Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Jan Dennis Bartels

Entwurf einer Software zur Planung von API Abhängigkeiten zwischen Microservices

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Jan Dennis Bartels

Entwurf einer Software zur Planung von API Abhängigkeiten zwischen Microservices

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science, Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

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

Eingereicht am: 18. Oktober 2018

Jan Dennis Bartels

Thema der Arbeit

Entwurf einer Software zur Planung von API Abhängigkeiten zwischen Microservices

Stichworte

Microservices, Schnittstellen, API, Swagger, REST

Kurzzusammenfassung

Inhalt dieser Bachelorarbeit ist die Konzeption einer Software, welche sich dem Problem der Schnittstellen Abhängigkeiten zwischen Microservices widmet. Die Kommunikation zwischen Microservices findet immer über Schnittstellen statt. Hierbei ist es Notwendig, dass beide Services mithilfe der gleichen Daten kommunizieren. Vor allem bei der Änderung von Schnittstellen kann es schnell zu Fehlern kommen, die nicht von Build-Prozessen erkannt werden und so erst in Integrations- oder Produktionsumgebungen zu kritischen Problemen führen. Die zu entstehende Software soll eine Möglichkeit bieten diese Fehler frühzeitig aufzudecken.

Jan Dennis Bartels

Title of the paper

Conception of a Software for the planing of api dependencies between microservices

Keywords

Microservices, interfaces, API, Swagger, REST

Abstract

This document is about conception a software to solve problematic scenarios between the interfaces of a microservice system. Microservices always communicate over interfaces, on this occasion its important that both services use the same datamodel to communicate. When changing an existing interface it’s not always easy to identify those services which consume the interface. Many times those problem are first found on integration or production systems and may lead to system failures. This work will try to solve those problems by identifying them early during development or planning of the interfaces.

Inhaltsverzeichnis

1 Einführung 1
1.1 Motivation . . . 1
1.2 Zielsetzung . . . 1
1.3 Aufbau der Arbeit . . . 2
1.4 Themenabgrenzung . . . 3
1.4.1 API Management . . . 3

2 Grundlagen 5
2.1 Microservices . . . 5
2.1.1 Was sind Microservices . . . 5
2.1.2 Gründe für Microservices . . . 7
2.2 Schnittstellen . . . 8
2.2.1 Application Programming Interfaces . . . 8
2.2.2 Schnittstellen Design . . . 8
2.2.3 Schnittstellentypen . . . 12
2.2.4 REST . . . 12
2.3 Spezifikationen . . . 15
2.3.1 Die OpenApi Spezifikation . . . 15
2.4 Fallbeispiel: Orderservice . . . 16

3 Anforderungsanalyse 19
3.1 Anwendungsfälle . . . 19
3.2 Funktionale Anforderungen . . . 20
3.2.1 Kommandozeilen Anwendung . . . 21
3.2.2 Webanwendung . . . 21
3.2.3 Planungsvorgehen . . . 22
3.2.4 Planung der Anforderungen . . . 22
3.3 Technische Anforderungsanalyse . . . 26
3.3.1 Anwendungskern . . . 26
3.3.2 Konsistenzprüfungen . . . 27
3.3.3 Web-Interface . . . 29
3.3.4 Webanwendung . . . 31
3.4 Nicht Funktionale Anforderungen . . . 32

4 Architektur 34
4.1 Architektursichten . . . 34

4.2 Kontext und Bausteinansicht . . . 34
4.3 Technologieentscheidungen . . . 36
4.4 Komponenten . . . 37

5 Umsetzung **39**
5.1 Anwendungskern . . . 39
5.1.1 Definition der Abhängigkeits-Datei . . . 39
5.1.2 Swagger Parser . . . 45
5.1.3 Prüfkomponente . . . 48
5.2 Kommandozeilen Anwendung . . . 51
5.3 REST Schnittstelle . . . 53
5.3.1 Entstandene Schnittstellen . . . 54
5.4 Web Anwendung . . . 56
5.4.1 Anzeige der Abhängigkeiten . . . 56
5.4.2 Persistenz . . . 56
5.5 Modell Modul . . . 57

6 Test und Auswertung **59**
6.1 Modellieren des Fallbeispiels . . . 59
6.2 Test der Anwendungen . . . 60
6.3 Auswertung . . . 61

7 Fazit und Ausblick **63**
7.1 Ausblick . . . 63
7.1.1 Verlinken von Generierten Swaggerdaten . . . 64
7.1.2 Andere Spezifikationen . . . 64
7.1.3 Erweiterung des Lebenszyklusses der Daten . . . 64
7.1.4 Versionshistorie . . . 64
7.1.5 Zyklische Abhängigkeiten . . . 65

Tabellenverzeichnis

3.1 Repräsentationen: XML und JSON . . . 27
3.2 Schnittstellendefinition des Webinterfaces . . . 31

5.1 Datenmodell für Services . . . 40
5.2 Datenmodell für Schnittstellen . . . 41
5.3 Datenmodell für Daten (1. Ansatz) . . . 42
5.4 Datenmodel für Datenfelder (1. Ansatz) . . . 42
5.5 Datenmodell für die Datenmodelle (final) . . . 43
5.6 Swagger-Konvertierung: Allgemein . . . 46
5.7 Swagger-Konvertierung: Schnittstellen . . . 47
5.8 Swagger-Konvertierung: Daten-Objekte . . . 48
5.9 Umsetzung der Prüfungen . . . 50
5.10 Erweiterte Prüfungen . . . 51
5.11 Entstandene Schnittstellen . . . 55

6.1 Auswertung der funktionalen Anforderungen . . . 62

Abbildungsverzeichnis

2.1 ”Kleinere Einheiten statt monolithische Systeme”, Ursprung: Starke (2014) . . 5
2.2 Fallbeispiel - System-Übersicht 16
2.3 Fallbeispiel - Auszug eines möglichen Sequenzdiagrammes 17

3.1 UseCase Diagramm 20
3.2 Entwurf des Webfrontends 32

4.1 Kontext und Bausteinsicht 35

5.1 Datenmodell der Abhängigkeitsdatei 44

Listings

5.1 Plugin-System der Konsistenzprüfung 49
5.2 Bewertung und Beschreibung der zu prüfenden Eigenschaften 51
5.3 Übersetzungen über RessourceBundle 52
5.4 Spring: Binden von Methoden an Rest-Endpunkte 54
5.5 Lombok-Beispiel: ServiceModel 58

1 Einführung

1.1 Motivation

Der Trend in der Softwareentwicklung entfernt sich zunehmend von der Idee eines Monolithischen Systems hin zu einem auf Microservices basierenden System. Entgegen des Monolithen bauen Microservices darauf, dass einzelne Services Teilaufgaben übernehmen und über Netzwerk-Schnittstellen miteinander kommunizieren. Schnittstellen gibt es in sehr vielen verschiedenen Arten. Häufig werden REST-Schnittstellen oder Message Queues verwendet. Eine große Herausforderung ist es, sicherzustellen, dass alle miteinander verbundenen Services die jeweils anderen Services verstehen, also das die Schnittstellendefinition bekannt ist. Vor allem bei Änderungen an einer Schnittstelle kommt es sehr häufig zu Problemen, die in vielen Fällen erst im Betrieb auffallen. Diese Arbeit soll sich diesem Problem widmen und es soll eine Lösung entstehen, um Schnittstellen und Änderungen an den Schnittstellen zu planen sowie mögliche Probleme frühzeitig entdecken zu können.

1.2 Zielsetzung

Es soll eine Anwendung entstehen, mit der die Abhängigkeiten von Microservice-Schnittstellen untereinander auf Fehler überprüft werden können. Hierbei sollen unterschiedliche Fehlerarten erkannt und bewertet werden. Es sollen unterschiedliche Schnittstellentypen berücksichtigt werden und unter verschiedenen Repräsentationen von Daten unterschieden werden.
Die Anwendung soll sowohl von Entwicklern als von automatischen Systemen, wie zum Beispiel Build-Tools, verwendbar sein.
Eine Visualisierung der Abhängigkeiten untereinander sowie der Probleme soll möglich sein.

1.3 Aufbau der Arbeit

Diese Arbeit ist wie folgt aufgebaut.

Kapitel 1: Einführung
Im ersten Kapitel werden die Motivation und Zielsetzung der Arbeit erläutert. Ebenfalls wird eine Abgrenzung zum Themengebiet des API-Managements vorgenommen.

Kapitel 2: Grundlagen
Das zweite Kapitel behandelt Grundlagenwissen zu Microservices, Schnittstellen und Spezifikationen von Schnittstellen. Es werden Designgrundlagen für Schnittstellen erläutert und ein Fallbeispiel für einen Anwendungsfall der Bachelorarbeit vorgestellt.

Kapitel 3: Anforderungsanalyse
Das Kapitel Anforderungsanalyse behandelt die Anforderungen an die zu entstehende Software sowie die erdachten Anwendungsbereiche und Aufgaben.

Kapitel 4: Architektur
Im Architekturkapitel werden die einzelnen Komponenten der Software genauer definiert, ebenfalls finden sich Architektursichten für die Komponenten und deren Verbindungen.

Kapitel 5: Umsetzung
In diesem Kapitel werden technische Details behandelt, dies schließt konkreten Programmcode zu interessanten Themen ein. Als Hauptteil dieses Kapitels wird das Format für die Speicherung der Abhängigkeiten ermittelt.

Kapitel 6: Test und Auswertung
Als vorletztes wird die entstandene Anwendung einem Test unterzogen und es erfolgt eine Auswertung wie die gesetzten Anforderungen umgesetzt wurden.

Kapitel 7: Fazit und Ausblick
Abschließend wird ein Fazit zur Thematik der Arbeit abgegeben und Ideen für Verbesserungen und mögliche Erweiterungen der Software aufgelistet.

1.4 Themenabgrenzung

Initial war als Themenschwerpunkt dieser Arbeit der Bereich des API Managements geplant. Während der Einarbeitung hat sich der Fokus auf die Planung von API-Abhängigkeiten statt des Managens geändert.
In diesem Teil wird kurz auf den Aspekt des API Managements eingegangen.

1.4.1 API Management

Der Bereich des API Managements ist sehr breit gefächert. Software zum Managen von Web-APIs kann unterschiedliche Aufgabenbereiche übernehmen.
Einige beliebte Aufgaben sind zum Beispiel:

Lastverteilung Aufteilen von Anfragen auf mehrere identische Knoten, um eine hohe Performance gewährleisten zu können.

Sicherheit Grundsätzlich schalten sich Management-Softwares oft zwischen die eigentliche Schnittstelle und den Nutzer. Auf diese Weise kann die Software weitere Sicherheitsaspekte übernehmen. So kann eine token-basierte Zugriffskontrolle, wie beispielsweise OAuth, angeboten oder andere Sicherheitsmaßnahmen hinzugefügt werden.

Versionierung Eine Schnittstelle kann sich während ihres Lebenszyklus verändern. Dies hat zur Folge, dass Nutzer der Schnittstelle ihre Anwendungen anpassen müssen, um Kompatibilität gewährleisten zu können.
Eine Management-Software kann mehrere Versionen eines Mircoservices managen und einem Nutzer so die gewünschte Version weiter zur Verfügung stellen.
Der Nutzer gibt in seiner Anfrage die benötigte Version mit und wird an den entsprechenden Service weitergeleitet.
Anpassungen auf Seiten der API Nutzer entfallen so womöglich komplett.

Caching Eine Management Software kann Antworten zwischenspeichern und auf gleiche Anfragen direkt antworten, ohne den Service aufrufen zu müssen. So wird der Service entlastet.

Transformation Die Management-Software kann sowohl Anfragen als auch Antworten zwischen Nutzer und Service übersetzen und so mehr Kompatibilität erzeugen.

Fassade Sie kann als Gesamtfassade einer Anwendungslandschaft dienen. Nach außen wird ein Einstiegspunkt gegeben, über den alle Anfragen laufen können.

Ein User muss nicht alle Microservices kennen, sondern muss nur wissen, wo sich die Fassade befindet. Diese leitet die Anfragen dementsprechend weiter.

2 Grundlagen

2.1 Microservices

Schon immer gab es in der Softwareentwicklung Ansätze, mit denen man Software besser macht, neue Programmiersprachen oder neue Architekturen. So entstand auch die Idee von Microservices. Die grundlegende Idee hierbei ist, ein System nach Aufgaben aufzuteilen.
Dieses Kapitel liefert einen Einblick in das Thema Microservices und wieso Microservices abhängig von gut definierten Schnittstellen sind.
Die Informationen dieses Kapitels stammen hauptsächlich aus den Werken von Gernod Starke (Starke (2014)), Sam Newman (Newman (2015)) sowie Eberhard Wolff (Wolff (2015))

2.1.1 Was sind Microservices

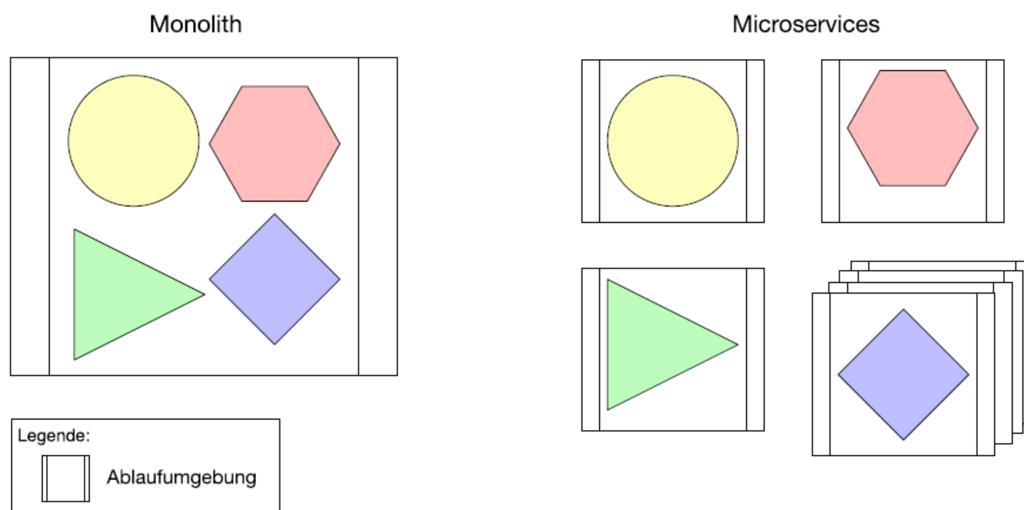

Abbildung 2.1: "Kleinere Einheiten statt monolithische Systeme", Ursprung: Starke (2014)

Microservices sind ein Ansatz zur Modularisierung von Software. Sie bestehen, im Gegensatz zu herkömmlicher Software, aus mehreren Programmen, welche in eigenen Prozessen ausgeführt werden. Wolff (2015)
Um als größeres System funktionieren zu können, müssen Microservices miteinander kommunizieren. Dies geschieht über das Netzwerk und entsprechende Schnittstellen. Die Services sind sich zueinander nur über eine fest definierte API bekannt. Killalea (2016)
Eine einheitliche Definition des Begriffs Microservice gibt es nicht. Grundsätzlich erwartet man von Microservices allerdings folgende Charakteristiken:

Nur eine Aufgabe

Ein einzelner Microservice sollte nur eine Aufgabe übernehmen. Diese sollte er aber komplett abbilden. Die Begrenzung des Services sollte sich an den Komponenten eines Projektes orientieren. Taibi u. a. (2017)
Indem der Service nur eine klare Aufgabe besitzt, wird verhindert, dass der Service unnötig groß wird und damit bleibt der Code klar und verständlich.

Autonom

Jeder Microservice ist eine für sich komplette Anwendung. Sie lässt sich unabhängig von anderen Microservices deployen. Wenn der Code eines Microservices verändert wird, muss nur dieser Service neu gestartet werden und nicht das gesamte System. Newman (2015)
Ein Microservice enthält alle von ihm benötigten Daten selbst. Dies schließt eventuell benötigte Datenbanken oder zumindest getrennte Tabellen in einer gemeinsam genutzten Datenbank mit ein.
Sämtliche Kommunikation findet über Netzwerkaufrufe statt. So wird die Unabhängigkeit der Services sichergestellt. Ebenfalls ist es nicht notwendig, dass alle Services auf derselben Maschine laufen. Im Gegenteil, es ist sogar besser, wenn die Services auf unterschiedlichen Systemen laufen, um eine hohe Leistung sicherzustellen. Einzige Voraussetzung hierbei ist, dass jeder Service alle Services, von denen er abhängig ist, erreichen kann.

