Fehlerbehandlung in einer Mikroservicearchitektur

-

Ein Ansatz aus der funktionales Programmierung

Projektarbeit Nr.:	[redacted]
vorgelegt am:	[redacted]
von:	[redacted]
Matrikelnummer:	[redacted]
Berufsakademie:	[redacted]
Studienbereich:	Informatik
Studiengang:	B. Eng. Praktische Informatik
Kurs:	[redacted]
Ausbildungsstätte:	dotSource GmbH Goethestraße 1 07743 Jena
Betreuer Praxisbetrieb:	[redacted]
Gutachter Berufsakademie:	[redacted]

Head Office Jena
Goethestraße 1
07743 Jena
FON +49 (0) 3641 797 9000
FAX +49 (0) 3641 797 9099
E-MAIL info@dotSource.de

Office Berlin
Pappelallee 78/79
10437 Berlin
FON +49 (0) 30 220 122 360

Office Leipzig
Hainstraße 1–3
04109 Leipzig
FON +49 (0) 341 9919 1000

Bankverbindung
Deutsche Bank Jena
IBAN DE63 8207 0000 0633 7778 00
BIC DEUTDE8EXXX

Commerzbank Jena
IBAN DE31 8204 0000 0259 9934 00
BIC COBADEFF821

Sparkasse Jena
IBAN DE35 8305 3030 0018 0037 61
BIC HELADEF1JEN

Geschäftsführer
Christian Otto Grötsch
Christian Malik
Frank Ertel

Gerichtsstand
Amtsgericht Jena
HRB 210634
USt-IdNr.: DE246243309
Steuer-Nr.: 162/107/03164

Sperrvermerk

Die vorliegende Arbeit beinhaltet interne und vertrauliche Informationen der Firma dotSource GmbH. Die Weitergabe des Inhaltes der Arbeit und eventuell beiliegender Zeichnungen und Daten im Gesamten oder in Teilen ist grundsätzlich untersagt. Es dürfen keinerlei Kopien oder Abschriften - auch in digitaler Form - gefertigt werden. Ausnahmen bedürfen der schriftlichen Genehmigung der Firma dotSource GmbH

I Inhaltsverzeichnis

II Abbildungsverzeichnis

1 Einleitung

In größeren Programmarchitekturen ist ein gutes Konzept zum Umgang mit Fehlern unabdingbar. Jeder Entwickler eines Teams ist geneigt etwas anders mit Fehlern umzugehen, sollte man sich innerhalb des Teams nicht auf ein gutes Konzept zur Fehlerbehandlung geeinigt haben.

Die dotSource GmbH entwirft und entwickelt nun schon seit über 14 Jahren eCommerce-Plattformen für Kunden in verschiedenen Größen und Bereichen. Abhängig von der im Projekt eingesetzten Technologie, kommen die unterschiedlichsten Techniken zur Fehlerbehandlung zum Einsatz. In einer produktiven Umgebung eines Kunden sollen die Anwendungen im Falle eines Fehlers nicht den Betrieb der Plattform beeinflussen, indem sie abstürzen oder sogar andere Teile der gesamten Plattform zum Absturz bringen. Im besten Fall gibt die Anwendung eine Fehlermeldung aus, um sie in einem Log zu speichern und arbeitet, ggf. nach einem Neustart, wie gewohnt weiter. Speziell in Microservicearchitekturen ist es wichtig, dass jede der vielen kleinen Anwendungen stets bereit ist Anfragen zu bearbeiten, da zwischen den Anwendungen sehr viel Kommunikation stattfindet und sie deshalb abhängig voneinander sind. In solch einer Architektur wird immer häufiger NodeJS als Technologiebasis verwendet, da hier der Ressourcenverbrauch sehr gering ist. Bei NodeJS handelt es sich um eine Skriptsprache auf Basis von JavaScript. Durch die dynamische Typisierung bietet diese Skriptsprache jedoch viele Möglichkeiten für Laufzeitfehler. Darum ist es wichtig, dass über alle Microservices hinweg eine einheitliche Programmstruktur herrscht, die dabei hilft Fehler gänzlich zu vermeiden oder diese elegant zu behandeln.

Bei der Entwicklung solcher Architekturen werden Fehler häufig auf unterschiedliche Art und Weise behandelt. Dabei werden Fehler meistens geworfen, um sie später an andere Stelle zu behandeln. Das macht es schwer die bestehende Codebasis zu erweitern, Fehler zu finden oder kann zu inkonsistentem Logging führen. Ein Ansatz aus der funktionalen Programmierung soll dabei helfen die Programmlogik noch klarer von den Fehlern zu trennen und eine robusterer Codebasis schaffen. Dadurch wird das Logging verbessert und Fehler können effizienter gesucht und behoben werden.

Außerdem muss genau definiert sein, wann ein Fehler vorliegt. Sollte ein Programm aufgrund von äußeren Einflüssen nicht funktionieren, liegt nicht immer sofort ein Fehler vor. Weiterhin müssen echte Fehler auch von Ausnahmen unterschieden werden. In der folgenden Arbeit soll zunächst erläutert werden, wann ein Fehler wirklich ein Fehler ist und wann nur eine Ausnahme. Außerdem soll geklärt werden welche Ursachen solche Fehler im Allgemeinen haben können und wie sie auf dieser Grundlage am besten behandelt werden.

Dabei werden verschiedene Konzepte zur Fehlerbehandlung erklärt und wie der Weg von einem unübersichtlichen *Try-Catch*-Konzept hin zu einem Ansatz aus der funktionalen Programmierung in einem Microservice aussieht, um zu evaluieren wie effizient ein derartiges Konzept in einem realen Kundenprojekt umgesetzt werden kann.

2 Was sind Fehler?

Um Fehler korrekt behandeln zu können muss erst einmal klar sein, welches Verhalten eines Programmes überhaupt einen Fehler darstellt oder möglicherweise gewollt ist. Wird an einem Server ein bestimmter Datensatz per ID angefragt und dieser Datensatz kann in einer Datenbank nicht gefunden werden, ist das durchaus ein normales Verhalten und es gibt keinen Grund kein Fehler zu werfen. Der Server hat an dieser Stelle seine Arbeit korrekt erledigt und mitgeteilt, dass dieser Datensatz nicht existiert. Möglicherweise liegt das Problem bei dem Aufrufenden selbst. Ein weiteres Beispiel ist eine Schnittstelle, bei der eine Authentifizierung notwendig ist. Wurden hier die falschen Zugangsdaten angegeben und der Vorgang kann nicht fortgesetzt werden, ist dies ebenfalls kein Fehler, sondern so vorgesehen. Soll ein Programm aber Inhalte aus einer Datei lesen und die Datei ist nicht vorhanden, die Zugriffsrechte fehlen oder die Datei enthält ein ungültiges Format, so ist das durchaus ein Fehler.

Fehler sind generell in zwei Arten zu unterteilen. Zum einen in nicht behebbare Fehler, welche unweigerlich zum Programmabbruch führen und zum anderen in behebbare Fehler. Diese Fehler können z.B. durch eine erneute Abfrage einer Nutzereingabe behandelt werden. Solche nicht behebbaren Fehler werden in der folgenden Arbeit nur noch als Fehler (im engl. *Error*) bezeichnet. Behebbare Fehler sollen nun als Ausnahmen (im engl. *Exception*) bezeichnet werden. Dadurch soll eine klare Trennung der Begrifflichkeit hergestellt werden, da diese Begriffe auch heute noch von dem ein oder anderen Softwareentwickler verwechselt oder sogar gleichgesetzt werden.

