. Bei Nachrichten, die von einem Service zu einem Zielservice gesendet wurden, läuft der Prozess komplett anders herum. Zuerst wird die Nachricht mit dem String in ein kanonisches Datenmodell gebracht, welches dann in das Format des jeweiligen Services formatiert wird.

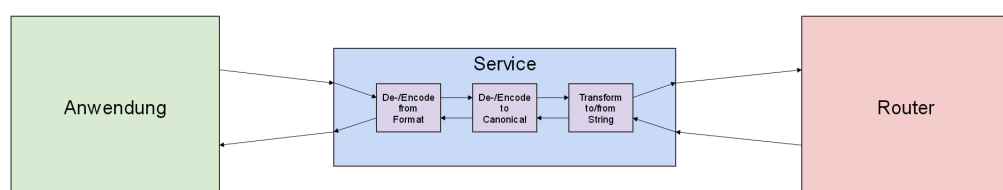


Abbildung 17: Pipes and Filter

Die oben beschriebene Gesamtarchitektur ist in Abbildung 18 dargestellt. Insgesamt wurde aus der monolithischen Architektur mit drei Altanwendungen und einem monolithischen ESB eine Microservice-Architektur. Die drei Altsysteme sind unberührt geblieben, weil diese nicht Teil der Integrationsdomäne sind. Die Integrationsdomäne bezieht sich lediglich auf die Integration von Anwendungen in einem Businesskontext. Es wurde untersucht, wie die Anwendungen miteinander kommunizieren können. Aus dem monolithischen ESB sind insgesamt drei Microservices geworden. Für jede Anwendung gibt es je einen Service, der die Nachrichten der Anwendungen aufbereitet. Dabei wird das Eingangsformat in ein kanonisches Datenformat umgewandelt bzw. von einem kanonischen Datenformat in das entsprechende Format umgewandelt, welches die zum Microservice passende Anwendung verarbeiten kann. Die Services können sich untereinander Nachrichten per JMS zusenden.