Ferner bedeutet Unabhängigkeit auch, dass Konsumenten eines Microservices sich nicht ändern müssen, wenn sich der entsprechende Service ändert.
Voraussetzung hierfür ist, dass die API von Anfang an gut ist. Schnittstellen lassen sich im Nachhinein oft nur schwer ändern, weil sich bereits viele andere Services auf die Schnittstelle verlassen. Änderungen an der Schnittstelle ziehen sich oft durch die gesamte Anwendungs-

landschaft und es ist oft schwer die Folgen einer Änderung einzuschätzen.
Genau diesem Problem widmet sich diese Arbeit.

2.1.2 Gründe für Microservices

Technologien

Da Microservices komplett voneinander getrennt sind, kann jeder Microservice eigene Technologien verwenden.
Dies beinhaltet nicht nur verwendete Frameworks, sondern auch Programmiersprachen, Betriebssysteme und die Infrastruktur der Services. Jeder Service kann die für seine Aufgabe besten Werkzeuge frei wählen, solange die Kommunikation gewährleistet ist.

Ausfallsicherheit

Sollte es zu einem Ausfall eines Services kommen, so ist nur die Funktionalität des jeweiligen Services beeinträchtigt. Der Ausfall eines Microservices hat nicht den Komplettausfall des Gesamtsystems zur Folge.
Ferner können Ausfälle von Microservices einfacher behandelt werden. Die betroffenen Services können unabhängig von dem restlichem System neu gestartet werden. Dieser Vorgang kann sogar automatisiert geschehen.
Sollte der Microservice angepasst werden, um beispielsweise einen Bug zu beheben, so hat dies nicht unbedingt Auswirkungen auf das Gesamtsystem: Nur der fehlerhafte Service muss neugestartet werden, der Rest vom System läuft weiter.

Skalierung

Ein Monolith skaliert gleich. Wenn beispielsweise eine stärkere Hardware eingesetzt wird, so betrifft dies das Gesamtsystem, auch wenn nur wenige Komponenten tatsächlich von der stärkeren Hardware profitieren würden.
Es kann zu Problemen führen, wenn einige Programmteile mehr Ressourcen beanspruchen und dafür andere Teile weniger Ressourcen zur Verfügung haben.
In einem Microservicesystem kann jeder Service für sich skalieren. Services können nach den jeweiligen Anforderungen auf unterschiedlichen Maschinen platziert werden: Services, welche höhere Anforderungen an die Hardware haben, können auf leistungsstärkeren Maschinen laufen und Services, mit geringen Anforderungen, auf einfacherer Hardware.
Ebenfalls ist es sehr einfach neue Instanzen von Microservices zu erstellen, wenn der Bedarf wächst. Es kann ein Loadbalancer eingeschaltet werden, welcher die Anfragen auf mehrere Microservices gleicher Art verteilt.

Deployment
Microservices haben den großen Vorteil, dass sie unabhängig voneinander gestartet werden können. Ein Monolithisches System muss bei Änderungen im Ganzen neu gestartet werden. Wohingegen ein Microservice für sich neu gestartet werden kann, ohne dass die anderen Services beeinträchtigt werden.

2.2 Schnittstellen

Schnittstellen sind für die Kommunikation zwischen Microservices essentiell. Ohne sie könnten Microservices nicht existieren. Microservices bestehen, im Gegensatz zum Monolith, aus mehreren eigenständigen Komponenten. Diese Komponenten sind über Schnittstellen verschiedenster Arten miteinander gekoppelt, um Daten auszutauschen.

2.2.1 Application Programming Interfaces

Ein Application Programming Interface, kurz API, kann man allgemein definieren als einen Programmteil, der von einem Softwaresystem anderen zur Anbindung zur Verfügung gestellt wird. Spichale (2016)
API ist nur ein Oberbegriff für verschiedene API Arten. In dieser Arbeit wird der API Begriff ausschließlich für Remote-APIs beziehungsweise Web-APIs genutzt. Gemeint sind Schnittstellen, welche nicht innerhalb von Programmen, beispielsweise zwischen Programmkomponenten, sondern zwischen in sich abgeschlossenen Programmen stehen. Diese sind über Netzwerkaufrufe zu erreichen und erfüllen einen bestimmten Zweck. Spichale (2016)
Bekannte Arten von Web-Schnittstellen sind beispielsweise SOAP (Lafon u. a. (2007)) oder RESTful HTTP Schnittstellen sowie Messanging Schnittstellen wie AMQP (Advanced Message Queing Protokoll) (Vinoski (2006)) aber auch Remote Procedure Calls (Thurlow (2009)) zählen zu Web-APIs.

2.2.2 Schnittstellen Design

Schnittstellen-Design ist sehr wichtig, denn das Design bestimmt, wie die Schnittstellen genutzt werden. Berjon und Song (2012)
Wie bereits im vorangegangenen Abschnitt erwähnt, sollte das API Design von Anfang an gut durchdacht sein, da sich Änderungen im Nachhinein durch viele Services und Schichten ziehen können.

Basis dieses Abschnittes ist das Werk von Brian Mulloy - Web API Design. Berjon und Song (2012)
Ein paar Prinzipien für gute REST Schnittstellen sind folgende:

Nomen statt Verben

Wenn Ressourcen abgefragt werden, sollten keine Verben in der URL stehen. Statt "/getDogs" sollte einfach nur "/dogs" genutzt werden.
Stattdessen sollten die HTTP Verben wie GET, POST, DELETE und andere genutzt werden. Weitere Info hierzu gibt es im Abschnitt REST.
Ausnahme sind Operationen, bei denen ein übergebenes Objekt verarbeitet und zurückgegeben wird, wie zum Beispiel bei Berechnungen, Konvertierungen oder Übersetzungen.
Wann immer aber eine Ressource nur abgerufen und zurückgegeben wird, sollten keine Verben in der URL vorkommen.

Wenige Endpunkte pro Ressource

Oftmals werden für viele ähnliche Ressourcen viele eigene Endpunkte erstellt.
Einige Beispiele hierfür sind:

```
getAllDogs
getAllDogsExcludingPuppies
getAllBrownDogs
getAllBrownDogsWithoutPuppies
```

Besser ist es nur zwei Schnittstellen pro Ressource zu erstellen:

1. Alle Objekte abrufen:
   ```
   /dogs
   ```
2. Bestimmtes Objekt abrufen:
   ```
   /dogs/1234
   ```

Auf diese Weise bleibt die Anzahl an Schnittstellen überschaubar.
Weitere Eingrenzungen lassen sich über Anfrageparameter realisieren.
Sinnvolle Alternativen zum oben genannten Beispiel könnten so aussehen:

```
/dogs
/dogs?isPuppy=false
/dogs?furColor=brown
/dogs?furColor=brown&isPuppy=false
```

Fehler Behandlung

In REST-Aufrufen lassen sich die Fehlercodes des HTTP Protokolls nutzen. Allerdings sollte darauf geachtet werden, nicht zu viele Fehlercodes zu verwenden. Denn für die Konsumenten der API bedeutet dies, dass sie auf jeden Fehler versuchen zu reagieren. Für viele Fehler ist dies nicht sinnvoll.
In vielen Fällen reicht es drei Arten von Codes zu verwenden. Diese sind:

```
200 - OK
400 - Bad Request
500 - Internal Server Error
```

Viele populäre APIs kommen mit acht bis neun verschiedenen Codes aus. Zusätzlich ist es ratsam eine möglichst genaue Fehlerbeschreibung an die Antwort anzuhängen und dem Klienten so mitzuteilen, was nicht funktioniert hat.

Versionierung

Da sich Schnittstellen ändern können und diese Änderungen sich durch große Teile des Systems ziehen können, sollte man die API mit einer Version kennzeichnen.
Andere Microservices oder sonstige Konsumenten können dann im Aufruf die bekannte API Version angeben. So wird sichergestellt, dass die Antwort verarbeitet werden kann.
Wenn eine neue Version einer Schnittstelle verfügbar ist, sollte die alte Version noch eine Zeit lang unterstützt werden, um den Konsumenten Zeit zu geben, sich an die neue Version anzupassen.
Versionierungen können auf verschiedene Weise realisiert werden. Populäre Ansätze sind Folgende:

URL Versionierung:
Die API Version wird direkt in der Anfrage URL angegeben. Beispiel

```
/v1/dogs
```

Parameter Versionierung:
Die Version wird als Parameter an die Anfrage gehängt. Beispiel:

```
/dogs?version=1
```

Besonders empfehlenswert ist die URL Versionierung. Indem die Version ein zwingend erforderlicher Teil der URL ist, wird sichergestellt, dass keine falsche Version angefordert wird.

Teilauswahl / Paging

Es ist ratsam, bei Schnittstellen, welche eine große Menge an Objekten zurückgeben, diese Menge aufzuteilen. Bei einer wachsenden Menge an Objekten kann es zu Problemen führen, alle Objekte auf einmal zurückzugeben.
Der Vorgang der Aufteilung wird als Paging bezeichnet.
Hierfür könnte man zum Beispiel Parameter übergeben, welche die Menge und die Startposition enthalten.
Beispielsweise:

```
/dogs/?start=25&count=50
```

Ebenfalls ist es gut, Metainformationen wie zum Beispiel die Menge an Objekten, die aktuelle Position und die Menge an aktuell zurückgegebenen Objekten, zu übermitteln.

Ressourcen Typ

Wenn eine Schnittstelle aus verschiedenen Quellen genutzt wird oder gar öffentlich zugänglich ist, empfiehlt es sich, dem Konsumenten der Schnittstelle die Möglichkeit zu geben, die Repräsentationsart des Objektes angeben zu können.
Dies lässt sich beispielsweise auf folgende Arten realisieren:

ACCEPT - Header

Der gewünschte Datentyp lässt sich über den HTTP Header "Accept" angeben. Da REST auf HTTP aufbaut kann für REST-Aufrufe auch HTTP Header genutzt werden.
Beispiel für den Accept Header:

```
Accept:application/json
```

Pseudo Dateityp

Es ist denkbar, den gewünschten Datentyp per Punktnotation anzugeben.
Diese Notation ist allgemein geläufig, da alle Dateisysteme diese Notation für den Datentyp nutzen.
Beispiel:

```
/dogs.json
```

Anfrage Parameter

Über einen Parameter lässt sich der gewünschte Datentyp angeben.
Eine Anfrage kann beispielsweise wie folgt aussehen:

```
/dogs?type=json
```

Ganz egal wie man sich entscheidet, dem Konsumenten der Schnittstelle die Wahl über den Datentypen zu ermöglichen, in der Spezifikation der Schnittstelle sollte beschrieben werden, wie der Datentyp bestimmt werden kann und welcher Datentyp standardmäßig genutzt wird.

2.2.3 Schnittstellentypen

Grundlegend betrachtet gibt es zwei unterschiedliche Arten von Schnittstellen:
Die synchronen und die asynchronen Schnittstellen.

Synchrone Schnittstellen

Unter einer synchronen Schnittstelle versteht man, dass die Schnittstelle bei einer eingehenden Anfrage sofort eine Antwort schickt. Der Konsument blockiert nach der Anfrage in vielen Fällen und wartet auf die Antwort der Schnittstelle.
Ein weit genutztes Beispiel hierfür ist der Schnittstellentyp REST.

Asynchrone Schnittstellen

Im Gegensatz zu den synchronen Schnittstellen gibt es bei den asynchronen Schnittstellen keine direkte Antwort.
Der Sender kann sich nach dem Senden der Anfrage anderen Aufgaben widmen und der Empfänger einer Nachricht muss nicht zwangsläufig eine Antwort schicken.
Populäre Beispiele für asynchrone Schnittstellen sind die sogenannten Messaging Schnittstellen. Bekannte Frameworks sind beispielsweise RabbitMQ oder ApacheMQ und bauen auf Messaging Schnittstellen auf.

2.2.4 REST

REST steht für REpresentional State Transfer und ist ein Architekturstil, welcher am grundlegenden Ansatz des World Wide Webs angelehnt ist.
REST ist zustandslos und synchron. Das bedeutet, dass der Client bei einer Anfrage auf die Antwort wartet und erst nach Erhalt der Antwort mit dem Programmablauf weitermacht. Hieraus ergeben sich Probleme, wenn beispielsweise ein aufgerufener Service weitere Services aufruft oder es auf sonstige Weise zu längeren Verarbeitungszeiten kommt. Der Client wartet nur eine bestimmte Zeit auf seine Antwort und geht nach Ablauf dieser Zeit davon aus, dass eine Antwort nicht mehr kommen wird.

Ressourcen

Einer der wichtigsten Konzepte von REST ist das Konzept der Ressourcen. Eine Ressource ist ein Objekt, das der zugehörige Service kennt. Diese Ressource verfügt über eine URI (Uniform Resource Identifier), diese identifiziert die Ressource global eindeutig.
Ressourcen können über eine feste Menge an Operationen manipuliert werden.
Der Server kann verschiedene Repräsentationen des Objektes erzeugen. Die externe Repräsentation des Objektes ist komplett entkoppelt von der Version, wie der Server sie intern verarbeitet. Ein Client kann beim Aufruf selbst die Version anfordern, die für seinen Aufgabenbereich die beste ist. So können über die gleiche Schnittstelle Ansichten für Mensch und Maschine erzeugt werden. Dies kann für eine HTTP-basierende REST-Schnittstelle beispielsweise über den ACCEPT Header geschehen.

REST & HTTP

REST selbst ist nur ein Architekturstil und kann auf unterschiedlichsten Protokollen aufbauen. Am meisten verbreitet ist allerdings die Umsetzung mit HTTP. Genannt wird sie RESTful HTTP.
In diesem Kontext bedient sich REST den aus HTTP bekannten Verben. Die wichtigsten sind:

- **GET**
  Anfordern einer Ressource
- **POST**
  Erstellen einer neuen Ressource
- **PUT**
  Ändern einer Ressource
- **DELETE**
  Löschen einer Ressource

Auf diese Weise können die gleichen Schnittstellen für unterschiedliche Zwecke genutzt werden und es wird vermieden, dass Endpunkte wie "createDog" erzeugt werden müssen - Die Verben landen nicht in der URI.
Ebenfalls vom HTTP nutzbar sind die HTTP Headers. Mit ihnen lässt sich beispielweise die Repräsentation des Objektes wählen oder die Schnittstelle, um Sicherheitsaspekte zu erweitern. Eine weitere nutzbare Eigenschaft sind die Statuscodes des HTTP. Über sie lässt sich erkennen, ob die Schnittstelle die Anfrage richtig bearbeitet hat, ohne dass der Konsument die Anfrage

vollständig einlesen muss. Im Falle eines Fehlers kann anhand der Codes eine grobe Einschätzung des Fehlerfalles erfolgen.

HTTP bringt bereits eine breite Palette an Tools und Technologien mit. Dies hat den positiven Effekt, dass diese Technologien auch für RESTful HTTP Schnittstellen funktionieren. Beispiele hierfür sind Caching oder Load-Balancer sowie verschiedene Sicherheitstools.

2.3 Spezifikationen

Spezifikationen für APIs gibt es in verschiedenen Formaten von verschiedenen Unternehmen. Einige verbreitete Spezifikationen sind beispielsweise RAML (RAML Workgroup (2017)), API Blueprint [1] und die OpenApi Specification (SmartBear Software (2018)).
Jede der Spezifikationen bietet die Möglichkeit eine API zu dokumentieren. Im Rahmen dieser Arbeit wird allerdings ausschließlich auf die OpenApi Spezifikation eingegangen.

2.3.1 Die OpenApi Spezifikation

Die OpenApi-Spezifikation, ehemals Swagger Spezifikation, ist eine Spezifikation für REST Schnittstellen. Sie ist gleichermaßen von Mensch und Maschine lesbar und erlaubt es somit, verständlich den Zweck sowie die Anforderungen der Schnittstelle zu vermitteln.
Die OpenAPI-Spezifikation enthält unter anderem folgende Informationen:

- Verfügbare Endpunkte sowie verfügbare Methoden für jeden Endpunkt
- Erwartete Eingabeparameter und Ausgabetyp für jede Operation
- Authentifizierungsmethoden
- Allgemeine Informationen zum Service, dazu gehören die Kontaktinformationen sowie Lizenzvereinbarungen unter denen die API genutzt werden darf

Die Spezifikation lässt sich mithilfe von Swagger automatisch für verschiedene Programmiersprachen aus dem Quellcode generieren. Alternativ ist es möglich, die Spezifikation selbst zu schreiben oder sie mithilfe von Tools zu erstellen.