2.1 Fehler

Fehler sind meistens nicht behebbare Probleme, die den Programmablauf im Wesentlichen beeinträchtigen oder sogar verhindert.[1] Stellt das Betriebssystem beispielweise nicht genügend Arbeitsspeicher für eine Anwendung bereit, so kann diese durch eine *OutOfMemoryError* beendet werden. Hier hat man oft gar nicht die Möglichkeit eine Fehlerbehandlung innerhalb der Anwendung vorzunehmen, da selbst für die Fehlerbehandlungsroutine schon nicht genügend Arbeitsspeicher zur Verfügung steht.[2]

2.2 Ausnahme

Als Ausnahme wird ein behebbares Fehlverhalten oder korrigierbare Störung des Programmflusses gesehen.[3] Solch ein Fehlverhalten ist meistens vorhersehbar und wird in Form von verschiedenen Fehlerbehandlungsroutinen abgefangen oder durch eine geschickte Implementierung bereits gänzlich vermieden. Das ermöglicht ein sauberes Beenden der Anwendung und ist dadurch meistens nicht mit Datenverlust verbunden.

[1] [Anu18].
[2] [Jon16].
[3] [Anu18].

3 Fehlverhalten und dessen Ursachen

Bei der Behandlung verschiedener Fehler ist es wichtig zu unterscheiden welche Quelle oder Ursache ein Fehler hat.

3.1 Benutzer

Verarbeitet ein Programm Eingaben eines Benutzers, können potenziell immer Fehler entstehen, wenn die Eingabe nicht sorgfältig geprüft wird. Es muss immer davon ausgegangen werden, dass ein Benutzer eine falsche Eingabe tätigt.[4] Dies kann unabsichtlich oder sogar mutwillig geschehen. Deshalb sollte man dafür sorgen, dass alle Eingaben so früh wie möglich im Programmablauf validiert werden.

3.2 System

Eine weitere Ursache für fehlerhaftes Verhalten kann das System selbst sein, auf dem das Programm ausgeführt wird. Stellt das Betriebssystem beispielweise nicht genügend Ressourcen zur Ausführung des Programms bereit oder ein Filehandle kann nicht korrekt geöffnet oder geschlossen werden, kann es den Programmablauf wesentlich beeinflussen.[5] Ein solches Fehlverhalten gilt in der Regel als nicht deterministisch, da es nicht vorhersehbar ist. Funktionierte das Programm im ersten Durchlauf noch reibungslos, so kann es beim nächsten Durchlauf derart beeinflusst werden, dass es sogar zu einem Absturz kommt. Ob dieses Fehlverhalten als Ausnahme oder als Fehler behandelt werden sollte, ist sehr stark von der Ursache abhängig.

3.3 Entwickler

Die schwerwiegendste Ursache für Fehlverhalten eines Programmes sind Fehler in der Programmlogik durch den Entwickler selbst.[6] Diese lassen sich meistens nicht zur Laufzeit beheben und erfordern die Aufmerksamkeit eines Entwicklers, der den Fehler sucht und anschließend eine Anpassung am Code vornimmt.

---

[4] [Jon16].
[5] Vgl. ebenda.
[6] Vgl. ebenda.

4 Arten der Fehlerbehandlung

4.1 Try/Catch Konzept am Beispiel von Java

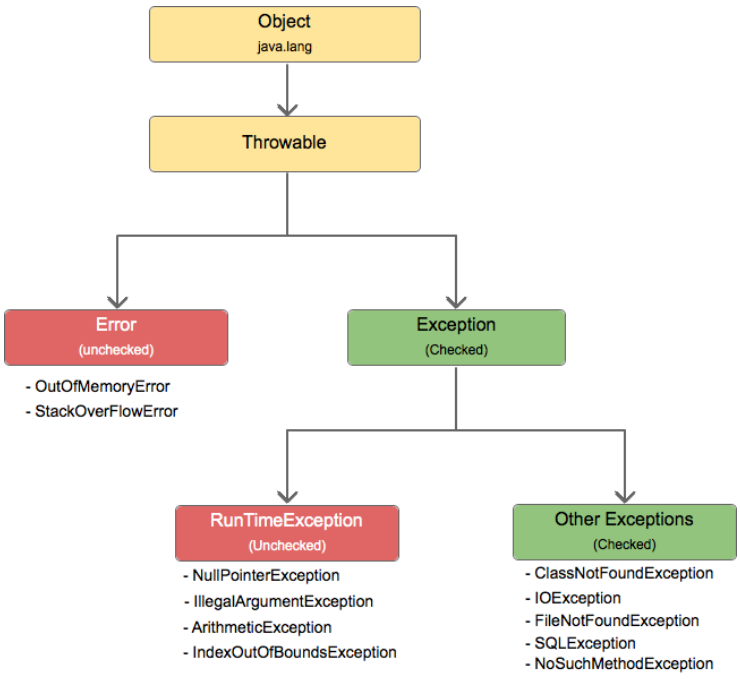
In vielen objektorientierten Programmiersprachen, wie z.B. Java oder C# aber auch in Sprachen wie JavaScript hat sich das *Try/Catch*-Konzept durchgesetzt, um Fehlerbehandlung vom restlichen Programmcode zu kapseln. Teil dieses Konzeptes ist es, im Fehlerfall alle Informationen über den Fehler zu sammeln und diese in einem Fehlerobjekt zusammenzufassen. Dieses Fehlerobjekt wird anschließend geworfen, um es später, an anderer Stelle, zu fangen und den Fehler entsprechend zu behandeln. Wird ein solcher Fehler nicht gefangen, so wird er bis zur obersten Ebene im Callstack durchgereicht. Hierbei handelt es sich meistens um das Betriebssystem selbst, welches das Programm mit dem entsprechenden Fehlercode beendet, es stürzt ab. Im folgenden Beispiel wird eine Funktion zur Division definiert. Sollte der Divisor null sein wird eine *ArithmeticException* mittels *throw* geworfen. Beim Aufruf der *divide*-Funktion wird diese Ausnahme nicht behandelt so, dass die Ausnahme bis zum Aufrufer der main-Funktion durchgereicht und das Programm mit einer Fehlermeldung beendet wird.