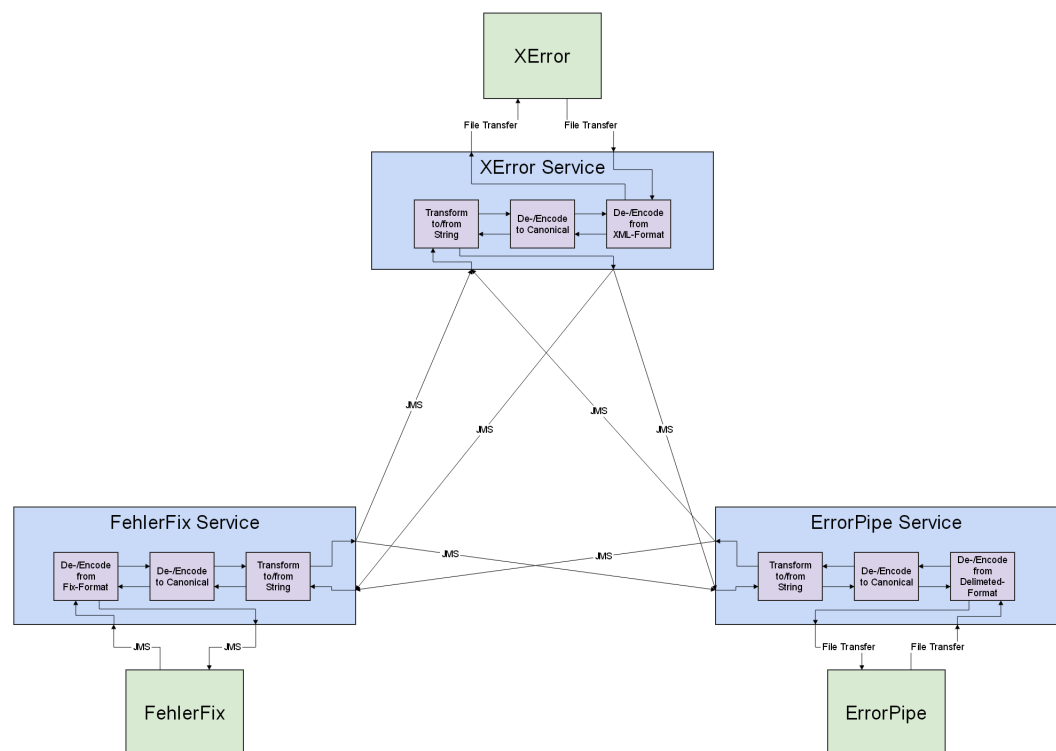


Abbildung 18: Umgesetzte Architektur

## 6.1 Umsetzung

Die Microservices liegen als sogenannte Fat-Jars vor. Dies sind Java Archive, die bereits alle Abhängigkeiten (zum Beispiel zu bestimmten Bibliotheken) enthalten um lauffähig zu sein. Codiert wurden die einzelnen Services in Eclipse, einem quelloffenen Programmierwerkzeug zur Entwicklung von Software. Die Fat-Jars wurden mit Maven [Com08], einem Build-Management-Tool der Apache Software Foundation, welches auf



Java basiert, erstellt. Die einzelnen Services können über eine Kommandozeilenkonsole aufgerufen werden. Voraussetzung zum erfolgreichen Einsetzen der Services ist ein installiertes Java Development Kit (JDK) und die Datei "context.xml" sowie eine Properties-Datei. Die Datei "context.xml" wird vom Spring-Container benutzt und gibt der Anwendung vor, in welcher Abfolge die einzelnen Klassen und Funktionen aufgerufen werden. Die Properties-Datei enthält die Addresspfade der benötigten Ordner (zum Beispiel den Ordner aus welchen die Fehlernachrichten der Anwendungen ausgelesen werden) und die Namen der JMS-Queues. Somit können diese Eigenschaften angepasst werden, ohne die Anwendung neu kompilieren zu müssen.

## 6.2 Testen der Architektur

Im Folgenden wird auf das Testen der Microservice-Architektur und die dabei aufgetretenen Schwierigkeiten eingegangen. Dabei können verschiedene Testverfahren in drei Sektionen unterteilt werden:

- Klassentests der einzelnen Service
- Funktionstests der Services
- Gesamtsystemtest.

Da die Services in Java geschrieben wurden, sind die Klassentests der einzelnen Services mit JUnit relativ einfach zu bewerkstelligen. JUnit ist ein Framework zum Testen von Java-Programmen, das besonders für automatisierte Unit-Tests geeignet ist. [Wik15d]

Funktionstests der einzelnen Services sind schwieriger durchzuführen. Um die vollständige Funktion des Microservices zu testen, muss der Dateneingang und -ausgang simuliert werden. Dies könnte beispielsweise mit einem Mock-Objekt bewerkstelligt werden.

Ein Mock-Objekt ist in der Softwareentwicklung ein Objekt, welches als Platzhalter für echte Objekte innerhalb von Modultests verwendet wird. Es ist nicht immer möglich oder erwünscht, ein einzelnes Objekt vollkommen isoliert zu testen. Soll die Interaktion eines Objektes mit seiner Umgebung überprüft werden, muss vor dem eigentlichen Test die Umgebung nachgebildet werden. Das kann umständlich, zeitaufwändig, nur eingeschränkt oder überhaupt nicht möglich sein.

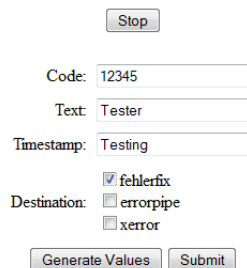
In diesen Fällen können Mock-Objekte helfen. Mock-Objekte implementieren die Schnittstellen, über die das zu testende Objekt auf seine Umgebung zugreift. Sie stellen sicher, dass die erwarteten Methodenaufrufe vollständig, mit den korrekten Parametern und in der erwarteten Reihenfolge durchgeführt werden. Das Mock-Objekt

liefert keine Echtdateien zurück, sondern vorher zum Testfall festgelegte Werte. Das Mock-Objekt kann somit dazu verwendet werden, ein bestimmtes Verhalten nachzustellen. [Wik15e]

Eine mögliche Lösung für das Testen des Microservices der Anwendung XError ist es ein Mock-Objekt zu schreiben, welches Dateien erstellt, die der Service zu verarbeiten hat.