Swagger

Swagger ist ein Open-Source Framework zum Designen und Dokumentieren von REST Schnittstellen. Mithilfe von Swagger können die Funktion einer API sowie die Interaktionsmöglichkeiten mit allen vorhandenen Ressourcen abgebildet werden. Swagger bietet die Möglichkeit online alle Endpunkte einzusehen und diese sogar direkt zu testen. SmartBear Software (2018)

[1] Web: https://apiblueprint.org/

2.4 Fallbeispiel: Orderservice

Viele Schnittstellen und Services, welche abhängig voneinander sind, bringen oft unvorhersehbare Probleme. Ein denkbares Szenario könnte folgende Anwendungslandschaft vorhanden sein:
Als Beispiel sei folgendes Service-System zum Verarbeiten von Bestellungen in einem Onlineshop gegeben:

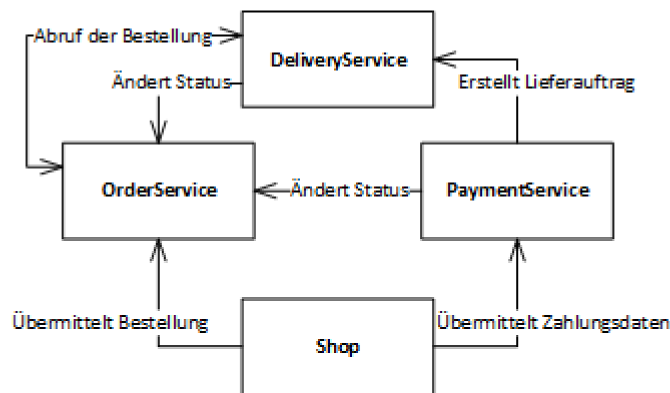

Abbildung 2.2: Fallbeispiel - System-Übersicht

Der Orderservice beinhaltet alle Informationen zu den Bestellungen. Der Paymentservice verarbeitet den Bezahlvorgang und kann bei eingegangener Zahlung dem Deliveryservice einen Lieferauftrag zukommen lassen.
Ein normaler Vorgang könnte wie folgt aussehen:

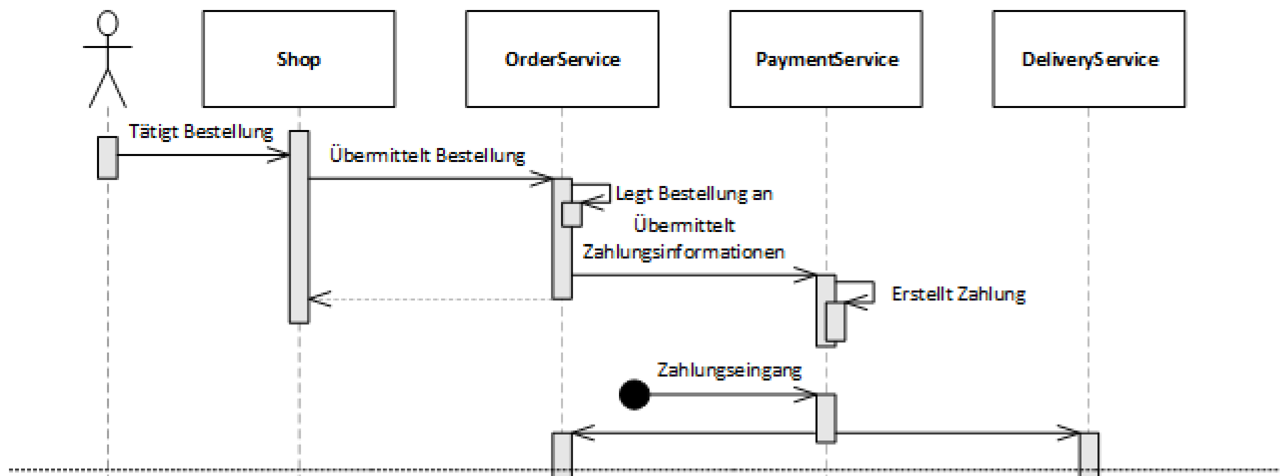

Abbildung 2.3: Fallbeispiel - Auszug eines möglichen Sequenzdiagrammes

Ein Kunde tätigt eine Bestellung

Der Shop übermittelt die Bestellung an den OrderService, dieser ordnet sie dem Kunden zu und ruft den PaymentService auf
Der OrderService übermittelt die Bestellung an den PaymentService, dieser legt eine offene Zahlung an.

Die Zahlung geht ein

Der PaymentService erkennt eine eingegangene Zahlung und signalisiert dem OrderService und dem DeliveryService, dass sich der Status der Bestellung aktualisiert hat. Daraufhin kann die Bestellung versandt werden.

Die verschiedenen Services besitzen jede für sich eine Auffassung wie die Bestellung aussieht. Während dem Shop noch nahezu alle Attribute der Order bekannt sein könnten, reicht es für den Paymentservice aus, die Zahlungsdaten, den Status der Bestellung sowie eine ID zu kennen. Der Deliveryservice benötigt die Zahlungsdaten nicht, dafür aber Liefer- und Rechnungsadresse.

Mögliche Probleme

Angenommen das System funktioniert und es wird nun um eine Retouremöglichkeit erweitert. Die Bestellung kann nun weitere Status besitzen, diese seien "Rücksendung" und "Zurückgenommen".
Oder noch simpler die OrderID wird von einem fortlaufenden nummerischen Wert auf eine alphanummerische ID geändert. In diesen Fällen wird der Orderservice dementsprechend

erweitert. Nun wäre es möglich, dass ein Service als ID nur einen nummerischen Wert erwartet. Daher müssen die abhängigen Services angepasst werden, um Kompatibilität zu gewährleisten. Nicht für jede Änderung müssen alle Services geändert werden. In einem kleinen System wie diesem mag es einfach sein, die abhängigen Services zu finden und die Änderungen zu übernehmen. Aber je mehr mögliche abhängige Services existieren, umso komplexer wird die Anpassung der Services.

Viele dieser Probleme werden nicht während der Entwicklung entdeckt und treten in vielen Fällen erst in einer Produktionsumgebung auf. In diesen Fällen sind die Probleme oft mit Umsatzeinbußen und kurzfristigen Änderungen verbunden.
Ebenfalls kann es vorkommen, dass die Services von verschiedenen Teams entwickelt werden und Änderungen an einem Service nicht ausreichend kommuniziert werden.

Lösung der Probleme

Um derartige Probleme vermeiden zu können, wäre eine Software von Nutzen, welche die Schnittstellen der Services in irgendeiner Weise überwachen könnte und zusätzlich noch Informationen besitzt, welcher Service auf welche Schnittstellen zugreift sowie die Information, welche Daten der jeweilige Service erwartet.
Diese Software könnte dann feststellen, an welchen Stellen im System falsche Daten erwartet werden und Warnungen ausgeben.
Die Software sollte für Entwickler von Microservices eine graphische Oberfläche bieten, über die festgestellt werden kann, ob es problematische Kollisionen bei den API Dokumentationen gibt.
Ebenfalls denkbar wäre eine Art automatisierter Test der Kompatibilität auf Build- oder Integrationssystemen, sodass die Probleme keinesfalls erst im Live-Betrieb auftreten würden.

3 Anforderungsanalyse

In diesem Kapitel werden die Anforderungen an die Software festgelegt. Da die geplanten Anwendungsbereiche sehr unterschiedliche Aufgaben übernehmen werden, werden die Anforderungen in Abhängigkeit zum Anwendungsfall ermittelt.

3.1 Anwendungsfälle

Wie schon im Fallbeispiel erfasst, gibt es zwei wesentliche Anwendungsfälle für die Software:

- Planung per GUI
- Verifikation durch Build-Prozess

Daraus ergeben sich zwei grundsätzliche Benutzer der Software. Zum einem normale User. Dies können Entwickler der API sein. Zum anderen Build-Tools wie zum Beispiel Jenkins (Smart (2011)) oder Travis[1]
Die Anwendungsfälle fallen für die beiden Benutzer unterschiedlich aus, wie im folgendem Use-Case Diagramm visualisiert:

[1] Web: https://travis-ci.org/

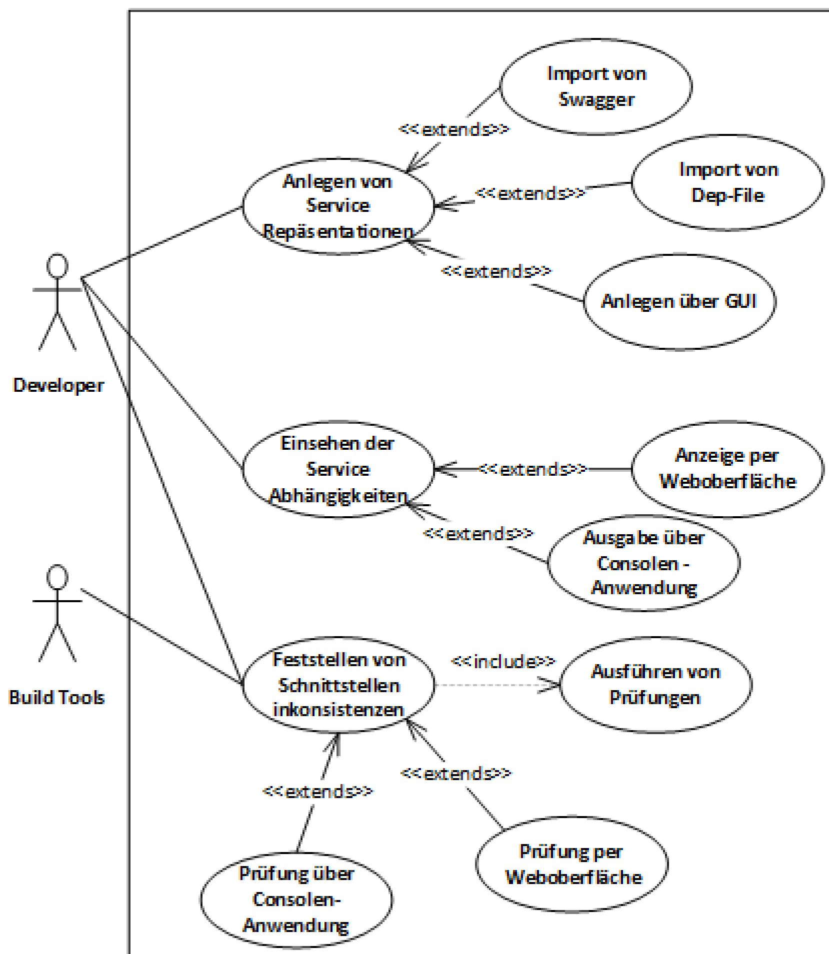

Abbildung 3.1: UseCase Diagramm

Für die Build-Tools ist lediglich die Prüfung der Abhängigkeiten interessant. Aus diesem Grund besteht die Anforderung an die Kommandozeilenanwendung, zumindest die Prüfungen durchführen zu können und dem Build-Tool zu signalisieren, ob Kollisionen aufgetreten sind.

3.2 Funktionale Anforderungen

Als ersten Teil der Anforderungsanalyse werden die rein funktionalen Anforderungen definiert. Durch sie werden die Kernaufgaben des Systems abgebildet. Starke (2014)
In diesem Teil werden ausschließlich Blackbox Anforderungen definiert. Eine verfeinerte, technische Anforderungsanalyse folgt im nächsten Abschnitt.

3.2.1 Kommandozeilen Anwendung

Die Kommandozeilenanwendung sollte die folgenden beiden Aufgaben übernehmen können:

- **Prüfung der Kompatibilität**
- **Auflistung der Abhängigkeiten**

Die Kommandozeilenanwendung ist hauptsächlich dafür gedacht, bei automatisierten Build-Tools eine Prüfung der API-Abhängigkeiten vorzunehmen. Aus diesem Grund sollte sie alle Überprüfungen vornehmen und per Return Code angeben können, ob die APIs kompatibel sind.
Hierbei sollte es möglich sein, die Sensitivität einstellen zu können, so dass in bestimmten Build-Prozessen nur auf gröbere Fehler geprüft wird und an anderer Stelle jeglicher Fehler zum Abbruch des Build-Vorganges führt.
Zusätzlich ist eine Nutzbarkeit der Anwendung für die Entwickler gewünscht: Es soll dem Entwickler möglich sein, über die Konsolenanwendung mögliche Probleme zu erkennen. Außerdem soll es möglich sein, die Abhängigkeiten eines gegebenen Services zu anderen anzeigen zu lassen.

3.2.2 Webanwendung

Der andere Anwendungsfall ist der einer Webanwendung. Die Aufgaben der Anwendung lauten wie folgt:

- **Planen**

  Services, Interfaces & Daten anlegen

- **Visualisieren**

  Gesamtübersicht - Alle Services

  Detailansicht - Einzelner Service

  Vergleichsansicht - Zwei Services

- **Prüfen**

  Ausführung und Auswertung von Konsistenzprüfungen

Im Folgenden Abschnitt wird das Planungsvorgehen erläutert.

3.2.3 Planungsvorgehen

Das Vorgehen dieser Planung setzt sich aus zwei Phasen zusammen. In diesem Vorgang wird ein detaillierter Aufgabenkatalog erstellt, welcher der jeweiligen Anwendung bzw. der Komponente zugeordnet ist.
Ein wichtiger Schritt der Planung ist bereits erledigt. Dies ist die Definierung der Komponenten, der Aufgaben der Anwendungen sowie der drei Hauptaufgaben der Software:
Planung, Prüfung sowie Visualisierung von Abhängigkeiten

Definieren der benötigten Funktionalitäten

Zunächst werden alle Funktionen definiert, ohne hierbei eine Zuordnung vorzunehmen, welcher Anwendung sie zugehören oder in welcher Komponente sie definiert werden. Zu jeder Hauptaufgabe sind schrittweise Unteraufgaben zu definieren, um die Aufgabe zu konkretisieren.

Zuordnen der Funktionen

Nachdem alle benötigten Funktionen definiert sind, werden sie den Programmteilen zugeordnet. Auf diese Weise wird ein Aufgabenkatalog je Programm und Komponente entstehen.

3.2.4 Planung der Anforderungen

Um die Abhängigkeiten der Microservices planen zu können, muss eine Möglichkeit geschaffen werden, Repräsentationen der Services anzulegen.

A1: Microservice-Repräsentationen anzulegen

Es soll möglich sein, Microservice-Repräsentationen anzulegen. Dies beinhaltet einige Teilaufgaben, die unmittelbar dazugehören:

A1.1 Microservice anlegen
: Ein Microservice besteht aus Sicht der Anwendung aus einer Menge von eigenen Schnittstellen. Weitere benötigte Informationen der Services sind Name und Version des Services.

A1.1 Interfaces anlegen
: Um die vom Service angebotenen Schnittstellen zu erfassen, müssen sie für einen Service angelegt werden können.
Die benötigten Informationen für Schnittstellen sollten zumindest Angaben wie Ein- bzw. Rückgabe-Typ, Parameter sowie Schnittstellentyp (REST / AMQP / SOAP oder

ähnliche) beinhalten.
Ferner werden Informationen über abhängige Schnittstellen benötigt, also Schnittstellen, auf die diese Schnittstelle während der Verarbeitung einer Anfrage zugreift.

A1.2 Datenstrukturen definieren
Die Daten, die über die Schnittstellen definiert werden, müssen mit angelegt werden. Für die Daten werden zumindest Informationen über die vorhandenen Datenfelder, also Name und Datentyp, sowie gegebenenfalls Name des Strukturtyps benötigt.
Ferner wären zum Beispiel denkbar:

- Eine Flag, die angibt, ob ein Feld optional ist.
- Eine Art Datenvalidation zum Beispiel in Form eines Regex. Dieser könnte mit anderen Repräsentationen verglichen werden.
- Mögliche Werte, gerade, wenn das Feld nur eine bekannte Menge an Werten annimmt, ist es sinnvoll, diese zu definieren und mit anderen Services zu vergleichen.

A1.2 Datenstrukturenkatalog
Datenstrukturen müssen von mehreren Services verstanden werden. Damit sie nicht an mehreren Stellen angelegt werden müssen, muss eine globale Komponente erschaffen werden, welche die Datenstrukturen verwaltet und ähnliche Datenstrukturen erkennt. Ferner muss es natürlich möglich sein, Datenobjekte aus dieser Komponente auszuwählen, wenn eine Schnittstelle definiert wird.

A1.3 Abhängigkeiten anlegen
Eine Schnittstelle kann, während sie eine Anfrage bearbeitet, weitere Schnittstellen ansprechen. Diese Abhängigkeiten sollten angegeben werden können.
Ferner sollten auch für Services selbst Abhängigkeiten zu Schnittstellen anderer Services angegeben werden können, da ein Service nicht zwangsweise passiv auf Anfragen warten muss. Wenn der Service aktiv Anfragen stellen kann, ohne dass eine seiner Schnittstellen angesprochen wird, so sollen diese Abhängigkeiten für den Service selbst angelegt werden können.