Abbildung 1: Java Beispiel - Throw

```
1   public class MyClass {
2       static double divide(int a, int b) {
3           if (b == 0) {
4               throw new ArithmeticException("Division durch 0");
5           } else {
6               return a/b;
7           }
8       }
9
        Run | Debug
10      public static void main(String[] args) {
11          double result = divide(4, 0);
12          System.out.println(result);
13      }
14  }
```

In Java kann jedoch nicht jedes beliebige Objekt mit einem *throw* geworfen werden. Das Objekt muss eine Instanz der Klasse *Throwable* oder aus einer daraus abgeleiteten Klasse sein.[7] Java besitzt nativ bereits zwei Subklassen, welche aus *Throwable* abgeleitet sind. Zum einen gibt es die Klasse *Exception,* welche für typische

Abbildung 2: Java - Throwable Hierarchie



Quelle: [MAN18]

Probleme, wie z.B. Division durch Null oder ähnliches gedacht ist.[8]

Solche *Exceptions* werden meistens explizit von einer Funktion geworfen, da sie in der Regel vorhersehbar sind. Anschließend werden sie später gefangen und vom Programm selbst behandelt. Zum anderen gibt die Klasse *Error*, um schwerwiegende Probleme, wie z.B. zu wenig Arbeitsspeicher, abzubilden.[9] Solche Fehler werden von der Laufzeitumgebung, etwa der *Java Virtual Maschine*, selbst geworfen und sind oft nicht vorhersehbar. Dementsprechend können diese auch nur sehr schwer gefangen werden, da sie an jeder Stelle im Programm, zu jeder Zeit auftreten können. Sie führen somit meistens zum Absturz des Programms.

[7] [Anu18].
[8] [Ora20].
[9] Vgl. ebenda.

Individuelle Fehlerobjekte müssen also von einer dieser beiden Klassen abgeleitet werden, um sie werfen zu können. Hierbei sollte aber ausschließlich von der Klasse *Exception* abgeleitet werden.

Ein *Try/Catch*-Ausdruck besteht im Wesentlichen aus zwei Blöcken. Im *Try*-Block sind die Befehle gekapselt, die potenziell zu Fehlern führen können, also direkt ein *throw* enthalten oder eine Funktion verwenden, die ein *throw* enthält.[10] Da innerhalb dieses Blockes ein Fehler auftreten kann sollten darin keine Variablen initialisiert oder andere für den weiteren Programmablauf kritische Operationen vorgenommen werden. Im Fehlerfall könnte das Programm gar nicht dazu gekommen sein, diese Anweisungen darin auszuführen. Deshalb sollte man sich im darauffolgenden Programmcode, der außerhalb dieses Blocks liegt, nicht auf Werte verlassen, welche innerhalb des *Try*-Blocks erzeugt werden, da diese möglicherweise fehlerhaft oder gar nicht initialisiert wurden. Im *Catch*-Block werden Fehler behandelt, die vorher im *Try*-Block aufgetreten sind.[11] Dieses Mal ist der Aufruf der divide-Funktion in ein *Try/Catch*-Ausdruck verpackt. Die dort geworfene *ArithmeticException* wird gefangen und die Fehlernachricht auf der Konsole ausgegeben.

Abbildung 3: Java Beispiel - Try/Catch

```
10      public static void main(String[] args) {
11          try {
12              double result = divide(4, 0);
13              System.out.println(result);
14          } catch(ArithmeticException e){
15              System.err.println(e.getMessage());
16          }
17      }
```

In objektorientierten Programmiersprachen sind die verschiedenen Fehler auch in Form verschiedener Fehlerklassen dargestellt. So kann für jede Art von Fehler ein spezieller *Catch*-Block definiert werden, um diese unterschiedlich zu behandeln. Sollte beispielsweise eine Konfigurationsdatei nicht gefunden werden, muss das Programm möglicherweise beendet und vorher noch ein Eintrag in ein Log geschrieben werden. Hat ein Benutzer aber eine falsche Eingabe getätigt, kann man ihn draufhinweisen und es erneut versuchen. Es werden also für verschiedene Fehler, verschiedene *Catch*-

[10] [W3S20].
[11] Vgl. ebenda.

Blöcke benötigt. Ähnlich wie bei überladenen Funktionen entscheidet die Laufzeitumgebung darüber welcher *Catch*-Block für den entsprechenden Fehler ausgeführt wird. In nicht objektorientieren Programmiersprachen wie JavaScript gibt es diese Möglichkeit oft nicht oder sie entspricht nicht dem Standard. Hier wird nur ein einzelner *Catch*-Block für alle Arten von Fehlern bereitgestellt.

Häufig gibt es noch einen *Finally*-Block, welcher hinter den *Catch*-Blöcken definiert wird. Dieser ist optional und wird abschließend nach allen *Catch*-Blöcken ausgeführt.[12] *Finally* wird meistens genutzt, um offene Sockets zu schließen oder wenn möglich Daten zu speichern, bevor das Programm beendet wird.

[12] Vgl. ebenda.

4.1.1 Geprüfte Ausnahmen

Bei geprüften Ausnahmen (engl. *checked exceptions*) handelt sich um Ausnahmen, die bereits vom Compiler geprüft werden und nicht erst zur Laufzeit auftreten.[13] Das bedeutet, dass eine Funktion alle Ausnahmen kennt, die explizit ausgelöst werden. Sobald diese Funktion an anderer Stelle aufgerufen wird, ist man vom Compiler gezwungen diese Ausnahme zu behandelt oder sie an die nächste Funktion im Callstack nach oben weiter zu reichen. Soll eine Funktion beispielsweise Inhalte aus einer Datei lesen, so ist es möglich, dass die Datei nicht existiert und eine *FileNotFoundException* auftritt. Hier weis der Compiler schon vorher, dass möglicherweise eine solche Ausnahme ausgelöst wird und für diese eine Fehlerbehandlungsroutine in Form eines *Try/Catch*-Ausdrucks vorgesehen werden sollte.[14] Eine solche Ausnahme ist genauso Teil der Schnittstelle einer Funktion, ähnlich wie Funktionsparameter oder der Rückgabewert. [15]

Abbildung 4: Java Beispiel - Checked Exception

```java
import java.io.FileNotFoundException;
import java.io.FileReader;

public class MyClass {
    static FileReader open(String path) throws FileNotFoundException {
        FileReader file = new FileReader(path);
        return file;
    }

    Run | Debug
    public static void main(String[] args) {
        FileReader file = open("text.txt");
    }
}
```

[13] [Lok20].
[14] Vgl. ebenda.
[15] [Ora19].

In der Funktionsdefinition von *open* wird ein *FileReader* Objekt erzeugt. Dabei kann eine *FileNotFoundException* auftreten. Diese kann direkt mittels eines *Try/Catch*-Ausdrucks behandelt werden oder sie wird an den Aufrufer weitergereicht. Das geschieht mittels des *throws*-Schlüsselworts hinter der Parameterliste der Funktionsdefinition. Hier werden alle unbehandelten Ausnahmen aufgeführt, die an den Aufrufer weitergereicht werden. In Zeile 11 wird diese Funktion genutzt, ohne diese Ausnahme zu behandeln. Die rote Markierung der Entwicklungsumgebung suggeriert bereits eine nicht behandelte Ausnahme.

4.1.2 Ungeprüfte Ausnahmen

Neben den geprüften Ausnahmen gibt es auch die vom Compiler nicht geprüften Ausnahmen. Hier wird der Aufrufende einer Funktion, welche eine solche Ausnahme wirft, nicht gezwungen diese in einem *Try/Catch*-Block zu behandeln. In Java sind diese Ausnahmen von der Klasse *RuntimeException* abgeleitet.[16] Von dieser Klasse sind alle Laufzeitfehler abgeleitet. Da Laufzeitfehler nicht beim Kompilieren auftreten, können sie auch nicht vom Compiler geprüft werden.

