Unternehmens die Granularität der Kontexte zu setzen.

Der Fokus auf ein spezifische Problemstellung, ermöglicht jedoch eine einheitliche Sprache innerhalb eines Kontexts einzufügen, sowie eine enge Verbindung zwischen Geschäftslogik und technische Entwicklung zu erzeugen. Somit vertritt Domain Driven Design den Ansatz, dass technische Modelle und Prozessbeschreibungen die gleiche Terminologie besitzen. Dadurch soll die Kommunikation zwischen Fachexperten und Softwareentwickler gestärkt werden.

Konkret für die Einteilung von Microservices bedeutet dies, dass ein Microservices ein Kontext abbildet, welche an ein Geschäftsprozess orientiert ist. Diese Eigenschaft deckt sich mit der von Sam Newman beschriebenden Definition, dass Microservices *„um eine Geschäftsdomäne herum modelliert sind“* (Abschnitt 2.2). Des Weiteren fordert Domain Driven Design, dass ein Team, welches für ein Service verantwortlich ist, auch Personen aus der Geschäftsabteilung hat und dass das Modell, durch ein fachlich diverses Team entsteht. In der strengsten Umsetzung von Domain Driven Design, besteht folglich ein Team aus Fachexperten, sowie Softwareentwicklern und besitzt einen engen Kontakt zu Usern. Nun handelt es sich bei Domain Driven Design jedoch um eine Ansatz, welcher nicht immer absolut umgesetzt wird.

Die Umsetzung des Domain Driven Design hat wiederum einige Vorteile. So wird durch das abgrenzen in einzelne Kontexte, die Autonomität des Teams weiter bestärkt und der Austausch zwischen Fachexperten und Softwareentwicklern durch die einheitliche Sprache verstärkt. Dadurch entstehen eigenständige Services, die hinsichtlich ihrer verwendeten Daten, als auch der verwendeten Terminologie, entkoppelt sind. Eingeteilt wird anhand von Geschäftsprozessen.

2.3 Microservice

Wie aus dem vorhergehenden Abschnitt hervor geht sind Microservices einzelne, unabhängigen Services, die um eine Geschäftsdomäne modelliert sind. Die auf Grundlage der klaren Schnittstellen entkoppelt und durch einzelne Kontext begrenzt sind. Die Kontexte ermöglichen das Verwenden von Service internen Terminologie und Verwendung ausschließlich benötigter Daten. Abschließend hat Abschnitt 2.2.2 gezeigt, dass nur ein Team für ein Service zuständig ist.

In seinem Buch Monolith to Microservices geht Newman darauf ein, dass wohl die wichtigste Eigenschaft von einem Microservice die unabhängige Einsetzbarkeit ist. Er sieht darin die Verkörperung von Unabhängigkeit und eigenständigen Teams, welche durch die Kommunikation der Services durch standardisierte Schnittstellen ermöglicht wird. Die meist verbreitete Art der Schnittstelle ist die nach dem REST-Standard, welche sich an

eine Kommunikation über HTTP richtet (siehe Abschnitt 2.1.4).

Da viele Programmiersprachen den Informationsaustausch über HTTP ermöglichen, ist ein Microservice unabhängig einer bestimmten Technologie. Somit kann jedes Team eigenständig entscheiden, welche Programmiersprache sie verwenden wollen. Dadurch können Sprachen nach Präferenz und Problemstellen gewählt werden, ohne dass eine Inkompatibilität zum Rest des Systems entsteht.

Vielmehr ermöglicht der Austausch über standardisierten Kommunikationswege, dass Services erstellt werden, die austauschbar sind. Somit können zum Beispiel einige Services durch kommerzielle Diente ausgeführt werden und Entwicklungskosten gespart werden. Außerdem wird dadurch die Möglichkeit geschaffen auf bestehender Logik aufzubauen und ältere Systeme mit modernen Technologien zu verbinden.

Abschnitt 2.2.3 hat gezeigt das Kontextgrenzen Microservices es ermöglicht Daten nach eigenem ermessen zu benennen und zu verwalten. Konkret besitzt jeder Dienst eine eigene Datenspeicherung, in Form zum Beispiel eines File-Storage-Systems oder einer Datenbank. Für das Gesamtsystem bedeutet dies, dass keine einheitliche Datenspeicherung vorliegt, sondern jeder Service die notwendigen Informationen verwaltet. Um jedoch die Funktionalität des gesamten Systems zu erhalten, müssen im Vorfeld bestimmt werden, wie der Informationsaustausch zwischen den Services abläuft. Hierfür ist es wichtig zu bestimmen, welche Services welche Informationen verwalten und falls ein Abhängigkeiten zu anderen Diensten gibt, die Schnittstellen festzulegen. Insbesondere die genauen Endpunkt und die zu erwarteten Informationen müssen bestimmt werden. Die Handhabung innerhalb eines Services, ist dann wieder dem einzelnem Team überlassen.