A2 Abhängigkeiten prüfen

Die Hauptaufgabe der Software soll das Erkennen von Problemen zwischen den Schnittstellen der Services sein. Die unterschiedlichen Arten von möglichen Problemen sind hier definiert.

A2.1 Falsche URI
Grundsätzlich ist es denkbar, dass beim Überarbeiten der API die Adresse verändert

wird. Beispielsweise könnte eine Versionierung hinzugefügt oder auf eine neue Version erhöht werden.
Die Anwendung muss sicherstellen können, dass die URI des Services bekannt ist.

A2.2 Falsche Daten

Eines der Hauptprobleme bei API-Änderungen stammt von Änderungen an den übertragenen Daten. Hierbei können sowohl bei der Eingabe der Daten in eine Schnittstelle als auch bei der Rückgabe missverständliche Daten übermittelt werden. Gründe hierfür gibt es reichlich. Ein paar der wichtigsten sind die Folgenden:

- **Erweiterte / verkleinerte Daten**
  Es ist denkbar, dass die Daten einer Schnittstelle noch erweitert werden und die Schnittstelle diese neuen Daten zwangsläufig benötigt, andere Services allerdings diese benötigten Daten nicht übermitteln.
  Es kann auch sein, dass beispielsweise redundante oder unnötige Daten von der Schnittstelle nicht mehr verarbeitet werden und somit die Schnittstelle den "Alten Datensatz" nicht mehr versteht.
  Ferner können die Wertemengen von Datenfeldern verändert werden. Nehme man an, der Typstatus einer Bestellung wird erweitert, um den Status abzubilden, dass eine Sendung in einer Abholstation aufgegeben wurde. So würde dies zu Problemen führen, wenn ein Service diesen neunen Wert nicht verarbeiten kann.
- **Optionale Daten**
  Nicht immer werden von den Schnittstellen alle Daten benötigt. Es ist denkbar, dass Daten übermittelt werden, die nicht zwangsläufig benötigt werden, sobald sie aber vorhanden sind, dennoch Auswirkungen haben.
  Ferner ist es denkbar, dass diese optionalen Daten mit einem Mal Pflichtdaten werden, ohne die keine Verarbeitung mehr möglich ist.
- **Veränderte Formate**
  Viele Datentypen gibt es in unterschiedlichen Formaten. So nutzen unterschiedliche Nationen unterschiedliche Darstellungen für Zeitangaben. Besonders bei Datumsangaben gibt es viele verschiedene Formate.
  Wenn ein Service ein anderes Format für einen Datensatz verarbeiten kann, als bei diesem Service ankommt, so kann die Anfrage nicht verarbeitet werden.

A2.3 Falsche Datentypen

Datenrepräsentationen gibt es viele. Es ist meist nicht erforderlich, dass ein Mircoservice mehrere verstehen kann. Während einige Services bevorzugt das JSON-Format

verwenden, können andere Daten als XML übertragen. Demnach ist es notwendig, sicherzustellen dass die Datenrepräsentation insofern übereinstimmen, dass die benötigten Schnittstellen alle benötigten Formate verarbeiten können.

A2.4 Benötigte Header

Um Services vor unbefugten Zugriffen zu schützen, werden oftmals Authentifizierungstoken übermittelt. Im Kontext des HTTP-REST geschieht dies zumeist über HTTP-Header. Der Service sollte nicht in der Lage sein, konkrete Passworttoken zu überprüfen. Dies ist klare Aufgabe der Services oder etwaigen Sicherheitskomponenten. Dennoch ist es sinnvoll, schon in der API-Planung an Zugriffskontrollen zu denken und diese durch die Software prüfen zu lassen. Die Prüfung sollte nicht explizit auf Authentifizierungs-Header durchgeführt werden. Sondern es sollte möglich sein, benötigte sowie übermittelte Header anzugeben und zu prüfen, ob beide Partner die gleichen Informationen übermitteln beziehungsweise voraussetzen.

A3 Visualisierung

Um dem Nutzer der Software die Probleme sichtbar zu machen wäre eine visuelle Darstellung der Abhängigkeiten sowie der gefundenen Probleme wünschenswert.

A3.1 Ausführung und Visualisierung der Prüfungen

Über die Userinterfaces muss es möglich sein die Prüfungen ausführen zu können und eine Auswertung der Probleme anzeigen zu lassen.

A3.2 Übersicht der Verbindungen

Für Entwickler von Microservice-Systemen ist es wichtig, die Übersicht über das gesamte System zu behalten. Daher wird eine Übersicht gefordert, über die eingesehen werden kann, wie die Services miteinander verbunden sind.
In dieser Übersicht ist es nicht notwendig, aufzuzeigen welche Schnittstellen auf welche anderen Schnittstellen zugreifen. Es ist vollkommen ausreichend auf Service-Ebene die Zusammenhänge anzeigen zu können.

A3.3 Side-By-Side View

Zur Visualisierung der tatsächlichen Abhängigkeiten zwischen den Schnittstellen sollte es eine Möglichkeit geben die Abhängigkeiten zweier Services zueinander einfach zu betrachten. Hierbei sollen die Schnittstellen graphisch dargestellt und die Abhängigkeiten von Schnittstelle zu Schnittstelle gezeigt werden.

3.3 Technische Anforderungsanalyse

Im vorangegangenen Teil wurden die Anforderungen definiert, welche für die Nutzer der Software sichtbar sind. Im Folgenden werden konkrete technische Anforderungen definiert. Diese sind notwendig um die funktionalen Anforderungen umzusetzen.
Die Analyse bezieht sich immer auf einen bestimmten Teil der Software, daher wird in diesem Teil erstmals auf die Komponenten der Software eingegangen.
Ausführlicher werden die einzelnen Komponenten im Folgenden Architektur-Kapitel beschrieben.

3.3.1 Anwendungskern

Der Anwendungskern stellt den größten Teil der Funktionalität zur Verfügung. Daher werden an ihn auch die meisten Anforderungen gestellt.
Die Anforderungen stehen in Abhängigkeit zu externen Daten, der Kern soll folgende Aufgaben umsetzen können:

Bezogen auf einen einzelnen Service:

- Eigene Modelle abrufen
- Abhängige Modelle abrufen
- Abhängige Schnittstellen abrufen
- Konsistenzprüfung des Services zu anderen
- API-depencency einlesen
- API-dependency erstellen

Bezogen auf ein Service-System:

- Konsistenzprüfung des Systems, Ausgabe des Resultates
- Informationen zu allen definierten Services abrufen
- Informationen zu allen definierten Modellen abrufen
- Alle Abhängigkeiten abrufen

3.3.2 Konsistenzprüfungen

Die Hauptaufgabe der entstehenden Software soll die Feststellung von möglichen Problemen zwischen den einzelnen Services sein. Gründe für auftretende Probleme können komplett unterschiedlich sein. In diesem Teil der Analyse wird versucht, möglichst viele mögliche Probleme aufzudecken.

Mögliche Konsistenzprobleme, bezogen auf die DatenModelle

Dieser Abschnitt bezieht sich auf alle möglichen Fehlerarten, die im Zusammenhang mit den übertragenen Daten auftreten könnten.

Inkompatible Inhaltstypen
In REST-Services können die Datentypen unterschiedlich dargestellt werden. Zwei populäre Möglichkeiten sind XML und JSON.

XML Darstellung

```
<?xml version="1.0" ?>
<dog>
  <id>1</id>
  <name>Sammy</name>
  <breed>Havaneser-Malteser Mix</breed>
  <age>5</age>
</dog>
```

JSON Darstellung

```
{
  "dog": {
    "id": "1",
    "name": "Bobby",
    "breed": "Labrador",
    "age": "12"}
}
```

Tabelle 3.1: Repräsentationen: XML und JSON

Beide Repräsentationen können die gleichen Daten enthalten. Möglicherweise kann ein Microservice auch in unterschiedlichen Repräsentationen antworten.
Probleme können aber auftreten, wenn ein Service einen bestimmen Datentyp erwartet und diesen nicht erhält. Um eine Anfrage zu verarbeiten, braucht ein Service bestimmte Daten und andere sind optional. Die Konsistenzprüfung sollte in der Lage sein, die Datenmodelle zu vergleichen und zwischen optionalen und obligatorischen Daten unterscheiden können.

Benötigte Header
REST-Schnittstellen erwarten oftmals HTTP-Header, um ihre Funktionalität an die Anfrage anzupassen. Dies kann beispielsweise die Repräsentation des Datenobjektes betreffen oder zur

Überprüfung, ob der Nutzer berechtigt ist die Schnittstelle zu nutzen, indem eine Authentifizierung vorgenommen wird.
Während der Prüfung sollte auf ggf. angegebene Header Rücksicht genommen und verglichen werden, ob alle benötigten Header übermittelt werden.

Benötigte Daten

Nicht immer werden alle empfangenen Daten verarbeitet oder in anderen Fällen gibt es Daten, die nicht zwingend erforderlich sind, allerdings verarbeitet werden, wenn sie übermittelt wurden. Die Prüfung der Schnittstellen sollte in der Lage sein, zwischen optionalen und obligatorischen Daten zu unterscheiden.

Richtige Daten

Selbst wenn ein Datentyp den Erwartungen entspricht, ist es möglich, dass die Daten nicht kompatibel sind. Beispielsweise kann ein Datum, welches im Normalfall als String übertragen wird, auf der anderen Seite nicht verstanden werden, weil das Datumformat ein anderes ist.
In einem anderen Beispiel könnten Namen auf einer Seite eine andere Schreibweise haben als auf der Empfängerseite.
Um solchen Fehlern entgegenzuwirken, könnten Validierungen der Daten in Form von regulären Ausdrücken oder akzeptieren Werten angegeben werden.
Diese Anforderung ist eher als Anhaltspunkt zu sehen, wie die Software erweitert werden kann.

Mögliche Konsistenzprobleme, bezogen auf die Service und Schnittstellen-Modelle

In diesem Abschnitt werden zu erkennende Probleme definiert, die im Zusammenhang mit den Service- und Schnittstellen-Repräsentationen auftreten können.

Grundlegendes

API Adressierung
: Es muss sichergestellt werden, dass die angegebenen Pfade zu Service und Schnittstelle valide sind und sowohl Anbieter als auch Konsument der Schnittstellen die gleiche URI verwenden.
Problematisch kann es beispielsweise beim Ändern einer Versionierung werden. Probleme dieser Art sind weniger kritisch als Datenfehler. Die Software sollte in der Lage sein einzuschätzen, ob die neue Version einer API mit einer älteren noch kompatibel ist.

Schnittstellen-Art

Es muss geprüft werden, ob die Schnittstellenart (Beispiel: Http oder Message-Queue) übereinstimmt, mit dem was der Service erwartet. Ebenfalls muss die verwendete Methode geprüft werden. Angenommen eine Schnittstelle verwendet PUT, wird aber per POST angesprochen, so würde die Anfrage unbeantwortet bleiben.

3.3.3 Web-Interface

Die Software wird eine Website stellen über die der User Zugriff auf alle Hauptfunktionen erhält. Die eigentlichen Funktionen werden im Backend von einem Web-Interface zur Verfügung gestellt.
Die Schnittstellen zwischen dem User-Interface und dem Anwendungskern sollten den in 2.3.1 beschrieben Designrichtlinien entsprechen, um eine brauchbare API für die Webanwendung sowie möglicher anderer Implementationen bereitzustellen.
Die Anforderungen variieren je nach Schnittstelle, da die Benutzung der Schnittstellen sich stark unterscheidet. Beispielsweise ist die Wahrscheinlichkeit hoch, dass die Service- und Datenschnittstelle in mehreren voneinander unterschiedlichen Szenarien verwendet wird.
Aus diesem Grunde werden die Anforderungen pro Schnittstelle ermittelt.

Konkrete Anforderungen

Geringe Anzahl

Die Anzahl der verschiedenen Endpunkte sollte so gering wie möglich sein. Diese sollten dafür viele Funktionalitäten zur Verfügung stellen. Grundsätzlich sollten Endpunkte für folgende Daten ausreichen:

- /Service Für alle servicerelevanten Infos wie bekannte Modelle für angebotene und abhängige Schnittstellen sowie der bekannten Daten.
- /Schnittstellen Informationen über Schnittstellen, inklusive verwendeter Datenmodelle, zugehörigem Service und abhängiger Schnittstellen.
- /Daten Für alle verwendeten Datenobjekte der Schnittstellen.

Für alle drei Endpunkte sind sowohl die Abfrage mehrerer Elemente sinnvoll als auch individuelle Abfragen einzelner Modelle.
Auf einen Endpunkt für Systeme wird verzichtet. Eine Instanz der Webanwendung verfügt über eine Session, diese wiederum verfügt über ein System.

Filtermöglichkeiten

Es soll möglich sein, die Daten zu filtern. Denkbare Filter sind folgende:

ids

Zur Auswahl einer oder mehrerer expliziter IDs

Abhängigkeiten

Es soll möglich sein Schnittstellen und Datenmodelle, welche von einem bestimmten Service oder einer Schnittstelle benötigt werden, zu finden.
Brauchbar könnten folgende Zusammenhänge sein:

Abhängig von einem Service

Schnittstellenmodelle

Servicemodelle

Abhängig von einer Schnittstelle

Schnittstellenmodelle

Servicemodelle

Hierarchie

Eine weitere Filtermöglichkeit sollte sein, alle Datenmodelle und Schnittstellen abzufragen, welche von einem bestimmten Service bzw. einer Schnittstelle aktiv genutzt werden.
Das beinhaltet implementierte Schnittstellen sowie die davon verwendeten Datenmodelle.
Es ergeben sich folgende Filter:

Genutzt von Service

Daten und Schnittstellenmodelle

Genutzt von Schnittstelle

Datenmodelle

Zusammenfassung

Das Web-Interface sollte wie folgt aussehen:

	Modell	Methoden	Filter
Services /Services	name: String path: String version: String usedInterfaces: List<Interface> providedInterfaces: List<Interface>	• GET • POST • PUT • DELETE	filterById(id) filterDependingFrom(service) filterDependingOn(service)
Schnittstellen /Interfaces	name: String required: String path: String method: String header: List<Header> consumedData: List<DataModel> returnedData: List<DataModel>	• GET • POST • PUT • DELETE	filterById(id) filterDependingFrom(service) filterDependingOn(service)
Daten Modelle /Models	name: String required: Bool format: String possibleValues: List<String> values: List<DataModel>	• GET	filterById(id) filterUsedByService(service) filterUsedByInterface(interface)
Prüfungen ausführen /performChecks	Session ID	• POST	

Tabelle 3.2: Schnittstellendefinition des Webinterfaces

3.3.4 Webanwendung

Die Webanwendung ist eine graphische Anwendung in Form einer Webseite, die über das Webinterface auf die Funktionen des Anwendungskerns zugreift.
Die Webanwendung muss die Anforderungen aus den Anforderungsbereichen A1 (Anlegen von Repräsentationen) und A3 (Visualisierung) erfüllen.
Besonders wichtig sind die Kernfunktionen, also das Anlegen der Abhängigkeiten sowie die Prüfung auf Konsistenzprobleme.