Entwickler neigen häufig dazu Ausnahmen zu werfen die nicht geprüft werden oder eigene Klassen von der Klasse *RuntimeException* abzuleiten, da sie diesen Ausnahmen später keine Aufmerksamkeit bei der Implementierung schenken müssen. Zu ungeprüften Ausnahmen zählt beispielsweise die *NullPointerException*.[17] Im folgenden Beispiel wird eine Datei geöffnet und diese anschließend auf null gesetzt.

Abbildung 5: Java Beispiel - Unchecked Exception

```
1  import java.io.FileNotFoundException;
2  import java.io.FileReader;
3  import java.io.IOException;
4
5  public class MyClass {
6      static FileReader open(String path) {
7          try {
8              FileReader file = new FileReader(path);
9              file = null;
10             file.read();
11             return file;
12         } catch (final FileNotFoundException e) {
13
14         } catch (final IOException e) {
15
16         }
17         return null;
18     }
19
       Run | Debug
20     public static void main(final String[] args) {
21         FileReader file = open("text.txt");
22     }
23 }
```

Die geprüften Ausnahmen *FileNotFoundException* und *IOException* werden direkt in der Funktion gefangen. Jedoch wird *file* in Zeile 9 auf null gesetzt, was in der nächsten Zeile zu einer *NullPointerException* führt. Der Compiler zwingt hier aber an keiner Stelle zur Behandlung dieser Ausnahme.

[16] Vgl. ebenda.
[17] Vgl. ebenda.

4.2 Fehlercodes als Rückgabewert

Ein anderer Ansatz besteht darin den Fehler nicht zu werfen, sondern ihn als Rückgabewert direkt zurückzugeben, sobald er auftritt. Dazu sollten genutzte Funktionen keine weiteren Abhängigkeiten besitzen.[18] Das bedeutet, dass in der Funktion keine weiteren http-Anfragen, Zugriffe auf ein Dateisystem oder ähnliches stattfinden. Außerdem sollten sie für den gleichen Eingabewert oder auch Funktionsparameter immer den gleichen Rückgabewert liefern.[19] Entscheidend hierbei ist aber, dass immer ein Wert zurückgegeben wird und andere Werte innerhalb des Programms nicht verändert werden.[20] Solche Funktionen werden als pure Funktion bezeichnet. Das bedeutet im Umkehrschluss, dass solche Funktionen auch einen Fehler zurückgeben müssen und ihn nicht werfen dürfen. Damit der Aufrufende nun einen korrekten Rückgabewert von einem Fehler unterscheiden kann, muss der zurückgegebene Fehler außerhalb des regulären Wertebereiches der Funktion liegen. Betrachtet man eine Hauptfunktion der Programmiersprache C, findet das Konzept auch dort Anwendung.[21] Die Hauptfunktion gibt den Wert Null zurück, wenn das Programm fehlerfrei abgearbeitet wurde oder einen von Null verschiedenen Wert, der zu gleich Auskunft über den aufgetretenen Fehler gibt.[22] Es ist aber nicht immer eindeutig welcher Wert nicht zum regulären Wertebereich zählt.

In Programmiersprachen wie Go oder Lua wird dieses Konzept bereits nativ unterstützt.[23] Hier ist es möglich, dass eine Funktion zwei Rückgabewerte hat. Hier werden neben dem regulären Rückgabewert auch Informationen darüber zurückgegeben, ob in der Funktion Fehler aufgetreten sind oder ob sie korrekt ausgeführt wurde. [24]

---

[18] [Yaz19].
[19] [Chi18].
[20] Vgl. ebenda.
[21] [Mül20].
[22] Vgl. ebenda.
[23] [Jes17].
[24] Vgl. ebenda.

Abbildung 6: Go Beispiel - 2 Rückgabewerte von Funktionen

```
file, err := os.Open("text.txt")
if err ≠ nil {
    log.Printf("Fehler beim öffnen der Datei: %v", err)
    return fmt.Errorf("Fehler beim öffnen der Datei: %v", err)
}
defer file.Close()
```

So muss man der Fehler nicht vom regulären Wert unterschieden werden. Weiterhin ist es in verschiedenen Szenarien sinnvoll bei einem Fehler den Vorgang erneut zu versuchen, wie etwa bei einer falschen Eingabe eines Benutzers. Es ist viel performanter über die Funktion zur Abfrage der Benutzereingabe, mittels einer Schleife zu iterieren, bis der Rückgabewert valide ist, als diese Funktion in ein *Try/Catch*-Ausdruck zu verpacken und darüber zu iterieren.[25]

Des Weiteren ist der Entwickler bereits beim Benutzen der Funktion dazu gezwungen, den Fehler an der Stelle zu behandeln, an der er auftritt. Die Funktion stellt entweder ein korrektes Ergebnis bereit oder gibt das Fehlerobjekt zurück, um den Fehler direkt zu behandeln. Das Werfen eines Fehlers hingegen kann dazu führen, dass er sich auf andere Teile des Programmes auswirkt, sollte er nicht korrekt behandelt oder sogar vergessen werden. [26] Auch wenn hier keine Fehler mittels *throw* geworfen, sondern als Ergebnis der Funktion zurückgegeben werden, muss die aufrufende Funktion das Ergebnis immer auf einen Fehler prüfen. Das führt dazu, dass solche Funktionsaufrufe immer mittels *IF*-Ausdrücken geprüft werden müssen. Ähnlich wie bei dem *Try/Catch*-Konzept, wird das Programm an diesen Stellen in mehrere Pfade aufgeteilt. Das muss bei jeder Stelle geschehen, an der ein Fehler auftreten kann. Dadurch wird der Programmcode sehr schnell unübersichtlich und im schlimmsten Fall dupliziert.

[25] [Jon16].
[26] [Jes17].

4.3 Ein funktionaler Ansatz in JavaScript

Sollte in einem Programm ein Fehler auftreten, lässt sich die Programmlogik in zwei Teile oder auch Pfade gliedern. Bei einem *Try/Catch* oder einem *IF*-Ausdruck existiert neben der normalen Programmlogik auch eine Logik zum Behandeln des Fehlers selbst. Um diese Art von Fehlerbehandlung zu vereinfachen und den Programmcode dadurch lesbarer zu gestalten, wird der Ansatz der puren Funktionen verallgemeinert und diese Rückgabewerte zu einem sogenannten *Either*-Objekt zusammengefasst. [27] Dieses Objekt stammt aus der funktionalen Programmierung und ist in Form einer disjunkten Menge als algebraischer Datentyp zu verstehen.[28] Das bedeutet, dass eine Funktion entweder den berechneten Wert oder das Fehlerobjekt zurückgibt, aber niemals beides gleichzeitig.[29] Dabei gilt in verschiedenen Programmiersprachen und deren Programmbibliotheken die Konvention, dass der Pfad im Fehlerfall als „rechter" Pfad bezeichnet wird. Handelt es sich um ein korrektes Ergebnis ist dies der „linke" Pfad.[30] Im folgenden JavaScript-Beispiel soll ein einfacher Text auf der Konsole ausgegeben werden. Dazu wird für den linken und den rechten Pfad jeweils eine Klasse definiert.

Abbildung 7: JavaScript Beispiel - Either Schritt 1

```
1  class Right {
2      constructor() { }
3  }
4
5  class Left {
6      constructor() { }
7  }
```

[27] Vgl. ebenda.
[28] Vgl. ebenda.
[29] Vgl. ebenda.
[30] [Sin19].