Gibt es ein Service, der Informationen von einem anderen Dienst benötigt, fordert dieser die Daten über öffentliche Schnittstellen aktive an. Dies führt jedoch dazu, dass gleiche Informationen von mehrere Diensten verwaltet werden und es zu unterschieden Datenständen kommt. Es entsteht die Herausforderung Informationen zwischen Services Konsistent zu halten (siehe Abschnitt 2.5.1).

Volle Unabhängigkeit eines Microservice kann nur erreicht werden, wenn dieser auch für die Benutzeroberfläche verantwortlich ist. In der Praxis ist dies jedoch schwierig, da eine Webseite Inhalte von mehreren Services anzeigt. Insbesondere nachdem 2010 Angular.js veröffentlicht wurde, gibt es immer mehr JavaScript Frameworks die um die Benutzererfahrung zu verbessern mehr Logik in das Frontend legen. React.js ist einer dieser Frameworks. Die Beliebtheit hinter diesen Frameworks, liegt darin, dass jede einzelne Webseite nicht mehr als einzelne Seite betrachtet wird, sondern Informationen zwischen Benutzeroberflächen hinweg verwaltet werden. Dadurch kann ein Datenabruf über mehrere Webseiten geteilt werden und somit die Performance verbessern. Erreicht wird dies, indem das Routing zwischen den Webseiten durch das Framework verwaltet wird. Dieser

Ansatz wird als Single-Page-Application (SPA) bezeichnet und unterscheidet sich von der Idee des Model-View-Controller aus Abschnitt 2.1.3, da die nötigen Informationen aktive von der View geladen werden.

Da in der Praxis viele Webseiten Single-Page-Applicationen einsetzen, um eine gute Benutzererfahrung zu liefern, sollte in diesen Fällen über mögliche Aufteilung nachgedacht werden. Insbesondere wenn es sich um Applikation handelt, die von mehrere Teams verwaltet werden. Ist dies nämlich der Fall, kann darunter die Unabhängigkeit der Teams leiden.

Auch kann es vorkommen, dass ein Microservices keine graphische Darstellung benötigt, da dieser zum Beispiel nur E-Mails verschickt oder die beliebtesten Filme ermittelt. Somit gibt es keine absolute Aussage darüber, ob ein Microservice eine Benutzeroberfläche haben sollte.

Insbesondere punkte Skalierung bieten Microservices einige Vorteile. So lassen sich individuell, intensiv genutzte Service unabhängig vom Gesamtsystem skalieren. Dies kann ein großer Kostenvorteil darstellen, da nur ein einzelner Teil und nicht das gesamte System verstärkt werden muss. Auch können auf Basis der eingeteilten Teams weitere Entwickler angestellt werden und das Unternehmen hinsichtlich der Angestellten wachsen.

Während Microservices Skalierungen begünstigen, erschweren sie serviceübergreifende Veränderungen. Somit ist es auf Grund der technologischen Freiheit aufwendiger Entwickler von einem Service zu einem Anderen zu überführen. Dies wird dadurch verstärkt, dass unterschiedliche Daten gespeichert und verschiedene Terminologien verwendet werden. Auch eine technologische Überführung von Funktionen wird erschwert, wenn verschiedener Programmiersprachen eingesetzt werden. Äquivalent ist das zusammenführen von Datenschemata, welches durch die unterschiedlichen Begriffe boykottiert wird. Dies führt dazu, dass ein Zusammenschluss nur durch einen großen Aufwand erreicht wird kann.

2.4 Produktmarketfit bestimmen

In diesem Abschnitt soll geklärt werden, wie ein Produktmarketfit bestimmt werden kann. Hierfür möchte ich viel aus Running Lean ziehen.

Es geht darum zum Einen Produktmarketfit zu definieren und zum anderen eine Methode zu gegeben an dieser ein solcher Zustand gemessen werden kann.

2.5 Kommunikation von Services

In diesem Abschnitt möchte ich darauf eingehen welche anforderungen eine entkoppelte Infrastuktur an das Netztwerk und an die Kommunikation über standards hat.