Webdesign

Der planmäßige Entwurf der Webseite sieht wie folgt aus:

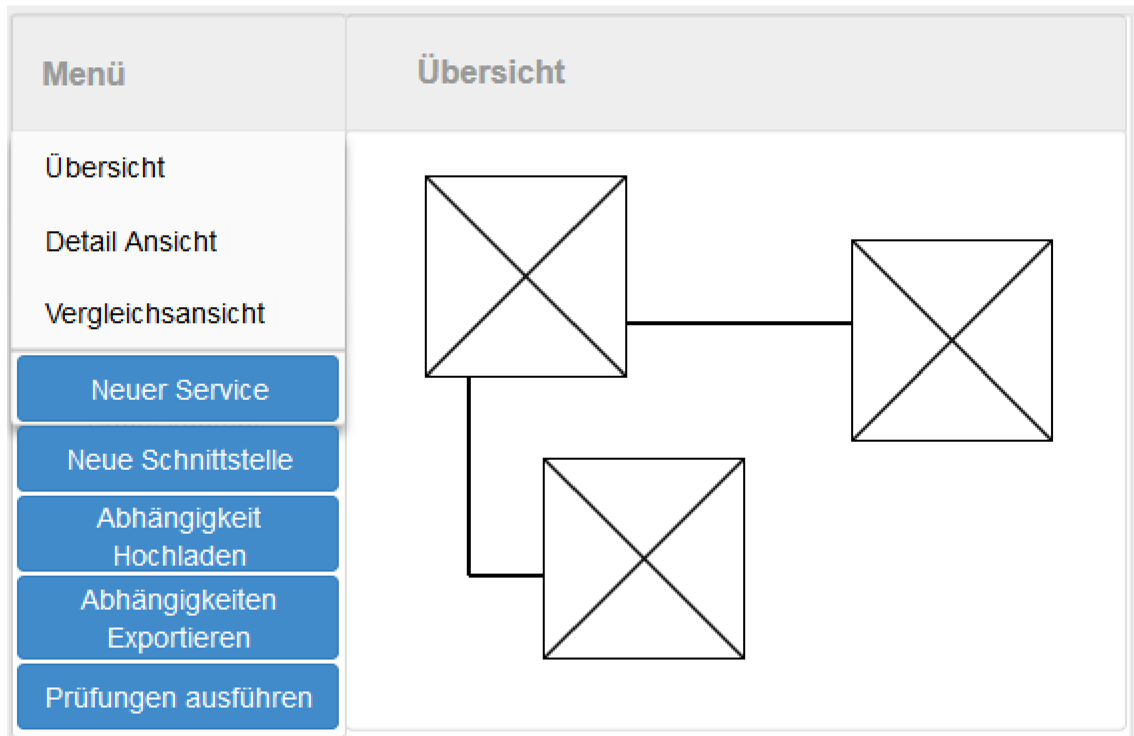

Abbildung 3.2: Entwurf des Webfrontends

Über ein Menü kann der User zwischen den verschiedenen Ansichten wechseln, daraufhin wechselt die Ansicht zu dem gewählten Zustand.
Im Menü befinden sich auch die Funktionen zum Anlegen der Modelle sowie der Ausführung der Prüfungen und dem Import bzw. Export der Abhängigkeitsdaten. Jede dieser Funktionen wird auf einem Popup-Fenster ausgeführt und an das Backend übermittelt.

3.4 Nicht Funktionale Anforderungen

Eine gute Software macht nicht nur die Funktionalität aus, sondern erfüllt auch nicht funktionale Anforderungen wie Qualitätsanforderungen. Starke (2014)
Normalerweise zählen hierzu auch Aspekte wie die Leistung der Software. Beispielsweise wie viele Objekte in einem gewissen Zeitraum verarbeitet werden oder wie performant ein entstehendes System werden soll.
Weitere Qualitätsanforderungen könnten unter vielen anderen Zuverlässigkeit, Nutzbarkeit

und Wartbarkeit der Software bzw. des Programmcodes sein. Starke (2014)
Weder Leistung noch Durchsatz sind Werte auf die für diese Software Wert gelegt wird. Aus dem Grund dass es sich vielmehr um eine prototypische Entwicklung handelt, als dass die Software mit dem Gedanken entwickelt wird, möglichst bald in eine bestehende Anwendungslandschaft integriert zu werden.
Mit diesem Hintergrund lassen sich dennoch einige Aspekte der nicht funktionalen Anforderungen festhalten.

Robustheit

Grundsätzlich gibt es zwei Möglichkeiten der Software Daten zuzuführen:

1. In Form von Importdaten
2. Per Web-Schnittstelle

Nun ist es möglich, dass diese Daten fehlerbehaftet sind. Gründe hierfür könnten folgende sein:
Zum einen ist es denkbar, dass die Daten unvollständig sind oder sie referenzieren Dateien, welche nicht länger vorhanden sind.
Die Software sollte in diesen Fällen nicht abstürzen. Für die fehlerhafte Anfrage sollte der Fehler erkannt und ausgegeben werden. Die restliche Funktionalität muss unbeeinträchtigt bleiben.

Lauffähigkeit

Die Software sollte auf möglichst vielen Systemen laufen. Besonders unter dem Aspekt der Prüfung während der Build-Prozesse muss gewährleistet werden, dass die Software während dieses Prozesses funktioniert.

4 Architektur

4.1 Architektursichten

Ähnlich wie beim Hausbau gibt es auch für den Entwurf von Software verschiedene Ansichten desselben Systems. Jede dieser Sichten dient dem Zweck, eine Eigenschaft der Software visuell darzustellen. Starke (2014)
Im Folgenden wird auf einige der Sichten eingegangen sowie eine Planung aus der jeweiligen Sicht für die zu entstehende Software gezeigt.

4.2 Kontext und Bausteinansicht

In der Kontextabgrenzung wird die Software im Zusammenhang mit der Umgebung gezeigt. Es soll verdeutlicht werden, wie die Software mit Drittsystemen interagiert. Sie soll hierbei sowohl einen technischen als auch einen fachlichen Einblick liefern. Starke (2014)
Die Kontextsicht wird oftmals als oberste Ebene der Bausteinsicht gezeigt. Die Bausteinsicht zeigt in mehreren verfeinernden Schritten die verschiedenen, immer tiefer gehenden Module und Komponenten der Software.
Graphisch dargestellt sieht die Kontextabgrenzung sowie Bausteinsicht der Software, benannt als "Janthir-Anwendung" wie folgt aus:

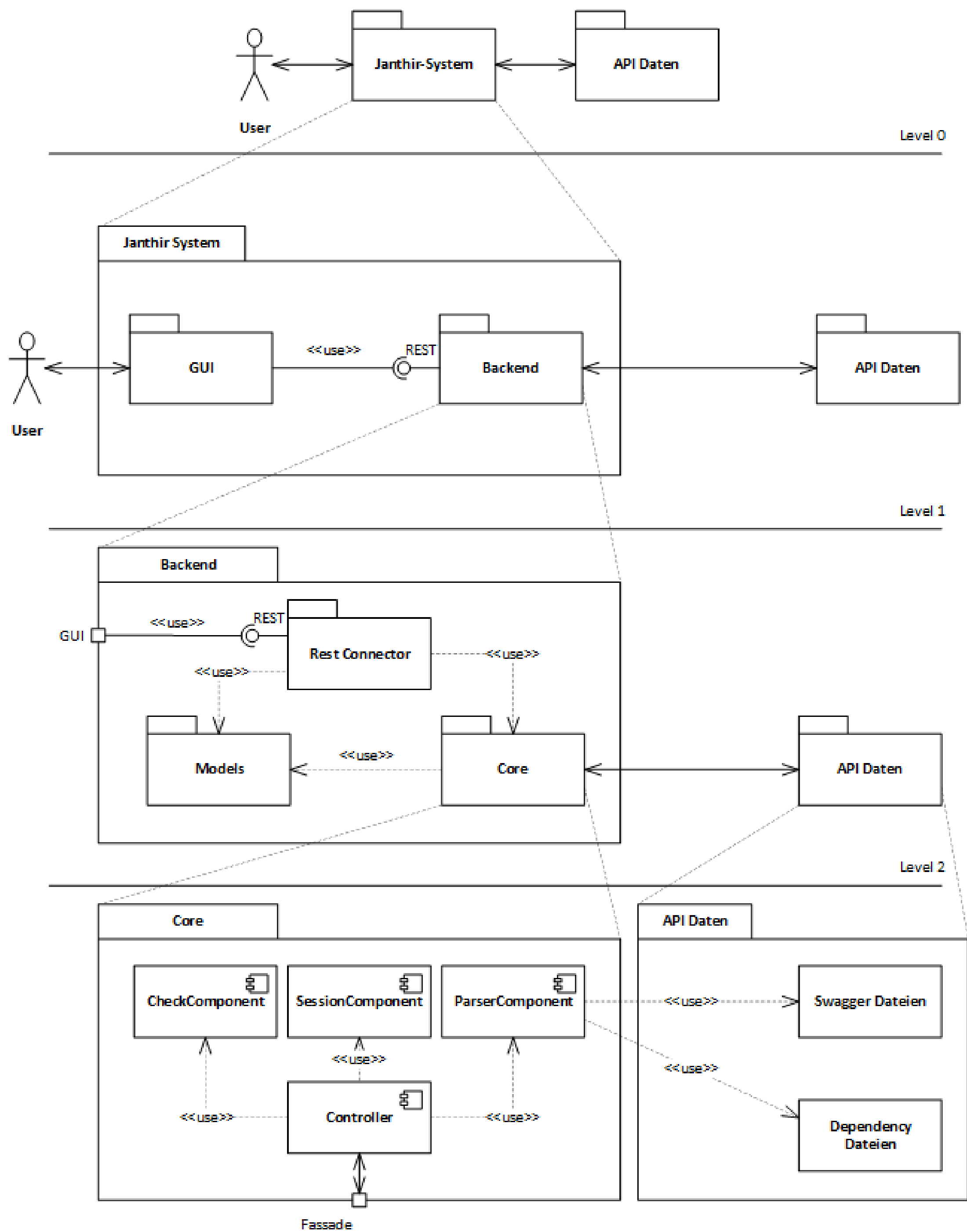

Abbildung 4.1: Kontext und Bausteinsicht

Das oberste Level der Bausteinsicht entspricht der Kontextsicht, es wird gezeigt wie die Anwendung mit anderen Systemen interagiert. In diesem Fall greift die Anwendung lediglich auf API Daten zu. Diese können in eine von zwei Formen vorkommen. Dies wird auf einer tieferen Ebene noch einmal verdeutlicht.
Auf der Ebene 1 wird eine erste Verfeinerung des Systems vorgenommen: Die Anwendung teilt sich in GUI (kurz für Graphisches User-Interface) und Backend auf.
In einem weiteren Verfeinerungsschritt wird das Backend weiter erläutert. Es besteht aus einem Rest-Interface, einem Modul für Modelle sowie des Anwendungskerns.
Auf der letzten Ebene wird der Aufbau des Anwendungskerns gezeigt. Er besteht aus einem Controller und Komponenten für folgende Funktionalitäten:

CheckComponent
Ausführung der Konsistenzprüfungen

SessionComponent
Verwaltung von Session. Eine Session ist eine Menge von Abhängigkeiten, auf die andere Funktionen zugreifen können.

ParserComponent
Eine Komponente zum Einlesen von Abhängigkeitsdaten. Diese können im eigenen Format vorliegen oder aus Swagger generiert werden.

4.3 Technologieentscheidungen

Gute Software macht vieles aus, so auch die Auswahl der verwendeten Technologien einschließlich der verwendeten Programmiersprachen. Verschiedene Technologien sind unterschiedlich gut für spezifische Aufgaben geeignet. Takai (2017)
Schlechte Entscheidungen können sich sowohl in der Qualität der Software als auch schon während der Implementierung der Software bemerkbar machen. Es lässt sich nicht immer leicht erkennen, ob ein genutztes Framework gut oder schlecht ist. So kann es sein, dass eine Software nach jahrelangem Betrieb aufgrund seiner Design- oder Technologieentscheidungen schlecht bis gar nicht erweiterbar bzw. wartbar ist.
Oftmals ist dies der Grund warum Softwareprojekte von Grund auf neu erstellt werden, statt an dem bestehendem Projekt weiterzuarbeiten.

Da die Anforderungen und Anwendungsfälle der einzelnen Komponenten der Software sich stark voneinander unterscheiden, variieren auch die verwendeten Technologien je Komponente

stark. Aus diesem Grund findet im Folgenden Kapitel für einige der Komponenten eine eigene Auflistung der Technologien statt.

Java

Die meisten Teile der Software werden in der Sprache Java entwickelt. Java bietet den bedeutenden Vorteil auf den meisten gängigen Plattformen ohne Anpassungen lauffähig zu sein. Dies wird erreicht indem Java Anwendungen in einer Virtuellen Maschine ausgeführt werden.

Apache Maven

Für die Organisation der Module sowie der jeweiligen Abhängigkeiten wird das Tool Maven von Apache verwendet. Mithilfe von Maven können Java-basierende Projekte gemanagt und gebaut werden. Maven ermöglicht es, einfach Abhängigkeiten zu Frameworks zu laden und neue Versionen automatisch zu integrieren, sofern dies gewünscht ist. Weiterhin ermöglicht Maven das Managen von mehreren Projektmodulen, welche alle über eigene Build-Regeln und Abhängigkeiten verfügen können. Foundation; Analysis (2010)

4.4 Komponenten

Im vorherigen Kapitel wurden bereits die geplanten Komponenten erwähnt. In diesem Kapitel werden diese Komponenten genauer vorgestellt und die Anforderungen an die einzelnen Komponenten geklärt. Die Anwendung setzt sich im Wesentlichen aus vier Komponenten zusammen.

Anwendungskern

Der Anwendungskern deckt einen Großteil der Hauptaufgaben ab und stellt sie anderen Komponenten über ein Interface zur Verfügung. Der Kern ist als Grundimplementierung gedacht, auf der die expliziten Anwendungen aufbauen. Durch diese Kapselung ist es möglich, weitere Anwendungsfälle ohne große Anpassungen im Ursprungscode abzudecken. Weitere denkbare Anwendungsfälle werden in Kapitel 7 als Ausblick angegeben.

Web Interface

Um die Anbindung an das Webprojekt zu ermöglichen, wird eine Webschnittstelle benötigt, die die Kernfunktionalitäten zur Verfügung stellt. Dieses Interface wird in RESTful HTTP entstehen und möglichst den in den Grundlagen vorgestellten Designprinzipien folgen.

Kommandozeilenanwendung

Die Kommandozeilenanwendung ist für automatische Builds gedacht und kann die Konsistenz der Schnittstellen während eines Code-Reviews automatisiert feststellen und anzeigen.
Ferner soll sie nutzbar sein, um die Abhängigkeiten der Microservices zueinander in Textform ausgeben zu lassen.

Webanwendung

Die Webanwendung ist für die Visualisierung und Planung der Schnittstellen gedacht. Sie kann gestartet werden, um die Abhängigkeiten visuell darzustellen und um Konflikte zwischen vorhandenen Schnittstellen einer Microservicelandschaft zu erkennen.
Über die Webanwendung soll es auch möglich gemacht werden, eine Depfile zu erstellen. Diese soll sowohl für einzelne Services als auch in Form einer Gesamtübersicht angelegt werden können.

5 Umsetzung

In diesem Kapitel wird für jede der Hauptkomponenten der Software eine Zusammenfassung der Ergebnisse und Erfahrungen, die während der Umsetzung gesammelt wurden, aufgelistet.

5.1 Anwendungskern

Der Anwendungskern ist die größte Komponente der Software, da sich im Kern alle Grundfunktionen befinden. Um überhaupt die notwendigen Prüfungen durchführen zu können, muss der Kern an alle benötigten Informationen kommen. Hierfür wird zuerst die Abhängigkeitsdatei definiert.
Anschließend wird ein Parser für Swagger Daten erzeugt und eine Komponente, welche alle in der Anforderungsanalyse geforderten Prüfungen durchführen kann.

5.1.1 Definition der Abhängigkeits-Datei

Der Service stützt sich auf eine Datei, welche alle Abhängigkeiten der Services zueinander beinhaltet. Um den Aufbau dieser Datei, fortan unter dem Namen "Dep-File" bekannt, bestimmen zu können, müssen erst einmal die benötigten Daten ermittelt werden.

Benötigte Daten

Um alle benötigten Daten ermitteln zu können, wird von außen nach innen vorgegangen. Im ersten Schritt wird ermittelt, welche Daten von den Microservices benötigt werden.
Daraufhin werden die benötigten Informationen über die Schnittstellen der Services bestimmt.
Schlussendlich wird ein Model der Datenmodelle bestimmt, womit die übermittelten Daten festgehalten werden.

Service
Auf den Service bezogen werden folgende Daten benötigt:
Es wird ein **Name** zur Identifizierung des Services zur Benutzeroberfläche benötigt. Der Service selber wird über eine Art **Basispfad** verfügen. Ebenfalls wären Informationen zu der **Version** des Services sowie möglicherweise der **API-Version** wissenswert.
Der Service besitzt eine Menge an **Angebotenen Schnittstellen** und für die Abhängigkeiten wäre eine Angabe der **Konsumierenden Schnittstellen** vorteilhaft.