Nun werden die Klassen um eine Funktion *execute* erweitert. Diese nimmt zwei Parameter entgegen. Der erste Parameter ist eine Referenz auf die auszuführende Funktion, der zweite Parameter enthält wiederum die Parameter für die auszuführende Funktion.

Abbildung 9: JavaScript Beispiel - Either Schritt 2

```
class Right {
    constructor() { }

    execute(fn, param){
        fn(param);
    }
}

class Left {
    constructor() { }

    execute(){
        // hier ist nichts zu tun
    }
}
```

Diese beiden Klassen sind polymorph, da sie die gleichen Schnittstellen, jedoch aber mit anderer Funktion, bereitstellen.[31] Im rechten Pfad soll die Funktion wie gewohnt ausgeführt werden. Falls nun im rechten Pfad ein Fehler auftritt und der Programmablauf im linken Pfad fortgeführt werden soll, ist die *execute*-Funktion in dieser Klasse leer. Jetzt können beide Klassen wie folgt genutzt werden.

Abbildung 11: JavaScript Beispiel - Either Schritt 3

```

const right = new Right();
const left = new Left();

right.execute(console.log, "Hallo Welt") // Gibt "Hallo Welt" auf der Konsole aus
left.execute(console.log, "Hallo Welt") // nichts geschieht
```

[31] Vgl. ebenda.

Um nun mehrere Funktionen innerhalb dieser Pfade aufrufen zu können, müssen diese ihr zugehöriges Objekt selbst zurückgeben.[32] Dazu wird die *execute*-Funktion wie folgt erweitert.

Abbildung 13: JavaScript Beispiel - Either Schritt 4

```
1   class Right {
2       constructor() { }
3
4       execute(fn, param){
5           fn(param); // Ausführen von 'fn'
6           return this;
7       }
8   }
9
10  class Left {
11      constructor() { }
12
13      execute(){
14          // hier ist nichts zu tun
15          return this;
16      }
17  }
```

Das ermöglicht es die *execute*-Funktion verkettet hintereinander aufzurufen und somit den Programmablauf auf oberster Ebene übersichtlich zu strukturieren.

Abbildung 15: JavaScript Beispiel - Either Schritt 5

```
21  const left = new Left();
22
23  right.execute(console.log, "Hallo")
24       .execute(console.log, "Welt") // Gibt "Hallo Welt" auf der Konsole aus
25  left.execute(console.log, "Hallo")
26      .execute(console.log, "Welt") // nichts geschieht
```

Hier wurde das Objekt *left* für den linken Pfad noch explizit erzeugt. Außerdem fehlt die Logik, um von dem rechten Pfad auf den linken zu wechseln, sollte ein Fehler auftreten. Dazu muss die *execute*-Funktion für den rechten Pfad so angepasst werden, dass die nicht ihr eigenes Objekt, sondern ein neues Objekt der Klasse *Left*

[32] Vgl. ebenda.

zurückgibt.[33] Für dieses Beispiel wird festgelegt, dass der übergebene Text nicht leer sein darf. Sollte er leer sein wird ein neues Objekt von *Left* zurückgegeben.

Abbildung 17: JavaScript Beispiel - Either Schritt 6

```
1  class Right {
2      constructor() { }
3
4      execute(fn, param){
5          if(!param) // im Fehlerfall ein neues 'Left' zurückgeben
6              return new Left();
7          fn(param); // Ausführen von 'fn'
8          return this;
9      }
10 }
```

Dadurch besteht die Möglichkeit jederzeit vom rechten in den linken Pfad zu wechseln, sollte der übergebene Parameter nicht korrekt sein. Somit muss das Objekt der Klasse *Left* nicht explizit erzeugt werden, da es im Fehlerfall durch den Kontrollfluss des *Either*-Objektes selbst erzeugt wird.

Abbildung 19: JavaScript Beispiel - Either Schritt 7

```
21
22 const right = new Right();
23
24 right.execute(console.log, "Hier")
25       .execute(console.log, "steht")
26       .execute(console.log, "ein")
27       .execute(console.log, null) // Hier tritt der Fehlerfall ein
28       .execute(console.log, "ganzer")
29       .execute(console.log, "Satz")
```

Dadurch wird die Hauptfunktion des Programms sehr übersichtlich. Es ist nicht nötig *IF*-Anweisungen im Programmcode zu verteilen, Fehler zu werfen oder zu fangen. Außerdem beeinflusst der Fehler nicht die restliche Programmlogik. Im Beispiel führt die Zeile 27 zu einem Fehler. Es werden alle Anweisungen vor dieser Zeile wie gewünscht ausgeführt. Die letzten beiden Anweisungen werden ignoriert, da die Programmlogik durch den Fehler auf den linken Pfad gewechselt ist. Hier ist zu beachten, dass natürlich auch jede andere Funktion anstelle von *console.log* aufgeführt und somit hintereinander verkettet werden kann.

[33] Vgl. ebenda.

5 Implementierung in eine Microservicearchitektur

Nun soll evaluiert werden, wie hoch der Nutzen des funktionalen Ansatzes mithilfe eines *Either*-Objektes in einem Kundenprojekt ist. Dabei soll der Programcode möglichst robust und übersichtlich. Da verschiedene Entwickler in diesem Projekt beteiligt sind soll ein kleines Konzept bei der Migration bestehender Anwendungen auf diese neue Struktur helfen. Es soll auch dazu dienen neue Anwendungen auf dieser Basis zu implementieren.

5.1 Die aktuelle Situation

Im Rahmen dieses Kundenprojektes wurden aktuell über 100 kleinere Anwendungen oder auch Microservices implementiert. Diese Microservices kommunizieren untereinander oder mit externen Systemen via http, wodurch Abhängigkeiten verschiedenster Arten entstehen. Sollte einer dieser Microservices aufgrund eines Fehlers nicht korrekt funktionieren, kann es sich auf andere Microservices unmittelbar auswirken. Dadurch ist eine Fehlersuche sehr aufwendig und nur mit guten Kenntnissen der gesamten Plattform möglich.

Dabei soll der Fokus auf einer bestimmen Gruppe dieser Microservices liegen, den Cronjobs. Ein Cronjob ist eine Funktion von Unix Betriebssystemen, welche von einem Cron-Daemon bereitgestellt wird.[34] Dieser läuft im Hintergrund und führt in zeitlich definierten Abständen bestimmte Aufgaben aus.[35] Auf eine ähnliche Art und Weise funktionieren die Cronjobs auf der Kundenplattform. Hier wird jedoch kein Cron-Daemon genutzt, da die Funktionalität in NodeJS selbst umgesetzt wurde und somit Plattform unabhängig ist. Dieser Cronjob bindet eine Messagequeue von Microsoft Azure an. Er wird genutzt, um asynchrone Prozesse in der Plattform abzubilden.

Der gewählte Cronjob ist dafür verantwortlich E-Mails zu versenden, sollte ein Kunde etwas im Shop bestellt haben oder sollte sich der Status einer bestehenden Bestellung ändern. Wenn ein anderer Microservice eine Änderung an einer Bestellung durchführt, wird eine Nachricht, mit allen Informationen zu dieser Änderung in eine Messagequeue

[34] [Men18].
[35] Vgl. ebenda.

geschrieben. Der Cronjob prüft diese Messagequeue in regelmäßigen Zeitabständen auf neue Nachrichten und bearbeitet sie.