Um die vollständige Funktion der Microservice-Architektur automatisiert testen zu können, wäre es erforderlich eine komplexe Testumgebung zu schreiben. Als Alternative dazu können manuelle Testszenarien benutzt werden. Ein mögliches Testszenario ist zum Beispiel das Versenden einer Nachricht von der Anwendung XError zur Anwendung FehlerFix. Eine mögliche Umsetzung ist im Folgenden beschrieben. Eine Fehlernachricht wird von der Weboberfläche von XError erzeugt (siehe Abbildung 19). Die zu übertragende Fehlermeldung hat folgende Werte: Code = 12345, Text=Tester, Timestamp=Testing.

#### XError - Send Messages



Stop

Code: 12345

Text: Tester

Timestamp: Testing

Destination: ☒ fehlerfix  
☐ errorpipe  
☐ xerror

Generate Values Submit

(c) NovaTec Consulting GmbH

[Deployed Webservices](#)  
[Schemadefinitions for the Webservices](#)  
[home](#)

Abbildung 19: Weboberfläche von XError zum Senden der Test-Nachricht

Nach dem Erzeugen der Fehlernachricht liegt diese als Datei "xml\_to\_fix.dat" im Verzeichnis "data/out" der XError-Anwendung vor.

Wenn die Microservice Architektur korrekt funktioniert, wird in der graphischen Oberfläche der FehlerFix-Anwendung die korrekte Nachricht angezeigt werden. In Abbildung 20 ist die korrekte Nachricht abgebildet. Dies zeigt, dass die Nachrichtenübertragung erfolgreich war. Dieser Test ist allerdings nicht besonders aussagekräftig, da er lediglich anzeigt, dass die Anwendung funktioniert. Für eine sinnvolle Nachverfolgung des Datenflusses kann beispielsweise ein Logging-System verwendet werden.



Abbildung 20: Graphische Oberfläche von FehlerFix mit der erhaltenen Fehlermeldung

### 6.3 Vorteile der umgesetzten Architektur

Die umgesetzte Architektur bringt einige Vorteile mit sich, die eine solche Lösung rechtfertigen. In Kapitel 3 wurden die Vorteile einer Microservices Architektur angesprochen. Im Folgenden wird überprüft, ob die einzelnen Vorteile auch auf die umgesetzte Architektur zutreffen.

- **Eliminierung eines Single-Point-of Failure:** In der umgesetzten Architektur existiert kein Single-Point-of-Failure. Sollte ein Microservice abstürzen so können die anderen Service weiterhin Nachrichten übertragen. Es können sogar Nachrichten an den abgestürzten Service übertragen werden, da der Service, sobald er wieder aktiv ist, die versendeten Nachrichten von der JMS-Queue abrufen kann.
- **Schnellere Iterationen:** Um einen Service zu aktualisieren muss bei der umgesetzten Architektur nur der geänderte Service neu bereitgestellt werden, die anderen Services können unberührt bleiben.
- **Effektive Skalierbarkeit:** Die Nachrichtenübertragung ist durch die Zerlegung des Monolithen skalierbar geworden. Einzelne Services können individuell mehrfach bereitgestellt werden. Das Verwenden von JMS-Queues fungiert hierbei als automatischen Lastenverteiler.
- **Versionierung:** Die Services in der umgesetzten Architektur können auf Anfrage aktualisiert werden, da sie ihrem eigenen Schema folgen.

### 6.4 Nachteile der umgesetzten Architektur

Während der Entwicklung der Microservice-Architektur haben sich einige der bereits in Kapitel 3 genannten Nachteile bestätigt:

- **Sprachflexibilität:** Die Entscheidung für JMS als Nachrichtensystem ging auf Kosten der Sprachflexibilität. Eine Änderung zu einem anderen Übertragungsmedium, zum Beispiel per HTTP, würde die Sprachflexibilität für die einzelnen Services allerdings wieder möglich machen.
- **Inter-Service Kommunikation:** Die Kommunikation zwischen den Services benötigt mehr Leistung, als dies in der monolithischen Architektur der Fall war. Allerdings liefert der Einsatz von JMS einen zuverlässigen und effektiven Weg für die Services, um miteinander zu kommunizieren.
- **Datenkonsistenz:** Das Sicherstellen von Datenkonsistenz ist in der umgesetzten Architektur zu vernachlässigen, da keine Datenbanken benutzt werden.
- **Aufrechterhaltung einer hohen Verfügbarkeit:** Bei der umgesetzten Architektur gibt es keinen Mechanismus, der eine hohe Verfügbarkeit sicherstellt. Bei einem Ausfall einzelner Services müssen diese manuell neu gestartet werden. Allerdings werden an Services gesendete Nachrichten im JMS-Broker zwischengespeichert und können später abgefragt werden, sodass keine Nachrichten verloren gehen.
- **Testen:** Das Testen der Architektur hat sich als relativ schwierig herausgestellt. Das Testen einzelner Klassen ist dank JUnit ohne Weiteres möglich. Um die Funktion eines Microservices zu testen wird allerdings eine individuelle Testumgebung für den einzelnen Service benötigt.