So ergibt sich folgende Auflistung der Attribute für die Service Modellierung:

Name	Datentyp	Beispiel (optional)
Name	String	"Dog-Service"
Pfad	String (URL)	"localhost:10080/dogs"
Version	String	"1.0.13"
API-Version	String	"V1"
Angebotene Schnittstellen	Liste:Schnittstellenmodell	

Tabelle 5.1: Datenmodell für Services

Schnittstelle
Bezogen auf die Schnittstelle werden zuallererst Daten über die Schnittstellenart in Form von **SchnittstellenTyp** sowie der **Methode** benötigt. Einige Schnittstellenarten benötigen möglicherweise keine Methode:
Während bei Rest-Schnittstellen die HTTP Methoden wie GET, PUT und POST verwendet werden, gibt es beispielsweise für Schnittstellen wie Messaging Schnittstellen nicht immer eine Methode. In diesem Fall bleibt das Feld leer.
Als nächstes wird die Lokation der jeweiligen Schnittstelle benutzt um Schnittstellen gegeneinander zu identifizieren. Hierfür werden der **Pfad** sowie der **Methode** (sofern gegeben) der Schnittstelle erwartet.
Schließlich werden die **Konsumierten** sowie **Produzierbaren Datenmodelle** benötigt.

Name	Datentyp	Beispiel (optional)
Schnittstellentyp	String	“Rest“ / “Stream“
Pfad	String (URL)	“localhost:10080/dogs“
Methode	String	“GET“ / “POST“
Eingabe-Typ	Liste: Datenmodell	
Ausgabe-Typ	Liste: Datenmodell	
Konsumiert-Von	Liste: Schnittstellenmodell	

Tabelle 5.2: Datenmodell für Schnittstellen

Daten, erster Ansatz
Für die Modellierung der Daten sind zwei Ansätze vorhanden. Der erste Ansatz erwies sich während der Umsetzung als unnötig komplex und wurde vom zweiten Absatz im nächsten Paragraphen ersetzt. Der Vollständigkeit halber wird zuerst auf den ersten Ansatz eingegangen. Grundgedanke des ersten Ansatzes war, dass Daten in einer von zwei Arten auftreten könnten:

Primitive Daten

Reine Basisdatentypen wie Strings, Wahrheits- oder nummerische Werte
Beispiele:

```
1 "OK"
2 "02.03.1998"
```

Zusammengesetze Daten

Datenmodelle mit mehreren Attributen
Beispiele:

```
1 "{name: Tobias, age: 20}"
2 "{Response: {Status: OK, date: 22.11.2017}}"
```

Der Gedanke für die primitiven Daten war, dass sie ohne Namen zurückgegeben wurden und keine weiteren Attribute existieren. Sie treten auf, wenn eine Schnittstelle einen einzelnen nummerischen Wert oder eine einzelne Nachricht zurückgibt.
In dem Namensfeld würde die Art des Datentyps stehen. In der Umsetzung werden verschiedene, häufig genutzte, primitive Typen vordefiniert. Zur Laufzeit sollten keine weiteren primitiven Daten erzeugbar sein, damit eine Grundmenge von Daten verfügbar ist, auf die zusammengesetzte Daten aufbauen können.
In Gegensatz zu den primitiven Daten würden zusammengesetzte Daten einen Namen sowie

eine Menge an Feldern besitzen. Die Felder selber sind im nächsten Schritt definiert.

Name	**Datentyp**	**Beispiel (optional)**
Name	String	“OrderObject“
Felder	Liste: Feldmodell	

Tabelle 5.3: Datenmodell für Daten (1. Ansatz)

Felder:
Der Feld-Typ enthält die Namen der Felder, eine Referenz auf den verwendeten Datentyp sowie die Information, ob das Feld gegeben werden muss. Es ist möglich, dem Feld noch weitere Eigenschaften mitzugeben: Um zu gewährleisten, dass beispielsweise Datumsformate gleich sind, kann als Validierung ein String des Datumsformats angegeben werden. Dieses Format muss für beide Seiten gegeben werden.
Oftmals werden sogenannte Enums als Felder eingesetzt. Enums sind Strings, die einen Wert aus einer Menge möglicher Werte enthalten. Ein Beispiel wäre der Status einer Bestellung: Dieser könnte initial den Wert “Erstellt“ betragen, später wird die Zahlung empfangen und der Status wechselt zu “Bezahlt“.
Hierfür soll es möglich sein, den Wertebereich mitzugeben. Die gegebenen Werte sollten im besten Falle auf beiden Seiten stimmen, es reicht aber auch aus, wenn der Anbieter einer Schnittstelle zumindest alle Werte des Anfragenden versteht, solange ein anderer Status nicht an die Anfragenden zurückkommen kann.

Name	**Datentyp**	**Beispiel (optional)**
Name	String	“OrderStatus“
Datentyp	Datenmodell	
Erforderlich	Wahrheitswert	true
Optional: Validierung	String	“Date:DD:mm:YYYY HH:MM:SS“
Optional: Mögliche Werte	Liste: Strings	“Erstellt,Verschickt,Bezahlt,Abgeschlossen“

Tabelle 5.4: Datenmodel für Datenfelder (1. Ansatz)

Daten, zweiter Ansatz

Im Verlauf der Umsetzung fiel auf, dass dieses ermittelte Datenformat unnötig komplex war. Besonders für die Darstellung von Mengen wie zum Beispiel eines Arrays von komplexeren Daten wurde klar, dass der Schritt auf die Datenfelder anders besser dargestellt werden könnte. Die Daten können entweder primitiv oder zusammengesetzt sein, daher sollten die Modelle auch nur diese beiden Fälle abdecken. Als Änderung wurden die Parameter der Felder auf das Level der Daten angehoben und die Datentypen so verändert, dass sie eine Liste von Daten enthalten konnten.

Konkret sieht die endgültige Version der Modelle wie folgt aus:

Name	Datentyp	Beispiel (optional)
Name	String	"OrderStatus"
Wert	Liste: Datenmodell	
Erforderlich	Wahrheitswert	true
Optional: Validierung	String	"Date:DD:mm:YYYY HH:MM:SS"
Optional: Mögliche Werte	Liste: Strings	"Erstellt,Verschickt,Bezahlt,Abgeschlossen"

Tabelle 5.5: Datenmodell für die Datenmodelle (final)

Dateiaufbau der Abhängigkeit Datei

Das Layout der API-dependency Datei orientiert sich an den vorangegangenen Ergebnissen.

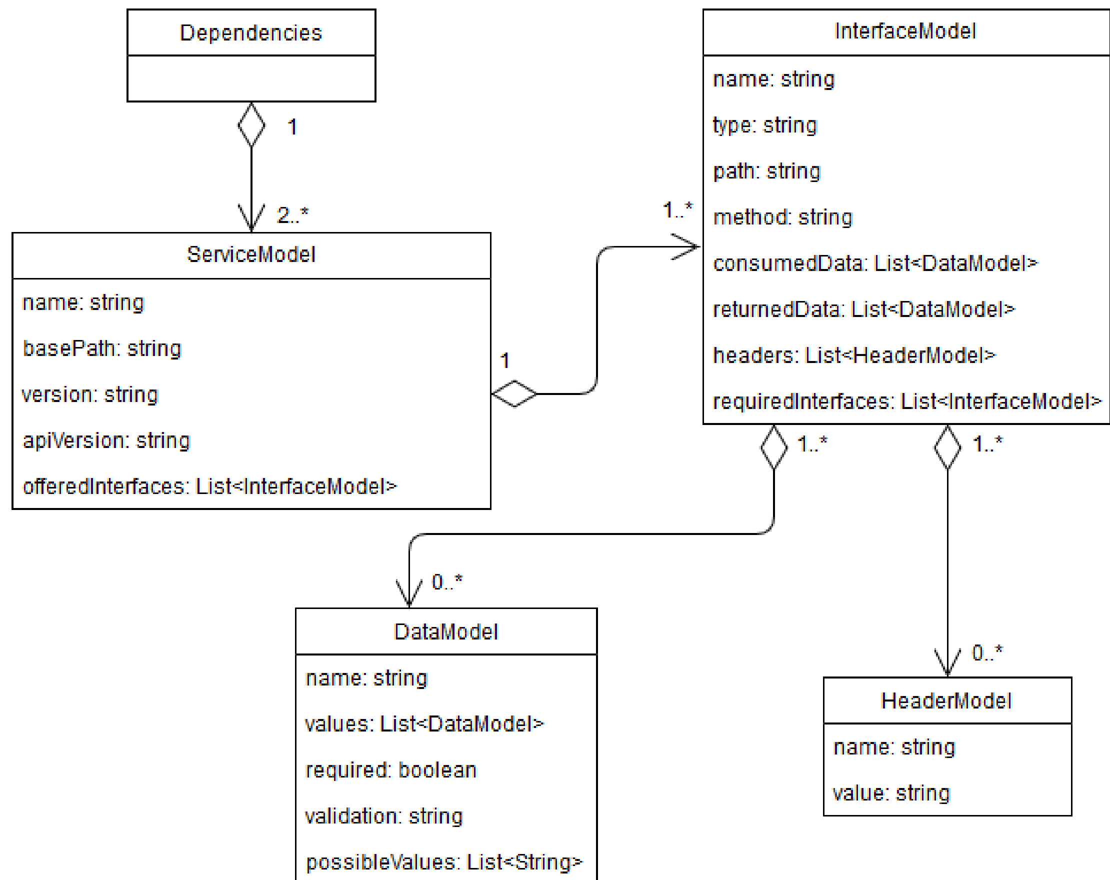

Abbildung 5.1: Datenmodell der Abhängigkeitsdatei

Die Abhängigkeiten werden in Form einer Liste aus Servicerepräsentationen gespeichert. Ein Service-Modell verweist auf alle eigenen Schnittstellen. Diese verfügen neben ihrer eigenen Daten eine Sammlung an erwarteten Schnittstellen.
Ein DatenModell bzw. ein HeaderModell kann zu mehreren Schnittstellen gehören. Eine Schnittstelle kann ein oder Daten und Header verwenden, kann aber auch ohne Daten und Header auskommen.
Eine Schnittstelle gehört immer zu einem Service, ein Service kann mehrere Schnittstellen besitzen.
Ein Service gehört zu genau einer Abhängigkeitsdefinition, in einer Definition sind aber mindestens 2 Services definiert. Es müssen mindestens zwei Services definiert sein damit Abhängigkeiten vorhanden sein können.

5.1.2 Swagger Parser

Der Anwendungskern sollte Swagger Dateien einlesen und eigene Definitionen beziehungsweise eine abstrakte Abhängigkeitsdatei aus ihr erstellen können.
Für diesen Zweck muss der Aufbau der Swagger-Daten analysiert und eine gute Übersetzung zum eigenen Datenformat erstellt werden.

Zielgesteuerte Analyse der Swagger Daten

Um möglichst effizient die erforderlichen Daten zu finden, ist die Analyse der Swagger-Daten rückwärts aufgebaut: Erst wird das Zielformat definiert und diese Felder werden möglichst passend aus dem Swagger-Format entnommen.

Erwartungen

Das Swagger Format wird nicht alle benötigten Daten enthalten. Wäre dies der Fall, würde das eigene Format nicht benötigt. Grundsätzlich wird erwartet, allgemeine Informationen zu dem Service sowie Informationen zu den Schnittstellen und übermittelten Daten zu finden.
Nicht erwartet werden Informationen zu anderen Schnittstellen oder gar Abhängigkeiten unter ihnen. Ebenfalls dürften Datenvalidationen und Nicht-REST Schnittstellen keine Erwähnung finden.
Diese Daten müssen über die Web-Oberfläche an die importierten Datensätze angefügt werden.

Analyse der Swagger Daten

Zuerst werden allgemeine Serviceinformationen wie **Name**, **Version**, **Pfad** und **API-Version** gesucht.
Swagger hält viele Informationen direkt im Hauptknoten des Dokumentes sowie im Infoobjekt. Die API Version findet keine Erwähnung, alle restlichen Felder konnten übertragen werden. Eine Besonderheit ist noch der Pfad, dieser wird von Swagger als Hostname & Basispfad angegeben. Kombiniert erfüllen sie die Erwartungen an das Feld.
Folgende Übersetzung ergibt sich für dem allgemeinen Bereich:

Ziel	**Swagger**
Name	/Info/Title
Version	/Info/Version
Pfad	/Hostname + /basePath
API-Version	**Nicht vorhanden**

Tabelle 5.6: Swagger-Konvertierung: Allgemein

Schnittstellen

Die Schnittstellen sind schon deutlich komplexer als noch die allgemeinen Informationen. Daher wird dieser Bereich aufgetrennt in die Schnittstelleninformationen und die Dateninformationen.
Die Dateninformationen beinhalten Ein- und Ausgabetypen sowie Parameter.

Erwartet werden zunächst Informationen zu dem **Schnittstellen Typ**, dem **Pfad** sowie der **Methode**.
Swagger hält Schnittstellen Daten getrennt. Schnittstellen sind nach Pfad und Operation unterteilt und enthalten eine schwache Referenz auf die jeweiligen Datenobjekte. Diese sind separat definiert. Ferner fiel auf, dass Swagger für jede Schnittstelle einen Namen beinhalten kann. Dies ist keine Anforderung nach der Analyse, erscheint aber als ausgesprochen sinnvoll, weswegen es übernommen wird.
Der Schnittstellentyp ist für sämtliche Schnittstellen der Swagger Dokumentation REST, da Swagger ausschließlich diese abbildet.
Die verwendeten Datenmodelle sind mit eindeutigen Identifikatoren für jede Schnittstelle versehen. Da das eigene Format auch mit Identifikatoren auf Schnittstellenebene arbeitet, müssen die von Swagger verwendeten Identifikatoren in die eigene Variante übersetzt werden. Aus diesem Grund bietet es sich an, in der Implementierung die Datenmodelle vor den Schnittstellen zu importieren und so schon eine Identifikator-Übersetzung vorliegen zu haben.
Swagger hält Ein- und Ausgabetypen nicht voneinander getrennt. Während des Importes muss der Verwendungszweck des jeweiligen Objektes geprüft werden.

Ziel	Swagger
Name	/Paths/(pfad)/(methode)/OperationId
Typ	**REST**
Pfad	/Paths/**(pfad)**/(methode)/
Methode	/Paths/(pfad)/**(methode)**/
Eingabe-Typen	/Paths/(pfad)/(methode)/parameters
Ausgabe-Typen	/Paths/(pfad)/(methode)/parameters

Tabelle 5.7: Swagger-Konvertierung: Schnittstellen

DatenObjekte

Als letztes muss die Übersetzung der DatenModelle ermittelt werden. Das eigene Format definiert sie mit folgenden Eigenschaften: **Name**, **Datentyp** sowie einer Liste an **Feldern**. Felder wiederum haben eigene Eigenschaften, die befüllt werden müssen, sofern das Datenobjekt kein einfacher Typ ist wie beispielsweise Integers, Strings oder andere grundlegende Datentypen. Für die Felder werden Attribute wie **Name**, **Datentyp** und ob das Attribut **optional** ist, erwartet. Die Felder besitzen darüber hinaus noch die Felder **Validierung** für Formate bzw. **Mögliche Werte** welche vermutlich nicht von Swagger unterstützt werden.

Im Swagger Dokument wird zwischen primitiven Datentypen und Datenstrukturen unterschieden. Die Datenstrukturen befinden sich in einem separaten Abschnitt des Dokumentes. In den Schnittstellen wird auf beide Arten lose per Name beziehungsweise Pfad zur Definition im Dokument verwiesen.
Daten besitzen in Swagger die Attribute **name**, **required** sowie **type** für primitive und **scheme** für zusammengesetzte Datentypen. Weitere Attribute sind für die benötigte Übersetzung nicht benötigt.
Die Datenstrukturen sind wie folgt aufgebaut: Sie besitzen einen Typ sowie Felder. Jedes Feld kann unter anderem die Attribute **type**, **format**, **enum**, **description** oder **default** besitzen. Überaus hilfreich sind die Attribute für das Format sowie die Aufzählung der Enumerationen. Erwartet wurden diese nicht, können aber direkt übernommen werden.

So ergibt sich folgende Übersetzung für die Datenobjekte:

Datentyp	Swagger
Name	Name
Datentyp	Übersetzt: type

Die Felder werden wie folgt übersetzt:

Datentyp	Swagger
Name	/**(Objekt)**/
Datentyp	Übersetzt: /(Objekt)/Type
Erforderlich	Enthalten in: /(Objekt)/required
Validierung	/(Objekt)/format
Mögliche Werte	/Objekt/enum

Tabelle 5.8: Swagger-Konvertierung: Daten-Objekte

5.1.3 Prüfkomponente

Für die Konsistenzprüfung der Abhängigkeiten zueinander entsteht eine Komponente. Eine einfache Prüfung auf Wertgleichheit auf die Schnittstellenobjekte wäre nicht ausreichend, denn so würden zwar Fehler in den Schnittstellen gefunden, allerdings wäre nicht feststellbar, welches Attribut der Schnittstelle fehlerhaft ist.
Die aufkommenden Probleme müssen eine Gewichtung erhalten, es wird hierbei zwischen vier Fehlerstufen unterschieden:

Trivial
Fehler welche vermutlich kaum bis gar keine Auswirkungen auf den Betrieb haben würden, werden mit Trivial betitelt. Ein Beispiel hierfür wäre, wenn ein Service mehr Daten schickt als die Schnittstelle akzeptiert, wenn beispielsweise ein Attribut eines Datenmodelles nicht mehr genutzt wird.