Die Implementierung dieser Cronjobs stellt den Grund für folgende Evaluierung dar. Aktuell werden selbst definierte Fehler geworden, welche vom *Error*-Objekt abgeleitet sind. Teilweise werden sie geworfen, um die direkt wieder zu fangen und in dem betreffenden *catch*-Block zu prüfen, ob es sich dabei überhaupt um einen Fehler handelt. Stellenweise wird also die *throw*-Mechanik zur Manipulation des Kontrollflusses im Programm missbraucht. Die folgende Abbildung zeigt einen Ausschnitt dieses Cronjobs. Die Funktion *processMessage* wird jedes Mal aufgerufen, wenn eine neue Nachricht in der Messagequeue gefunden wurde.

Abbildung 21: Microservice Beispiel - aktueller Stand

```
23  async function processMessage(message) {
24      try {
25          const {
26              body: { resource: { id: orderId } = {} } = {},
27              body: { projectKey } = {},
28              body: messageBody,
29          } = message;
30          validateMessage({ projectKey, orderId });
31
32          const partner = getPartnerByProjectKey(projectKey);
33          const order = await getOrderById(orderId, partner);
34          const orderWithResolvedStates = await getOrderWithResolvedOrderStates(messageBody, order, partner);
35          await sendOrderMailMessages(orderWithResolvedStates, partner);
36          apm.currentTransaction.result = 'success';
37
38          return message.complete();
39      } catch (error) {
40          if (error.code === 'INVALID_MESSAGE' || error.code === 'NO_UPDATE_NECESSARY') {
41              apm.currentTransaction.result = 'warning';
42              return message.complete();
43          }
44          winston.error(`A error occurred with statusCode: ${error.statusCode}, ${error.message}, errorBody: ${JSON.stringify(error.body)}`);
45          winston.error(`order-mail-message-sender: ${error.statusCode}, ${error.message}, errorBody: ${JSON.stringify(error)}`);
46          apm.currentTransaction.result = 'error';
47          apm.captureError(error);
48
49          return message.abandon();
50      }
51  }
```

Diese Funktion bekommt das *message*-Objekt, mit allen Informationen zu der geänderten Bestellung, als Parameter übergeben. Auffällig ist der große *Try/Catch*-Ausdruck, welcher sich durch die ganze Funktion zieht. Von Zeile 25 bis 29 wird das *message*-Objekt ausgepackt, um die darin enthaltenen Attribute direkt verwenden zu können. In Zeile 30 werden zwei dieser Attribute auf ihre Gültigkeit geprüft. Nur wenn diese gültig sind wird der Ablauf der Funktion fortgesetzt, da die *validateMessage*-Funktion eine Ausnahme wirft, sollte die Nachricht nicht valide sein. Ist sie valide gibt die Funktion keinen Wert zurück. Hier ist der erste Nachteil dieser Implementierung

versteckt, denn es sich nicht sofort ersichtlich, dass diese Funktion ein Hindernis für den weiteren Programmablauf darstellt.

In den Zeilen 32 bis 35 werden weitere Variablen gesetzt um anschließend die korrekte E-Mail zu versenden. Die mit dem *apm*-Objekt in Verbindung stehenden Anweisungen in Zeile 36, 41, 46 und 47 sind hier zu vernachlässigen. In Zeile 38 wird eine Funktion aufgerufen, um der Messagequeue mitzuteilen, dass die Nachricht fehlerfrei verarbeitet wurde und sie entfernt werden kann.

Im *Catch*-Block fällt sofort auf, dass der Fehler auf seine Ursache geprüft wird. Sollte er den Code „*INVALID_MESSAGE*“ oder „*NO_UPDATE_NECESSARY*“ enthalten, ist es im Grunde gar kein Fehler, denn es wird anschließend ebenfalls *message.complete* aufgerufen. Was dazu führt das die Nachricht aus der Messagequeue entfernt wird. Das bedeutet, dass der *Catch*-Block Code enthält, der gar nicht zur Fehlerbehandlung gehört. Weiterhin wird der Fehler in ein Log geschrieben und es wird *message.abandon* aufgeruft. Das führt dazu das die Nachricht nicht aus der Messagequeue gelöscht wird. Stattdessen wird sie später erneut abgefragt, um den Vorgang zu wiederholen.

5.2 Konzept

Um die Klassen und Funktionen für ein *Either*-Objekt nicht selbst bauen zu müssen, soll eine Bibliothek namens *crocks* genutzt werden. Diese Bibliothek enthält eine Vielzahl von algebraischen Datentypen und Funktionen, um eine derartige Programmstruktur zu schaffen. Es ist dennoch möglich nur die benötigen Objekte zu importieren, um Speicher zu sparen.

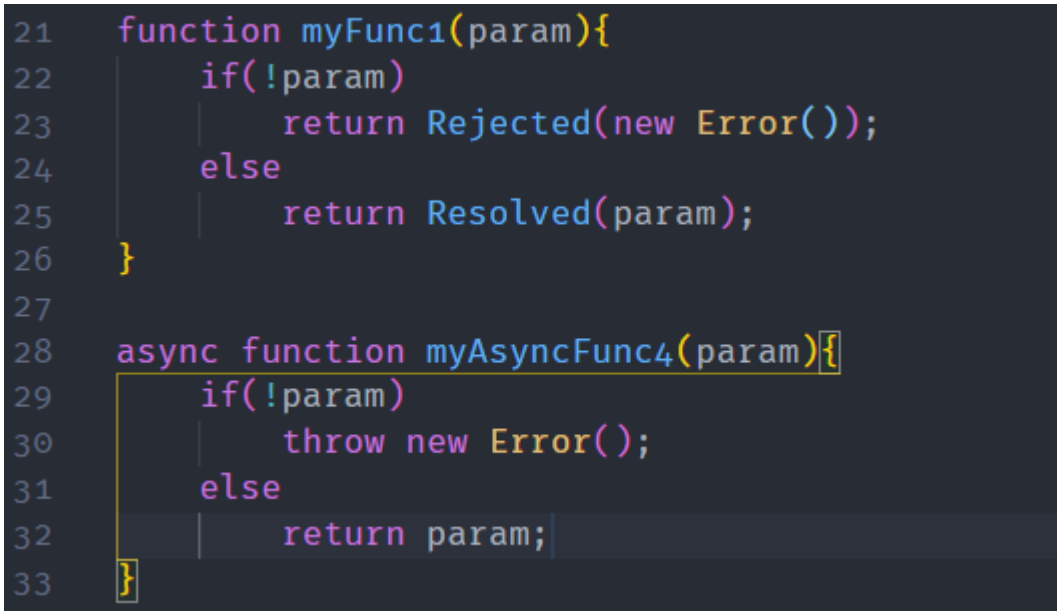
Zunächst muss die Funktion der oberen Ebene des Programms umstrukturiert werden. Alle elementaren Schritte in der Funktion werden mittels der Funktionen *compose*, *chain* und *Async.fromPromise* der *crocks*-Bibliothek verkettet und somit hintereinander ausgeführt. Dabei müssen nur asynchrone Funktionen an Async.fromPromise übergeben werden. Zum Schluss muss der Rückgabewert von *compose* noch mittels *fork* ausgewertet werden. Die *fork*-Funktion ist ebenfalls Teil der Bibliothek und nimmt zwei Parameter entgegen. Diese Parameter sind Referenzen auf Funktionen, die jeweils im Fehlerfall oder im Normalfall ausgeführt werden.