## 7 Ausblick

Eine Möglichkeit das Bereitstellen von Microservices zu vereinfachen ist Docker. Docker ist eine quelloffene Software Engine zur automatisierten Bereitstellung von Anwendungen in Containern. Die Container-Technologie existiert schon seit längerem und unterscheidet sich von der Hypervisor Virtualisierung, bei welcher eine oder mehrere unabhängige virtuelle Maschinen auf physischer Hardware über eine Vermittlungsschicht laufen. Container laufen im User Space auf einem Betriebssystemkern. Container Virtualisierung wird auch oft Betriebssystemvirtualisierung genannt. Die Containertechnologie erlaubt es mehrere isolierte User Space Instanzen auf einem einzelnen Hostsystem laufen zu lassen. Container werden im Allgemeinen als eine schlanke Technologie angesehen, weil sie nur einen geringen Leistungsverlust aufweisen. Anders als bei traditionellen Virtualisierungs- oder Paravirtualisierungstechnologien brauchen Container keine Emulations- oder Hypervisorschicht um lauffähig zu sein, sondern benutzen das normale Systemaufrufinterface des Betriebssystems. Dies reduziert den Leistungsverlust und erlaubt es eine größere Anzahl von Containern auf einem einzelnen Host laufen zu lassen. (vgl. [Tur14])

Docker fügt eine Anwendungsbereitstellungs-Engine, welche auf einer virtualisierten Container-Ausführungsumgebung läuft hinzu. Es wurde dazu entwickelt eine leichtgewichtige und schnelle Umgebung zum Ausführen von Code bereitzustellen. Docker-Images sind die Bausteine der Docker-Welt. Container werden von Images gestartet. Docker hilft dabei Container zu bauen und zu entwickeln, in welchen Anwendungen und Services laufen können. Container können einen oder mehrere laufende Prozesse beinhalten. Docker Container sind eine gute Ergänzung zu einer Microservice-Architektur, da sie den Bereitstellungsprozess vereinfachen. Services können dabei so entwickelt werden, dass sie als ein Prozess in einem Docker-Container laufen. Dies hat den Vorteil, dass nach richtiger Konfiguration nur der Docker-Container gestartet werden muss. Der Service läuft auf jedem System, denn der Docker-Container enthält sein eigenes Betriebssystem und die Anwendung wurde für dieses Betriebssystem optimiert.

Während der Zerlegung der monolithischen Architektur in eine Microservice-Architektur wurde versucht die einzelnen Services in Docker-Container lauffähig zu bekommen. Dabei hat sich herausgestellt, dass das Bereitstellen im Container sehr einfach vonstatten ging. Eine große Komplexität war jedoch die Kommunikation zwischen den einzelnen Containern per JMS. Aus zeitlichen Gründen musste der Versuch, die Services in Docker-Container zu verpacken, abgebrochen werden. Docker wäre eine gute Lösung für das bestehende System, denn es gefährdet die technologische Unabhängigkeit von Microservices nicht, weil die einzelnen Anwendungen noch immer in verschiedenen Programmiersprachen geschrieben werden können. Eine Möglichkeit Mi-

---

crosservices mit Docker in einem zukünftigen Projekt umzusetzen wäre zum Beispiel die Kommunikation der einzelnen Service via HTTP anstelle des hier verwendeten JMS.