Minor
Als Minor, also kleinerer Fehler, werden Probleme eingestuft, die unter Umständen den Betrieb beeinflussen können. Als Beispiel könnte bei einem Pfad oder einem Attributsnamen die Groß- bzw. Kleinschreibung falsch sein.

Major
Alle Probleme, welche mit hoher Wahrscheinlichkeit zu Ausfällen führen würden, werden

als Major-Probleme eingestuft. Hierzu zählen falsche Datenformate, falsche Datennamen und fehlerhafte mögliche Werte für Daten.

Critical

Als Critical werden alle Fehler eingestuft, mit denen der Betrieb eines Services oder einer Schnittstelle nicht mehr möglich wäre. Als Beispiel würden die folgenden Probleme als kritisch eingestuft werden: falscher Pfad einer Schnittstelle, unbekannte Schnittstelle, falscher Datentyp oder falsche Repräsentation (Mime-Type).

Prüfsystem basierend auf Plugins

Die entstehende Anwendung sollte ohne viel Aufwand erweiterbar sein, aus diesem Grund wird für das Prüfsystem mit Hilfe der Komponente org.reflections.Reflections.
Sie ermöglicht es zur Laufzeit alle Implementationen eines Interfaces zu laden. Dies wird genutzt um die Prüfungen der Prüfkomponente zu laden und so die Einbindung neuer Prüfungen obsolet zu gestaltet. Neue Prüfungen müssen nur ein Interface der Prüf-Plugins erweitern und in einem bestimmten Package-Namespace platziert werden. Bei der Erzeugung der Prüfkomponente werden mit folgendem Code alle Prüfungen geladen:

```
private void gatherPlugins() {
  Reflections reflections = new Reflections(...);
  this.checks = reflections.getSubTypesOf(InterfaceCheck.class)
    .stream()
    .map(e -> {
      try {
        return e.newInstance();
      } catch (Exception e1) {
        throw new IllegalArgumentException(...);
      }
    })
    .collect(Collectors.toList());
}
```

Listing 5.1: Plugin-System der Konsistenzprüfung

Ein weiterer Vorteil dieses Systems ist eine bessere Übersicht über alle vorhandenen Prüfungen. Die Prüfungen sind klar voneinander getrennt und erfüllen jede für sich einen bestimmten Zweck.

Prüfungen

Die Anforderungen aus Kapitel 4 wurden in folgenden Prüfungen umgesetzt:

Anforderung		**Prüfung**	**Bewertung**
A2.1	Falsche URI	ServiceCheck	Critical
A2.2a	Zu viel / wenig Daten	ModelCheck	Trivial - Critical
A2.2b	Optionale Daten	ModelCheck	Trivial
A2.2c	Datenformate	ModelCheck	Major
A2.3	Datentyp und Repräsentation	ModelCheck	Major - Critical
A2.4	Header	HeaderCheck	Critical

Tabelle 5.9: Umsetzung der Prüfungen

Einige der Prüfungen wurden noch weiter unterteilt, so ist es als weniger Problematisch eingeschätzt wenn ein Datentyp mehr Felder als nötig besitzt. Ebenfalls wurden für die Datentypprüfungen zwischen dem optional vorhandenem Datenformat (z.B. bei Zeitangaben) und dem Mime-Typen der Schnittstelle unterschieden.
Aus diesem Grund ergeben sich für einige Prüfungen mehrere Bewertungen.

Ferner wurden folgende, vorher nicht definierte Prüfungen umgesetzt:

Anforderung	**Bewertung**
Optional vorgegebene Werte	Major
Groß & Kleinschreibungsfehler bei allen Feldnamen	Minor
Vorkommen von Ähnlichen Datenmodellen	Trivial
Fehlender optionaler Inhalt	Trivial

Tabelle 5.10: Erweiterte Prüfungen

Die konkrete Implementierung der Bewertung der Anforderungen sieht wie folgt aus:

```
1  SERVICE_UNREACHABLE("Service␣unreachable", Critical),
2  SERVICE_UNKNOWN("Service␣unknown", Critical),
3  HEADER_MISSING("Header␣info␣Missing", Critical),
4  DATA_UNNECESSARY("Unnecessary␣Data␣beeing␣send", Trivial),
5  DATA_MISSING("Missing␣Required␣Data", Critical),
6  DATA_MISSING_OPTIONAL("Missing␣optional␣data", Minor),
7  DATA_REQUIREMENT_MISMATCH("Required␣data␣marked␣as␣optional", Major),
8  DATA_WRONG_FORMAT("Wrong␣Data␣format", Major),
9  DATA_WRONG_MIME("Incompatible␣Mime-Type", Critical),
10 DATA_WRONG_TYPE("Wrong␣Data␣send", Critical),
11 DATA_WRONG_NAME("Wrong␣data␣name", Major),
12 DATA_VALUE_MISMATCH("Possible␣Value␣Mismatch", Major),
13 DATA_CASESENSITIVE_NAME("Case␣insensitive␣naming", Minor),
14 DATA_SIMILAR_TYPES("Possible␣mismatch␣due␣to␣similar␣types", Trivial)
```

Listing 5.2: Bewertung und Beschreibung der zu prüfenden Eigenschaften

5.2 Kommandozeilen Anwendung

Die Kommandozeilen Anwendung ist eine einfache Implementierung der Kernfassade für den Gebrauch durch Build Tool. Auf diese Weise lassen sich inkompatible APIs bereits vor Auslieferung der Anwendung feststellen.

Verarbeitung der Programmparameter

Für die Erkennung der Programmparameter wird das Apache Commons CLI Framework[1] verwendet. Es ermöglicht eine einfache Verarbeitung der Programmparameter. Des Weiteren lässt sich gleichzeitig eine ansehnliche Hilfsdokumentation für die Kommandozeile generieren. Für Build Tools wird ein Parameter angeboten, mit welchem das Fehlerlevel zum Abbruch der Builds festgelegt werden kann.

Lokalisierung der Anwendung

Enthalten im Java SDK Umfang ist die Nutzung von Ressourcenbundles [2]. Über sie ist es sehr einfach möglich, verschiedene Übersetzungen zu pflegen. Dazu werden alle Meldungen der Software in eine Datei ausgelagert und per Message-Key identifiziert. Für jede zu unterstützende Sprache kann dann eine Datei angelegt werden, welche dieselben Message-Keys enthält. Im Programmcode kann dann anhand der Systemsprache (oder einer selbst definierten Logik) die jeweilige Sprachdatei geladen werden.
Im Code sieht das Ganze wie folgt aus:

Definieren eines Bundles

```
ResourceBundle.getBundle("Messages", Locale.getDefault());
```

Zugriff auf eine Mitteilung

```
message.getString("help.print.description")
```

Der Dateiaufbau für die Nachrichten der Kommandozeilenanwendung sieht wie folgt aus:

```
1 commander.syntax = janthir [OPTIONS] [FILES]
2 commander.info = Performs all Checks on the given API deps,
3                  you may set a break level to indicate an error to
4                  build processes
5 help.description = Show this help dialog
6 help.break-level.description = Set the level to break the application
7                                ([a]ll / [w]arnings / [e]rrors)
8                                Default: Don't break
9 help.print.description = Print the result of the checks
```

Listing 5.3: Übersetzungen über RessourceBundle

[1] Web: https://commons.apache.org/cli/
[2] Web: https://docs.oracle.com/javase/7/docs/api/java/util/ResourceBundle.html

Weitere Anwendungsmöglichkeiten

Abseits der Nutzung während der Build Prozesse lässt sich die Anwendung auch nutzen, um auf der Kommandozeile die Abhängigkeiten zu prüfen.
Dies beinhaltet die Prüfung der Abhängigkeiten sowie die Ausgabe aller abhängigen Services ausgehend von einem Service.

5.3 REST Schnittstelle

Die Verbindung zwischen dem Anwendungskern und der graphischen Oberfläche geschieht mit REST über http. Die REST Schnittstelle wird aufbauend auf der Kernfassade alle Funktionalitäten des Kerns nach außen anbieten. Auf größere Logiken oder Komponenten wird in diesem Modul verzichtet.

Spring Framework

Für die Entwicklung der Schnittstelle wird das Spring Framework verwendet.
Spring wurde ursprünglich als Alternative zu komplexen Java Technologien, speziell den Enterprice JavaBeans, entwickelt. Walls (2014)
Heutzutage ist Spring ein weit verbreitetes Framework, welche viele Funktionalitäten bietet. Eine der meist genutzten Funktionalitäten von Spring ist die der Dependency Injection, mit deren Hilfe Komponenten zur Laufzeit in andere Komponenten injiziert werden können, ohne dass die abhängigen Komponenten die benötigten Komponenten selbst erzeugen müssen. Dies hat den Vorteil, dass die Implementierung der Komponenten voneinander unabhängiger wird. Prasanna (2009); Walls (2014)
Ferner bietet Spring mit dem Spring MVC eine einfache sowie sichere Möglichkeit Webapplikationen zu entwickelt. So lassen sich REST Ressourcen einfach an Controller anbinden.
Spring MVC basiert auf dem Model-View-Controller Prinzip. Dies ist ein Muster zur Trennung von Software in die Komponenten Datenmodell (Model), Präsentation (View) und Programmsteuerung (Controller).

Erstellung der Schnittstellen

Ein Mapping mithilfe von Spring kann wie folgt aussehen:

```
@PostMapping("/Service")
public ResponseEntity addNewService(
        @RequestParam("session") String session,
        @RequestBody NewServiceModel model) {
  Dependencies deps = sessionHandler.getSession(session);
  deps.addService(ServiceModel.of(model));
  return ok();
}
```

Listing 5.4: Spring: Binden von Methoden an Rest-Endpunkte

Das Beispiel stellt eine Ressource unter der URI "/Service" zur Verfügung, dabei wird ein Query-Parameter mit dem Namen "session" erwartet und Daten in Form des Datentypens "NewServiceModel" erwartet.
Als Rückgabewert liefert die Schnittstelle in diesem Fall nur einen Statuscode.

5.3.1 Entstandene Schnittstellen

Basierend auf den Anforderungen aus Kapitel 3 wurden folgende Schnittstellen erzeugt:

Http-Methode	Pfad	Parameter	Funktion
GET	"/{service}"		Abfragen eines Services
POST	"/services"		Anlegen eines Services
PUT	"/{service}"		Ändern eines Services
DELETE	"/{service}"		Löschen eines Services

(a) Schnittstellenmodell: Services

Http-Methode	Pfad	Parameter	Funktion
GET	"/{service}/interfaces"	ids: String[]	Abfragen von Schnittstellen eines Services
POST	"/{service}/interfaces"		Anlegen einer Schnittstelle für einen Service
GET	"/{service}/dependingInterfaces"	ids: String[]	Abfragen von abhängigen Schnittstellen eines Services
POST	"/{service}/dependingInterfaces"		Anlegen einer Abhängigkeit zu einer Schnittstelle
GET	"/{service}/{interface}"		Abfragen einer Schnittstelle
PUT	"/{service}/{interface}"		Ändern einer Schnittstelle
DELETE	"/{service}/{interface}"		Löschen einer Schnittstelle
GET	"/{service}/dependingInterfaces/{id}"		Abfragen einer Abhängigkeit
PUT	"/{service}/dependingInterfaces/{id}"		Ändern einer Abhängigkeit
DELETE	"/{service}/dependingInterfaces/{id}"		Löschen einer Abhängigkeit

(b) Schnittstellenmodell: Schnittstellen

Http-Methode	Pfad	Parameter	Funktion
GET	"/{service}/{interface}/models"	ids: String[]	Abfragen von Modellen einer Schnittstelle
GET	"/{service}/models"	ids: String[]	Abfragen von Modellen eines Services
GET	"/models"	ids: String[]	Abfragen von Modellen eines Systems

(c) Schnittstellenmodell: Datenmodell

Tabelle 5.11: Entstandene Schnittstellen

5.4 Web Anwendung

Frontend: Html, Bootstrap und Javascript

Für die Darstellung der Benutzeroberfläche wird HTML mit Javascript verwendet. Speziell für das Styling wird das Framework Bootstrap[1] verwendet, ein großer Vorteil von Bootstrap ist die Möglichkeit, mit verhältnismäßig geringem Aufwand ein Design erzeugen zu können, welches auf den meisten gängigen Endgeräten dynamisch dargestellt wird.

Bootstrap

Bootstrap wurde mit dem Gedanken entwickelt, dass Webseiten vor allem auch für mobile Geräte oder Geräte mit kleinen Displays entwickelt werden sollten. Es sollte also nicht im Nachhinein eine eigene Variante der Webseite für mobile Nutzer erstellt werden, sondern die eigentliche Webseite soll auf allen Geräten gut dargestellt werden.
Um das zu ermöglichen gibt es unter anderem das Grid Layout. Mit dem Layout ist es möglich, Elemente an einem Gitter auszurichten. Ferner ist es auch möglich, für verschiedene Geräte verschiedene Einstellungen festzulegen, wie Elemente in diesem Gitter angeordnet werden.

5.4.1 Anzeige der Abhängigkeiten

Der Aspekt der Visualisierung auf der Webseite wurde während der Umsetzung nicht realisiert. Grund hierfür ist, dass die entstandene Software lediglich einem Prototypen entspricht und die Entwicklung einer Visualisierung keinen deutlichen Mehrwert für die Evaluierung der Funktionalitäten gebracht hätte.
Der Hauptzweck dieser Software soll es sein, Probleme bei API Abhängigkeiten zu erkennen. Dies ist auch ohne die graphische Visualisierung möglich, da die erkannten Probleme so ausgegeben werden, dass die Ursache des Problems beschrieben wird.
Dennoch wäre eine Visualisierung eine nützliche Erweiterung der Software und könnte zu einem späteren Zeitpunkt umgesetzt werden.

5.4.2 Persistenz

Eine Speicherung der Daten in Form einer Datenbank wird nicht umgesetzt, aus dem einfachen Grund, dass kein Anwendungsfall der Software diese Technologie sinnvoll nutzen würde.
Sowohl für die Prüfung der Abhängigkeiten durch Build-Tools als auch für die Visualisierung und Prüfung durch die Web-Oberfläche müssen alle Daten eingelesen werden, da nur der aktuelle Stand der Service-Abhängigkeiten geprüft werden muss.

[1] Web: https://getbootstrap.com/

Andere Anwendungsfälle wiederum können sehr wohl von einer Datenbank profitieren. Angenommen eine Microservicelandschaft wird in mehreren Iterationen überdacht, verändert oder erweitert. So würde es sich anbieten die alten Versionen der Anwendungslandschaft zu persistieren, um bei Fehlern auf die alte Version zugreifen zu können und anhand der vorherigen Versionen schnell eine Lösung zu finden.
In diesen Fällen würde sich womöglich eine Graph-Datenbank, welche Abhängigkeiten und Netze besonders gut abbilden kann, anbieten.
Für die aktuellen Anwendungsfälle wäre derartige Technologie nicht sinnvoll eingesetzt.

5.5 Modell Modul

Alle Modelle welche zwischen dem Anwendungskern und der Webanwendung genutzt werden, werden in einem separaten Modul definiert. Dies bietet den Vorteil, dass auf dem Web-Interface aufbauende Komponenten keine Abhängigkeiten zum gesamten Anwendungskern haben.

Lombok

Die Datenmodelle nutzen das Tool Lombok[1], es dient dazu Code per Annotation generieren zu lassen.
Dies bietet zwei wesentliche Vorteile:

Übersichtlicher Code
: Der durch Lombok generierte Code ist standardmäßig nicht sichtbar, das Tool agiert erst während der Kompilierung des Quellcodes.

Schnellere Implementierung
: Indem viele Standard Konstrukte einfach per Annotation der Attribute hinzugefügt werden, spart der Entwickler wertvolle Zeit bei der Entwicklung. Der Entwickler wird auch nicht einfach aus einem Denkprozess geworfen, nur weil dieser für eine Klasse ein Builder-Pattern[2] einfügen möchte.