Abbildung 22: Konzept Hauptfunktion - funktionaler Ansatz

```
1  const { Async, compose, chain } = require('crocks');
2
3  const fork = res ⇒ res.fork(
4      handleError,
5      handleSuccess,
6  )
7
8  function main(param){
9      const result = compose(
10         chain(Async.fromPromise(myAsyncFunc5)),
11         chain(Async.fromPromise(myAsyncFunc4)),
12         chain(myFunc3),
13         chain(myFunc2),
14         myFunc1,
15     )(param);
16
17     fork(result);
18 }
```

Zuletzt müssen die verketteten Funktionen (im Beispiel als *myFunc* bezeichnet) selbst angepasst werden. Dabei muss jeweils beachtet werden, ob es sich um eine synchrone oder asynchrone Funktion handelt. Synchrone Funktionen sind hier ein Sonderfall und müssen ihren Rückgabewert in ein *Resolved* oder *Rejected*-Objekt verpacken bevor er zurückgegeben wird. Diese Objekte sind ebenfalls Teil von *crocks*. Sie müssen in jeder Datei, die Funktionen enthält, importiert werden. Asynchrone Funktionen können normal genutzt werden. Es ist besonders wichtig, dass jede Funktion einen Rückgabewert besitzt damit sie den Anforderungen der funktionalen Programmierung entspricht. Der Rückgabewert der ersten Funktion in der Kette ist zugleich der Parameter, mit dem die nächste Funktion der Kette aufgerufen wird. Dadurch verbessert sich die Testbarkeit durch Unittests. Denn Funktionen ohne Rückgabewert sind nur sehr schwer oder gar nicht testbar.

Abbildung 23: Konzept Unterfunktionen - funktionaler Ansatz



5.3 Umsetzung in einem Microservice

Im betreffenden Cronjob wurde die Funktion *processMessage* grundlegend umstrukturiert. Der *Try/Catch*-Ausdruck wurde entfernt und die Funktionen *validateMessage, getPartnerFromProjectKey, getOrderById sowie sendOrderMailMessages und getOrderWithResolvedStates* werden weiter genutzt, jedoch wie im Konzept erwähnt, mittels *compose* miteinander verkettet.

Abbildung 24: Umsetzung der Hauptfunktion - funktionaler Ansatz

```
41  async function processMessage(message) {
42      const context = { message };
43
44      const result = compose(
45          chain(Async.fromPromise(sendOrderMailMessages)),
46          chain(Async.fromPromise(getOrderWithResolvedOrderStates)),
47          chain(Async.fromPromise(getOrderById)),
48          chain(getPartnerByProjectKey),
49          chain(validateMessage),
50          unwrapMessage,
51      )(context);
52
53      (m ⇒ m.fork(
54          () ⇒ {
55              apm.currentTransaction.result = 'error';
56              message.abandon();
57          },
58          () ⇒ {
59              apm.currentTransaction.result = 'success';
60              message.complete();
61          },
62      ))(result);
63  }
```

Die Funktion *unwrapMessage* wurde hinzugefügt, um das Auspacken des *message*-Objekts elegant in einer Funktion zu verpacken und diese in der Kette an erster Position einzureihen. Dabei ist zu beachten, dass die Funktionen in den Zeilen 45 bis 50 von unten nach oben ausgeführt werden. Das heißt *unwrapMessage* wird zuerst und *sendOrderMailMessages* zuletzt ausgeführt. Das hinter der Parameterliste von *compose* in Zeile 51 noch eine Parameterliste folgt, liegt daran, dass *compose* selbst nur eine Referenz auf eine andere Funktion zurückgibt, die mit der zweiten Parameterliste direkt aufgerufen wird. Das Ergebnis dieser zweiten Funktion wird in

Result gespeichert. Auf eine ähnliche Art und Weise funktioniert das Konstrukt in den Zeilen 53 bis 62. Hier handelt es sich um eine sogenannte „*Immediatly Invoked Function Expression“.* Das bedeutet so viel sich selbst aufrufende Funktionsdefinition. Dieses Konstrukt wird hier verwendet, damit das *message*-Objekt direkt in den zwei anonymen Funktionen, die in Zeile 54 und 58 an *fork* übergeben werden, verfügbar ist. Es gibt sonst keine Möglichkeit dieses Objekt als Parameter dorthin zu übergeben. Die obere anonyme Funktion wird im Fehlerfall, die untere im Normalfall aufgerufen. Hier kann also zentral *message.complete* oder *message.abandon* aufgerufen werden, um die Messagequeue über den Verarbeitungsstatus der bearbeiteten Nachricht zu informieren.

Abbildung 25: Umsetzung in Unterfunktionen - funktionaler Ansatz

```
12  function validateMessage(context) {
13      if (!context.projectKey) {
14          const errMsg = ('The message body does not contain a project key field, the message will be deleted.');
15          winston.warn(errMsg);
16          return Rejected(new InvalidMessageError(errMsg));
17      }
18
19      if (!context.orderId) {
20          const errMsg = ('The message body does not contain a resource id field, the message will be deleted.');
21          winston.warn(errMsg);
22          return Rejected(new InvalidMessageError(errMsg));
23      }
24
25      return Resolved(context);
26  }
27
28  function unwrapMessage(context) {
29      const {
30          body: { resource: { id: orderId } = {} } = {},
31          body: { projectKey } = {},
32      } = context.message;
33
34      return Resolved({
35          ... context,
36          orderId,
37          projectKey,
38      });
39  }
```

Nun müssen die Funktionen, die in der Kette aufgerufen werden an die neue Logik angepasst werden. Der, an *compose* übergebene Parameter *context,* wird in folgendem Ausschnitt als Funktionsparameter weiterverarbeitet.

In *unwrapMessage* wird das Objekt lediglich ausgepackt und einige Attribute neu an das Objekt angefügt. In Zeile 14 wird das um *orderId* und *projektKey* erweiterte *context*-Objekt in ein *Resolved*-Objekt verpackt und zurückgegben. In der nächsten Funktion *validateMessage* wird das eben verarbeitet *context*-Objekt wieder in der Parameterliste entgegengenommen, um die eben hinzugefügten Parameter zu prüfen. Neu ist nun, dass diese Funktion immer einen Wert zurückgibt. In der alten Implementierung war das, wie bereits erwähnt, nicht der Fall.