## 8 Fazit

Das Ziel der Arbeit, die Zerlegung einer monolithischen Architektur in eine Microservice-Architektur wurde erreicht. Das Zerlegen einer monolithischen Architektur in eine Microservice-Architektur weist eine Reihe von Vorteilen auf, die den zusätzlichen Aufwand, den eine solche Umstrukturierung mit sich bringt zu rechtfertigen. Bei der praktischen Umsetzung hat sich gezeigt, dass dies im Wesentlichen zutrifft. Ein Single-Point-of-Failure, welchen der monolithische ESB darstellte, wurde abgeschafft. Auch können Änderungen innerhalb eines Services schneller vollzogen werden und der Code eines einzelnen Service ist deutlich übersichtlicher als der Code des monolithischen ESB. Durch das Verwenden von Java Message Service als Nachrichtensystem konnte der eigentlich für Microservices geltende Vorteil der technologischen Unabhängigkeit allerdings nicht realisiert werden. Prinzipiell besteht jedoch die Möglichkeit die Services in unterschiedlichen Programmiersprachen zu schreiben, solange diese über die gleichen Schnittstellen verfügen.

Von besonderem Interesse während der Zerlegung waren die vielen Möglichkeiten die Nachrichtenpipeline des ESB in Services zu zerteilen. Dies ist auf Grund der geringen Komplexität der Nachrichtenpipeline bemerkenswert. Bei einer komplexeren Anwendung wären noch deutlich mehr Möglichkeiten denkbar gewesen. Dies führt zu dem Schluss, dass die Zerlegung einer monolithischen Architektur in eine Microservice-Architektur sehr individuell vorgenommen werden kann. Um dabei die optimale Lösung zu finden und auch auf Grund der hohen Komplexität, welche die Zerlegung mit sich bringt, muss ein Entwickler sehr gut mit dem zu zerlegenden System vertraut sein.

Insgesamt haben sich die am Anfang erwähnten Vor- und Nachteile einer Microservice-Architektur bestätigt. Ob sich die Zerlegung einer monolithischen Architektur in eine Microservice-Architektur in der Praxis lohnt, hängt von den individuellen Eigenschaften der Architektur ab. Größere und komplexere Architekturen werden die Vorteile einer Microservice-Architektur langfristig gut nutzen können. Bei kleineren Anwendungen wird sich der mit der Zerlegung auftretende Mehraufwand vermutlich nicht lohnen.

## Literaturverzeichnis

- [AF10] ABBOTT, M. ; FISHER, M.: *The Art of Scalability*. Pearson Education, Inc. , Rights and Contracts Department, 501 Boylston Street, Suite 900, Boston, MA 02116 : Pearson Education Inc., 2010
- [Com08] COMPANY, Sonatype: *Maven: The Definitive Guide*. 1st. 1005 Gravenstein Highway North, Sebastopol, CA 95472 : O'Reily Media, Inc., 2008
- [Dwy15] DWYER, I.: *Microservices - Patterns for building modern applications*. 2015
- [FL15] FOWLER, M. ; LEVIS, J.: Microservices: Nur ein weiteres Konzept in der Softwarearchitektur oder mehr? In: *OBJEKTSpektrum* 01/2015 (2015)
- [FPBF12] FISCHER, M. ; PARTNER, J. ; BOGOEVICI, M. ; FULD, I.: *Spring Integration in Action*. Manning, 2012
- [HW04] HOHPE, G. ; WOOLF, B.: *Enterprise Integration Patterns*. Pearson Education, Inc. , Rights and Contracts Department, 501 Boylston Street, Suite 900, Boston, MA 02116 : Addison-Wesley, 2004
- [JAX15] JAX: *JUnit*. <https://de.wikipedia.org/wiki/JUnit>. Version: Juli 2015
- [Lip15] LIPINSKI, K.: *Monolithische Software-Architektur*. <http://www.itwissen.info/definition/lexikon/Monolithische-Software-Architektur.html>. Version: Juli 2015
- [Ric15a] RICHARDS, M.: *Software Architecture Patterns*. First Edition. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472 : O'Reilly Media Inc., 2015
- [Ric15b] RICHARDSON, C.: *Microservices: Decomposing Applications for Deployability and Scalability*. <http://www.infoq.com/articles/microservices-intro>. Version: Juli 2015
- [Tur14] TURNBULL, J.: *The Docker Book*. 2014
- [Wik15a] WIKIPEDIA: *Altsystem - Wikipedia*. <https://de.wikipedia.org/wiki/Altsystem>. Version: Juni 2015
- [Wik15b] WIKIPEDIA: *Dependency Injection*. [http://de.wikipedia.org/wiki/Dependency\\_Injection](http://de.wikipedia.org/wiki/Dependency_Injection). Version: Juli 2015
- [Wik15c] WIKIPEDIA: *Java Message Service*. [http://de.wikipedia.org/wiki/Java\\_Message\\_Service](http://de.wikipedia.org/wiki/Java_Message_Service). Version: Juli 2015
- [Wik15d] WIKIPEDIA: *JAX 2015*. <https://jax.de/2015/>. Version: August 2015
- [Wik15e] WIKIPEDIA: *Mock-Objekt*. <http://de.wikipedia.org/wiki/Mock-Objekt>. Version: August 2015