[1] Web: https://projectlombok.org/
[2] Web: https://en.wikipedia.org/wiki/Builder_pattern

Ein Beispiel ist der folgende Code für Repräsentation von Services im Modell-Modul:

```
@Getter
@Builder
public class ServiceModel {
  private String Name;
  private String basePath;
  @Nullable
  private String version;
  @Builder.Default
  private List< ... > consumedInterfaces = Lists.newArrayList();
  @Builder.Default
  private List< ... > providedInterfaces= Lists.newArrayList();
}
```

Listing 5.5: Lombok-Beispiel: ServiceModel

Lombok generiert für die ServiceModel Klasse folgendes:

- Getter Methoden für alle Attribute
- Ein Builder Patter, inklusive Default-Werte für die Felder ProvidedInferfaces sowie ConsumedInterfaces

6 Test und Auswertung

Nachdem die Software realisiert wurde folgt nun ein Test der Anwendung. Für diesen Test wird das Fallbeispiel in der Janthir Software erstellt und daraufhin einige mögliche Fehlerszenarien provoziert.

6.1 Modellieren des Fallbeispiels

Das Fallbeispiel besteht aus 4 Services. Wie in der folgenden Grafik aus dem Grundlagenkapitel zu erkennen:

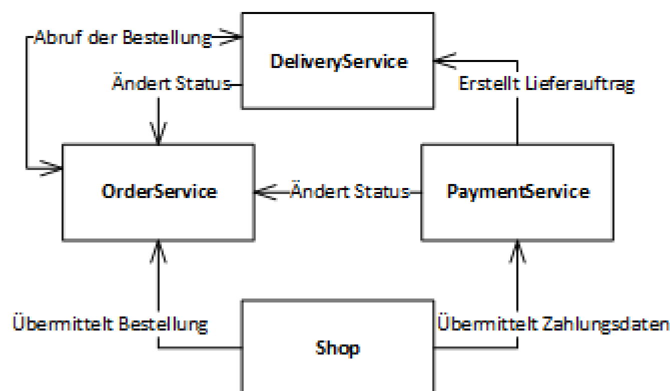

Abbildung 2.2: Fallbeispiel - System-Übersicht

Shop
: Über den Shop kann der Kunde eine Bestellung aufgeben. In diesem Beispiel hat der Shop nur eine Verbindung zum OrderService um Bestellungen zu übermitteln.

OrderService
: Der OrderService hält die Bestellungen, er verfügt über eine Schnittstelle zum anlegen von Bestellungen und zum Abfragen von Bestellungen.
Ferner besitzt er noch weitere Schnittstellen, für diesen Test allerdings werden nur ein kleiner Teil des Systems abgebildet.

DeliveryService
Der DeliveryService kann einen Lieferauftrag erzeugen und den Status im OrderService aktualisieren.

PaymentService
Der PaymentService hält alle Informationen zu Zahlungen, er kann auf Zahlungseingänge reagieren und daraufhin dem DeliveryService signalisieren eine Lieferung zu erzeugen.

Für den Test werden Folgende Modelle erzeugt:

OrderService
Schnittstellen

POST: orderservice/, Body: OrderModel

Abhängig von

PaymentService - Zahlung übermitteln

PaymentService
Schnittstellen

POST: paymentservice/, Body: PaymentModel

Abhängig von

DeliveryService - Lieferung beauftragen

OrderModel
Felder
ID: String
Staus: String
PayInfo: PaymentModel

6.2 Test der Anwendungen

Über die Webanwendungen ist es möglich die Modelle auf der graphischen Oberfläche anzulegen. Nachdem die zuvor definierten Modelle angelegt wurden und auf Probleme geprüft wurde, wurde das Ergebnis ausgegeben dass keine Probleme erkannt wurden.

Provozierte Szenarien

Als nächstes wurden ein paar absichtliche Fehler verbau und das Ergebnis geprüft:

Numerische OrderID

Das OrderModel besitzt im Beispiel einen String als Datenformat, testweise wurde hieraus ein Integer im PaymentService.
Die Prüfung kommt somit zu folgendem Ergebenis:

```
Violation(
  type=DATA_WRONG_TYPE,
  cause=POST: paymentservice/add
  data: Order.OrderID)
```

Daraus ist abzuleiten dass die Erwartungen an den Paymentservice nicht dem entsprechen was die Schnittstelle tatsächlich erwartet. Aus der Ausgabe lässt sich auch erkennen dass der Fehler aus einem Falschem Datentypen im Objekt Order->OrderId stammt.

Neuer Bestellstatus

Das OrderModel besitzt im Beispiel die möglichen Status "initial" und "paid", die Schnittstelle wird jetzt um den Status "delivered" erweitert. Dem abhängige Service fehlt diese Information jedoch.
Die Prüfung kommt somit zu folgendem Ergebenis:

```
        Violation(
        type=DATA_VALUE_MISMATCH,
        cause=POST: paymentservice/add
        data: Order.Status)
```

Es wurden in Form von Unit-Tests noch andere Testfälle abgedeckt um zu gewährleisten dass die Prüfungen funktionieren.

6.3 Auswertung

Nicht alle der in Kapitel 3 definierten Anforderungen wurde im Prototypen umgesetzt. Im Folgenden findet eine Auflistung aller Anforderungen statt, aus ihr geht hervor welche der Anforderungen umgesetzt wurden.

Use-Case Anforderungen

Funktionale Anforderung	Umsetzung
Kommandozeile: Prüfung	Umgesetzt
Kommandozeile: Auflistung der Abhängigkeiten	Umgesetzt
Webanwendung: Planung, anlegen von Modellen	Umgesetzt
Webanwendung: Visualisierung - Gesamtübersicht	Nicht umgesetzt
Webanwendung: Visualisierung - Detailansicht	Nicht umgesetzt
Webanwendung: Visualisierung - Vergleichsansicht	Nicht umgesetzt
Webanwendung: Prüfung und Auswertung	Umgesetzt

Tabelle 6.1: Auswertung der funktionalen Anforderungen

Die Anforderungen wurden zum Teil umgesetzt. Die Hauptaufgabe der Anwendungen wurde erfüllt, dies ist die Prüfung der Abhängigkeiten zwischen Schnittstellen.
Nicht erfüllt wurden die Anforderungen um eine Visualisierung des Systems und dessen Abhängigkeiten (Aufgabenbereich A3).
Allerdings wurde alle notwendigen Voraussetzungen erfüllt: Es wurde eine Webschnittstelle implementiert über die man die notwendigen Daten zur Visualisierung erhält.

Andere funktionale Anforderungen

Im folgendem wird kurz auf die restlichen nummerierten Anforderungen eingegangen:

A1, Anlegen von Modellen
Es ist möglich Modelle von Services, Schnittstellen und Daten sowie Abhängigkeiten zwischen Schnittstellen anzulegen.
Nicht umgesetzt wurde der Datenstrukturenkatalog: Der Code hierfür ist noch vorhanden, da er in einer früheren Version ein Bestandteil der Anwendung war. Allerdings stellte sich die Nutzung des Kataloges und damit verbundene die loose Referenzierung der Daten über IDs als unnötig komplex herraus. Imfolge dessen wurde die Funktionalität entfernt, sie wird nicht länger benötigt.
Über die Webschnittstelle ist es weiterhin möglich alle Modelle abzufragen.

A2, Prüfungen
Alle der geforderten Prüfungen wurden implementiert, darüber hinaus sind noch einige Verfeinerungen hinzugekommen. Genauere Informationen über die umgesetzten Prüfungen sind in Abschnitt 5.1.3 (Prüfkomponente) nachzulesen.

7 Fazit und Ausblick

Abschließend wird ein kurzes Fazit zur Software und dessen Nutzbarkeit gegeben.

Nutzbarkeit der Anwendung

Die Anwendung kann helfen, Probleme zwischen den Schnittstellen von Services zu erkennen. Hierbei ist der Anwendungsbereich nicht nur auf Microservices beschränkt.
Allerdings müssen hierfür immer die API Abhängigkeiten gepflegt werden. In der aktuellen Form besteht keine Möglichkeit, dass die Software die API Daten automatisch erzeugt oder gar die tatsächlichen Schnittstellen abfragt.
Dies hat den bedeutenden Nachteil, dass die Prüfungen nicht richtig ausgeführt werden, sofern die Daten nicht bei jeder Änderung gepflegt werden.
Wenn die Software für ein Softwareprojekt nutzbar gemacht werden soll, müssen Mechaniken entwickelt werden, um automatische Tests zu erlauben. Eine Möglichkeit hierfür wäre das Zurückgreifen auf generierte API Spezifikationen. Zu diesem Thema kommt im Ausblick Teil dieses Kapitels noch ein Abschnitt.

Abschluss

Abschließend lässt sich festhalten, dass die Probleme von API Abhängigkeiten nicht leichtfertig genommen werden sollten. Mit dem aktuellem Trend der Softwareentwicklung, immer mehr Microservicesysteme zu entwickeln und auf Web-APIs zuzugreifen, wird die Problematik der Web-API Abhängigkeiten zunehmend zu einem Problem.
Die Software kann bei korrekter Benutzung helfen, diese Probleme zu vermeiden.

7.1 Ausblick

Ziel der Arbeit war es, prototypisch eine Software zum Erkennen von möglichen Problemen zwischen dem erwarteten und den tatsächlichen Schnittstellen von Services zu erschaffen. Da während der Entwicklung nicht jede Anforderung bzw. jede Idee umgesetzt wurde, gibt es noch viele Möglichkeiten, um die Anwendung zu erweitern.

Abschließend sind hier einige Ideen zur möglichen Erweiterung des Prototypen, die während der Entwicklung der Software aufgekommen sind.

7.1.1 Verlinken von Generierten Swaggerdaten

Mithilfe von Swagger lassen sich die API Spezifikationen auch zur Laufzeit aus dem Quellcode generieren. Diese Daten kann Swagger als Rest Endpunkt anbieten.
Denkbar wäre eine Erweiterung der Software um Swagger Endpunkte. Die gespeicherten Daten würden dann keine Informationen zu den tatsächlichen Schnittstellen enthalten, sondern nur noch, welche Schnittstellen erwartet werden. Die Kompatibilitätsprüfung könnte dann direkt gegen die Swagger Endpunkte gehen.
Dieser Schritt vereinfacht das Pflegen der Abhängigkeiten.

7.1.2 Andere Spezifikationen

Aktuell werden nur Swagger-Spezifikationen akzeptiert. Eine einfache Erweiterung wäre die Implementation von weiteren Parsern für Spezifikationen von beispielsweise API-Blueprint oder RAML.
Obwohl Swagger bzw. OpenApi das wohl am weitesten verbreitetste Format ist, werden andere Spezifikationen ebenfalls häufig eingesetzt.

7.1.3 Erweiterung des Lebenszyklusses der Daten

Bisher wurden nur die Verbindungen zwischen Services sowie die Abhängigkeiten zwischen Schnittstellen berücksichtigt.
Denkbar wäre den weiteren Weg einer Anfrage abzubilden. Ein Beispiel hierfür wäre die Berücksichtigung von Datenbankstrukturen.
Wenn Datentypen verändert werden, muss sichergestellt werden, dass diese weiterhin persistiert werden können. Hierbei werden vermutlich weniger strenge Prüfungen vorgenommen als bei den Schnittstellen, da beispielsweise Formate für String-Typen irrelevant wären. Allerdings das Ändern eines Datentyps oder eines Datumsformates würde sich auch auf die Datenbanken auswirken.

7.1.4 Versionshistorie

Wie bereits im Umsetzungskapitel erwähnt, wäre eine mögliche Erweiterung das Umsetzen einer Versionshistorie. Wenn beispielsweise eine Microservice-Landschaft über mehrere Versionen hinweg verändert wird, wäre es sehr hilfreich auf alte Versionen zugreifen zu können.

Hieraus würde man als Vorteil ziehen, dass erkannt werden kann, welche Schnittstelle sich verändert hat und welche sich anpassen müssen. Außerdem kann ermittelt werden, wie alte Versionen der Schnittstelle aussahen. Dankbar wäre auch Kompatibilität zu mehreren verschiedenen Schnittstellenversionen zu testen, da man Kompatibilität der Klienten für einen möglichst langen Zeitraum schaffen will.
Denkbar hierfür wäre das Einsetzen einer auf Graphen basierenden Datenbank, welche die verschiedenen API-Versionen kennt. Bei Änderungen könnten die Änderungen, welche zu den Problemen führen, erkannt werden und so Hinweise auf die Lösung gegeben werden.

7.1.5 Zyklische Abhängigkeiten

Es wäre denkbar, dass es in einem Microservicesystem zu zyklischen Abhängigkeiten zwischen Services kommt. Dies wäre der Fall, wenn Services sich gegenseitig aufrufen und so ein möglicherweise endloser Zyklus entsteht, welcher im schlimmsten Fall die Services dauerhaft blockiert.
Da dies als sehr unwahrscheinlich erscheint, wurde diese Anforderung nicht aufgenommen. Allerdings ist es dennoch denkbar, dass ein derartiges System entstehen könnte. Dies wäre definitiv ein Problem, welches die Software erkennen sollte.

Literaturverzeichnis

[Analysis 2010] ANALYSIS, Structural: *Feature summary*. 2010. – URL https://maven.apache.org/maven-features.html

[Berjon und Song 2012] BERJON, Robin ; SONG, Jungkee: Web API Design Cookbook. In: *Crafting Interfaces that Developers Love*, apigee, 2012

[Foundation] FOUNDATION, The Apache S.: *Maven, Introduction*. – URL https://maven.apache.org/what-is-maven.html

[Killalea 2016] KILLALEA, Tom: The hidden dividends of microservices. In: *Communications of the ACM* 59 (2016), Nr. 8, S. 42–45. – URL http://dl.acm.org/citation.cfm?doid=2975594.2948985. – ISSN 00010782

[Lafon u. a. 2007] LAFON, Yves ; NIELSEN, Henrik F. ; GUDGIN, Martin ; KARMARKAR, Anish ; MENDELSOHN, Noah ; HADLEY, Marc ; MOREAU, Jean-Jacques: {SOAP} Version {1.2} Part 1: Messaging Framework (Second Edition) / W3C. apr 2007. – {W3C} Recommendation

[Newman 2015] NEWMAN, Sam: *Building Microservices: designing fine-grained systems*. 2015. – ISBN 978-1-491-95035-7

[Prasanna 2009] PRASANNA, Dhanji: *Dependency Injection*. 2009. – URL http://books.google.com/books?id=b6O6OgAACAAJ{&}printsec=frontcover

[RAML Workgroup 2017] RAML WORKGROUP: *RAML Version 1.0: RESTful API Modeling Language*. 2017. – URL https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/

[Smart 2011] SMART, John F.: *Jenkins: The Definitive Guide*. O'Reilly Media, Inc., 2011. – ISBN 1449305350, 9781449305352

[SmartBear Software 2018] SMARTBEAR SOFTWARE: *Swagger Documentation*. 2018. – URL https://swagger.io/docs/specification/about/

[Spichale 2016] SPICHALE, Kai: *API-Design.* dpunkt.verlag, 2016. – ISBN 978-3-86490-387-8

[Starke 2014] STARKE, Gernot: *Effektive Softwarearchitekturen.* 8. Hanser, 2014. – 409 S. – ISBN 978-3446436145

[Taibi u. a. 2017] TAIBI, D ; LENARDUZZI, V ; PAHL, C ; JANES, A: Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. In: *Proceedings of the XP2017 Scientific Workshops.* New York, NY, USA : ACM, 2017 (XP '17), S. 23. – URL http://dl.acm.org/citation.cfm?id=3120483. – ISBN 9781450352642

[Takai 2017] TAKAI, Daniel: *Architektur fuer Websysteme : serviceorientierte Architektur, Microservices, domaenengetriebener Entwurf.* Carl Hanser Verlag GmbH & Co. KG, 2017. – 400 S. – ISBN 9783446450561

[Thurlow 2009] THURLOW, R.: *RPC: Remote Procedure Call Protocol Specification Version 2.* RFC 5531. 2009. – URL https://www.rfc-editor.org/info/rfc5531

[Vinoski 2006] VINOSKI, Steve: Advanced message queuing protocol. In: *IEEE Internet Computing* 10 (2006), nov, Nr. 6, S. 87–89. – URL http://dx.doi.org/10.1109/MIC.2006.116. – ISBN 978-1-4673-5990-0

[Walls 2014] WALLS, Craig: *Spring in Action.* 4th Editio. Greenwich, CT, USA : Manning Publications Co., 2014. – 626 S. – ISBN 9781617291203

[Wolff 2015] WOLFF, Eberhard: *Microservices: Grundlagen flexibler Softwarearchitekturen.* 2. dpunkt.verlag, 2015. – ISBN 978-3-86490-313-7

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 18. Oktober 2018 Jan Dennis Bartels