6 Fazit

Der Implementierungsaufwand ist gering, vergleicht man ihn mit dem initialen Aufwand, der benötigt wurde, um den kompletten Cronjob zu implementieren. Außerdem sorgt die neue Struktur für sehr übersichtlichen und robusten Programmcode. Tests haben gezeigt, dass jeder Fehler korrekt durch den linken Pfad geführt wird und letztendlich sauber behandelt werden kann. Sollte der Programmcode später um zusätzliche Funktionen ergänzt werden, muss sich der Entwickler an die vorhandenen Programmstruktur halten und die Funktionskette erweitern. Dafür muss keine weitere Fehlerbehandlung implementiert werden. So kann die Fehlerbehandlung nicht vergessen werden, was im schlimmsten Fall zu einem Programmabsturz führen können. Weiterhin sind die meisten Funktionen pur und somit viel besser im Rahmen von Unittests testbar, was die Codequalität noch weiter erhöht. Andererseits ist es für Entwickler, die aus objektorientierten Technologien kommen sehr gewöhnungsbedürftig mit funktionaler Programmierung umzugehen. Gerade in JavaScript und NodeJS wird eher daten- und objektgetrieben entwickelt. Hier ist wahrscheinlich ein etwas höherer Einarbeitungsaufwand innerhalb des Entwicklerteams notwendig. Ein weiterer Nachteil ist das *context*-Objekt, welches durch die ganze Funktionskette gereicht wird. Jede Funktion benötigt nur wenige Attribute dieses Objektes, trotzdem muss das komplette Objekt immer zurückgegeben und damit der nächsten Funktion übergeben werden, da es später noch einmal gebraucht werden könnte. Das dürfte für einen geringfügig höheren Arbeitsspeicher verbrauch sorgen, dies wurde jedoch im Rahmen dieser Arbeit nicht genauer untersucht.

III Literaturverzeichnis

[Anu18] Anushka, Khattri: „Errors vs. Exceptions In Java“, 2018.
https://www.geeksforgeeks.org/errors-v-s-exceptions-in-java/
Abruf: 2020.09.18

[Chi18] Chidume, Nnamdi: „Understanding Javascript Mutation and Pure Functions“, 2018.
https://blog.bitsrc.io/understanding-javascript-mutation-and-pure-functions-7231cc2180d3
Abruf: 2020.09.20

[Jes17] Jesse, Warden: „Error Handling Strategies“, 2017.
https://dzone.com/articles/error-handling-strategies#:~:text=Various%20error%20handling%20strategies%20can,to%20throw%20your%20own%20errors.&text=They%20have%20helpful%20and%20negative,they%20can%20crash%20your%20program
Abruf: 2020.09.22

[Jon16] Jonathan, Müller: „Choosing the right error handling strategy“, 2016.
https://foonathan.net/2016/09/error-handling-strategy/
Abruf: 2020.09.22

[Lok20] Lokesh, Gupta: „Java Checked vs Unchecked Exceptions“, 2020.
https://howtodoinjava.com/java/exception-handling/checked-vs-unchecked-exceptions-in-java/#:~:text=Remember%20the%20biggest%20difference%20between,runtime%20and%20used%20to%20indicate
Abruf: 2020.09.22

[MAN18] MANISHSANGER: „Java Exception Hierarchy“, 2018.
https://www.manishsanger.com/java-exception-hierarchy/
Abruf: 2020.09.20

[Men18] Mennigen, Marvin: „Was ist ein Cronjob?“, 2018.
https://www.itsystemkaufmann.de/was-ist-ein-cronjob/
Abruf: 2020.09.20

[Mül20] Müller, Frank: „Die Golumne - Netter Versuch: Fehlerbehandlung in Go“, 2020.
https://jaxenter.de/golang/golumne-fehlerbehandlung-in-go-91004
Abruf: 2020.09.20

[Ora19] Oracle: „Unchecked Exceptions — The Controversy“, 2019.
https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html
Abruf: 2020.09.19

[Ora20] Oracle: „Class Throwable“, 2020.
https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html
Abruf: 2020.09.22

[Sin19] Sinclair, James: „ELEGANT ERROR HANDLING WITH THE JAVASCRIPT EITHER MONAD“, 2019.
https://jrsinclair.com/articles/2019/elegant-error-handling-with-the-js-either-monad/
Abruf: 2020.09.19

[W3S20] W3Schools: „Java Exceptions - Try…Catch“, 2020.
https://www.w3schools.com/java/java_try_catch.asp
Abruf: 2020.09.20

[Yaz19] Yazeed Bzadough: „What Is a Pure Function in JavaScript?“, 2019.
https://www.freecodecamp.org/news/what-is-a-pure-function-in-javascript-acb887375dfe/
Abruf: 2020.09.20

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich,

1. dass ich meine Studienarbeit mit dem Thema:

   Fehlerbehandlung in einer Microservicearchitektur

   –

   Ein Ansatz aus der funktionalen Programmierung

   ohne fremde Hilfe angefertigt habe,
2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe und
3. dass ich meine Studienarbeit bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

_____ _____

Ort, Datum

[Anu18] Anushka, Khattri: „Errors vs. Exceptions In Java“, 2018. https://www.geeksforgeeks.org/errors-v-s-exceptions-in-java/ Abruf: 2020.09.18

[Chi18] Chidume, Nnamdi: „Understanding Javascript Mutation and Pure Functions“, 2018.
https://blog.bitsrc.io/understanding-javascript-mutation-and-pure-functions-7231cc2180d3
Abruf: 2020.09.20

[Jes17] Jesse, Warden: „Error Handling Strategies“, 2017.
https://dzone.com/articles/error-handling-strategies#:~:text=Various%20error%20handling%20strategies%20can,to%20throw%20your%20own%20errors.&text=They%20have%20helpful%20and%20negative,they%20can%20crash%20your%20program
Abruf: 2020.09.22

[Jon16] Jonathan, Müller: „Choosing the right error handling strategy“, 2016.
https://foonathan.net/2016/09/error-handling-strategy/
Abruf: 2020.09.22

[Lok20] Lokesh, Gupta: „Java Checked vs Unchecked Exceptions“, 2020.
https://howtodoinjava.com/java/exception-handling/checked-vs-unchecked-exceptions-in-java/#:~:text=Remember%20the%20biggest%20difference%20between,runtime%20and%20used%20to%20indicate
Abruf: 2020.09.22

[Men18] Mennigen, Marvin: „Was ist ein Cronjob?“, 2018.
https://www.itsystemkaufmann.de/was-ist-ein-cronjob/
Abruf: 2020.09.20

[Mül20] Müller, Frank: „Die Golumne - Netter Versuch: Fehlerbehandlung in Go“, 2020.
https://jaxenter.de/golang/golumne-fehlerbehandlung-in-go-91004
Abruf: 2020.09.20

[Ora19] Oracle: „Unchecked Exceptions — The Controversy“, 2019.
https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html
Abruf: 2020.09.19

[Ora20] Oracle: „Class Throwable“, 2020.
https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html
Abruf: 2020.09.22

[Sin19] Sinclair, James: „ELEGANT ERROR HANDLING WITH THE JAVASCRIPT EITHER MONAD“, 2019.
https://jrsinclair.com/articles/2019/elegant-error-handling-with-the-js-either-monad/
Abruf: 2020.09.19

[W3S20] W3Schools: „Java Exceptions - Try…Catch“, 2020.
https://www.w3schools.com/java/java_try_catch.asp
Abruf: 2020.09.20

[Yaz19] Yazeed Bzadough: „What Is a Pure Function in JavaScript?“, 2019.
https://www.freecodecamp.org/news/what-is-a-pure-function-in-javascript-acb887375dfe/
Abruf: 2020.09.20