## Anhang

### Sourcecode des FehlerFix-Microservices

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:int="
4       http://www.springframework.org/schema/integration"
5       xmlns:file="http://www.springframework.org/schema/integration/
6       file"
7       xmlns:jms="http://www.springframework.org/schema/integration/
8       jms"
9       xmlns:context="http://www.springframework.org/schema/context"
10      xsi:schemaLocation="http://www.springframework.org/schema/beans
11      http://www.springframework.org/schema/beans/spring-beans.xsd
12      http://www.springframework.org/schema/integration
13      http://www.springframework.org/schema/integration/spring-
14      integration.xsd
15      http://www.springframework.org/schema/integration/file
16      http://www.springframework.org/schema/integration/file/spring-
17      integration-file.xsd
18      http://www.springframework.org/schema/context
19      http://www.springframework.org/schema/context/spring-context-3.2.xsd
20      http://www.springframework.org/schema/integration/jms
21      http://www.springframework.org/schema/integration/jms/spring-
22      integration-jms.xsd">
23
24   <context:property-placeholder
25       location="file:${user.dir}/../src/main/resources/
26       fehlerfixservice/fehlerfixservice.properties" />
27
28   <!-- Channel Definition -->
29
30   <int:channel id="ToXmlParser" />
31   <int:channel id="ToDelimParser" />
32   <int:channel id="ToXmlDecoder" />
33   <int:channel id="ToDelimDecoder" />
34   <int:channel id="ToXmlEncoder"/>
35   <int:channel id="ToDelimEncoder"/>
36
37   <int:channel id="FromXmlService"/>
38   <int:channel id="FromDelimService"/>
39   <int:channel id="EncodeChannel"/>
40   <int:channel id="DecodeChannel"/>
41
42   <!-- Bean Definition -->
43   <bean id="jms" class="fehlerfixservice.parser.JmsParser" />
```

```
38      <bean id="fixparser" class="fehlerfixservice.parser.
        FehlerFixParser" />
39      <bean id="MessageObjectBuilder" class="fehlerfixservice.parser.
        SendableJmsObjectConverter"/>
40
41      <!-- JMS -->
42
43      <bean id="connectionFactory" class="org.apache.activemq.spring.
        ActiveMQConnectionFactory">
44          <property name="brokerURL" value="tcp://localhost:61616"
            />
45      </bean>
46
47      <!-- From FehlerFix -->
48
49      <!-- To XError -->
50      <int-jms:message-driven-channel-adapter
51          id="fix_to_xml_Adapter" channel="ToXmlParser" destination
            -name="${JMSTOXML}" />
52
53      <int:service-activator input-channel="ToXmlParser"
54          ref="jms" method="parse" output-channel="ToXmlDecoder" />
55
56      <int:service-activator input-channel="ToXmlDecoder"
57          ref="fixparser" method="decode" output-channel="
            ToXmlEncoder" />
58
59      <int:service-activator input-channel="ToXmlEncoder"
60          ref="MessageObjectBuilder" method="objectToSendable"
            output-channel="ToXmlOut"/>
61
62      <int-jms:outbound-channel-adapter id="ToXmlOut"
63          destination-name="${FIXTOXML}" />
64
65
66      <!-- To ErrorPipe -->
67      <int-jms:message-driven-channel-adapter
68          id="fix_to_delim_Adapter" channel="ToDelimParser"
            destination-name="${JMSTOERRORPIPE}" />
69
70      <int:service-activator input-channel="ToDelimParser"
71          ref="jms" method="parse" output-channel="ToDelimDecoder"
            />
72
73      <int:service-activator input-channel="ToDelimDecoder"
74          ref="fixparser" method="decode" output-channel="
            ToDelimEncoder" />
75
```

```

76     <int:service-activator input-channel="ToDelimEncoder"
77         ref="MessageObjectBuilder" method="objectToSendable"
78         output-channel="ToDelimOut"/>
79
80     <int-jms:outbound-channel-adapter id="ToDelimOut"
81         destination-name="${FIXTODELIM}" />
82
83     <!-- From Services -->
84
85     <int-jms:message-driven-channel-adapter
86         id="fromMapperAdapter" channel="FromDelimService"
87         destination-name="${DELIMTOFIX}" />
88
89     <int:service-activator input-channel="FromDelimService"
90         ref="jms" method="parse" output-channel="DecodeChannel"/>
91
92     <int-jms:message-driven-channel-adapter
93         id="fromMapperAdapter" channel="FromXmlService"
94         destination-name="${XMLTOFIX}" />
95
96     <int:service-activator input-channel="FromXmlService"
97         ref="jms" method="parse" output-channel="DecodeChannel"/>
98
99     <int:service-activator input-channel="DecodeChannel"
100         ref="MessageObjectBuilder" method="sendableToObject"
101         output-channel="EncodeChannel"/>
102
103     <int:service-activator input-channel="EncodeChannel"
104         ref="fixparser" method="encode" output-channel="ToFixOut"
105         />
106
107     <int-jms:outbound-channel-adapter id="ToFixOut"
108         destination-name="${JMSFROMXERROR}" />
109
110 </beans>

```

Listing 2: context.xml

```

1 JMSFROMERRORPIPE=queue.FROM_DELIM
2 JMSFROMXERROR=queue.FROM_XML
3 JMSTOERRORPIPE=queue.TO_DELIM
4 JMSTOXERROR=queue.TO_XML
5
6 FIXTOXML=queue.FixToXml
7 FIXTODELIM=queue.FixToDelim
8 DELIMTOFIX=queue.DelimToFix
9 XMLTOFIX=queue.XmlToFix

```

Listing 3: Properties-Datei

```
1 package fehlerfixservice.App;
2
3 import java.io.IOException;
4
5 import org.springframework.context.ApplicationContext;
6 import org.springframework.context.support.ClassPathXmlApplicationContext
7     ;
8
9 public class App {
10
11     public static void main(String[] args) throws IOException{
12         ApplicationContext context = new
13             ClassPathXmlApplicationContext("fehlerfixservice/
14                 context.xml");
15     }
16 }
```

Listing 4: App.java (Programmstart)

```
1 package fehlerfixservice.parser;
2
3 public class ErrorObject {
4
5     public static enum ErrorType {
6
7         E,
8         R
9     }
10
11     private int code;
12     private String message;
13     private String timestamp;
14     private ErrorType type;
15
16     public ErrorObject (ErrorType type,
17         int code,
18         String message,
19         String timestamp) {
20         this.code = code;
21         this.message = message;
22         this.timestamp = timestamp;
23         this.type = type;
24     }
25 }
```

```
25
26 public int getCode () {
27     return code;
28 }
29
30 public void setCode (String code) {
31     try {
32         this.code = Integer.parseInt (code);
33     } catch (Exception e) {
34         this.code = 0;
35     }
36 }
37
38 public void setCode (int code) {
39     this.code = code;
40 }
41
42 public String getMessage () {
43     return message;
44 }
45
46 public void setMessage (String message) {
47     this.message = message;
48 }
49
50 public String getTimestamp () {
51     return timestamp;
52 }
53
54 public void setTimestamp (String timestamp) {
55     this.timestamp = timestamp;
56 }
57
58 public ErrorType getType () {
59     return type;
60 }
61
62 public void setType (ErrorType type) {
63     this.type = type;
64 }
65 }
```

Listing 5: Klasse für kanonisches Datenformat

```
1 package fehlerfixservice.parser;
2
3 import fehlerfixservice.parser.ErrorObject.ErrorType;
4
5
```

```
6 public class FehlerFixParser
7     extends MessageParser {
8
9     @Override
10    public ErrorObject decode (String message) {
11        if (message.length () > 18 || message.length () < 18) {
12            return null;
13        }
14
15        int code = 0;
16
17        try {
18            code = Integer.parseInt (message.substring (0,
19                                                         5));
20        } catch (Exception e) {
21
22            return null;
23        }
24
25        String errorMessage = message.substring (5,
26                                                  11);
27        String timestamp = message.substring (11,
28                                              18);
29
30        return new ErrorObject (ErrorType.E,
31                                code,
32                                errorMessage,
33                                timestamp);
34    }
35
36    @Override
37    public String encode (ErrorObject error) {
38        if (error == null || error.getMessage () == null || error.
39            getTimestamp () == null) {
40
41            return null;
42        }
43
44        String code = "" + error.getCode ();
45
46        if (code.length () > 5) {
47
48            return null;
49        }
50
51        while (code.length () < 5) {
52            code = "0" + code;
```

```
53
54     String message = error.getMessage () != null ? error.getMessage () :
55         " ";
56
57     if (message.length () > 6) {
58
59         return null;
60     }
61
62     while (message.length () < 6) {
63         message += " ";
64     }
65
66     String timestamp = error.getTimestamp () != null ? error.getTimestamp
67         () : " ";
68
69     if (timestamp.length () > 7) {
70
71         return null;
72     }
73
74     while (timestamp.length () < 7) {
75         timestamp += " ";
76     }
77
78     return code + message + timestamp;
79 }
80
81 @Override
82 protected boolean isUsefullEncoder (String type) {
83     return true;
84
85     // return type != null ? type.contains (
86     //     ApplicationProperties.getInstance ().getFehlerFixTransfer ())
87     // : false;
88 }
89
90 @Override
91 protected boolean isUsefullDecoder (String type) {
92     return true;
93
94     // return type != null ? type.contains (
95     //     ApplicationProperties.getInstance ().getFehlerFixSenderPreFix
96     //     ()) : false;
97 }
98 }
```

Listing 6: Klasse zum En-/Decodieren von Nachrichten

```
1 package fehlerfixservice.parser;
2
3 public class ErrorObject {
4
5     public static enum ErrorType {
6
7         E,
8         R
9     }
10
11     private int code;
12     private String message;
13     private String timestamp;
14     private ErrorType type;
15
16     public ErrorObject (ErrorType type,
17                         int code,
18                         String message,
19                         String timestamp) {
20         this.code = code;
21         this.message = message;
22         this.timestamp = timestamp;
23         this.type = type;
24     }
25
26     public int getCode () {
27         return code;
28     }
29
30     public void setCode (String code) {
31         try {
32             this.code = Integer.parseInt (code);
33         } catch (Exception e) {
34             this.code = 0;
35         }
36     }
37
38     public void setCode (int code) {
39         this.code = code;
40     }
41
42     public String getMessage () {
43         return message;
44     }
45
46     public void setMessage (String message) {
47         this.message = message;
```



```
48     }
49
50     public String getTimestamp () {
51         return timestamp;
52     }
53
54     public void setTimestamp (String timestamp) {
55         this.timestamp = timestamp;
56     }
57
58     public ErrorType getType () {
59         return type;
60     }
61
62     public void setType (ErrorType type) {
63         this.type = type;
64     }
65 }
```

Listing 7: Klasse für kanonisches Datenformat

```
1 package fehlerfixservice.parser;
2
3 import org.springframework.messaging.Message;
4
5 public class JmsParser {
6     public String parse(Message<String> message) {
7         String m = message.getPayload();
8         return m;
9     }
10 }
```

Listing 8: Klasse zum Auslesen von Inhalt aus JMS-Nachrichten

```
1 package fehlerfixservice.parser;
2
3 import java.util.StringTokenizer;
4
5 import fehlerfixservice.parser.ErrorObject.ErrorType;
6
7 public class SendableJmsObjectConverter {
8     public String objectToSendable (ErrorObject e) {
9         String s = e.getCode() + "." + e.getMessage() + "." + e.
10             getTimestamp();
11
12         return s;
13     }
14
15     public ErrorObject sendableToObject (String s) {
```

```
15         StringTokenizer st = new StringTokenizer(s, ".");
16
17         int code = Integer.parseInt(st.nextToken());
18         String message = st.nextToken();
19         String timestamp = st.nextToken();
20
21         return new ErrorObject(ErrorType.E, code, message,
22                               timestamp);
23     }
```

Listing 9: Klasse zum umwandeln von kanonischem Datenformat in String