Planung und Erstellung einer Backend-Microservices-Architektur aus den Anforderungen anhand des Spiels Stirnraten.

Michael Rothkegel

14. Juli 2019

Inhaltsverzeichnis

1 Glossar

Das Wörterverzeichnis dient als Nachschlagewerk für Begriffe, die bereits bekannt sein können oder im Laufe dieser Arbeit eingeführt werden. Zusätzlich gilt zu erwähnen, dass folgende Wörter als Synonym zueinander verwendet werden: Microservice/Service, Anfrage/Request/API Call, Dockercontainer/Container, Gatway/API Gateway.

Begriff	Erklärung
Deployen/Deployment	Ein Softwareprodukt updaten.
Hosten	Ein Softwareprodukt auf einem Server bereitstellen.
CI/CD	Continuous Integration/Continuous Delivery beschreibt den kontinuierlichen Prozess ein Softwareprodukt von der Entwicklung bis zur tatsächlichen Veröffentlichung zu begleiten.
User Interface (UI)	Oberfläche für Benutzer.
Virtuelle Maschine (VM)	Kapselung eines Rechnersystems innerhalb eines anderen.
Legacy (Erbe)	Es handelt sich um Altsysteme, die eine Erneuerung benötigen oder abgelöst werden müssen.
Technologiestack	Technologien, die für ein Produkt oder in einer Firma verwendet werden.
Downtime (Stillzustand)	Zeitpunkt, den eine Anwendung benötigt, bis sie neu gestartet ist. Während der Downtime ist eine Anwendung nicht erreichbar.
Overhead/Technische Schuld	Technische Entscheidungen, die zu Mehrarbeit führen.
Monolith	Eine Anwendung mit monolithischer Struktur.
Pattern (Muster)	Entwurfsmuster zum Bewältigen von Aufgaben.
Domain Driven Design (DDD)	Pattern zum Abtrennen von Microservices.
Bounded Contexts	Kontexte, die aufgrund ihrer Aufgaben zusammenhängen.
Up-stream	Stellt dem down-stream Informationen bereit.
Down-stream	Erhält vom up-stream Informationen.
Client	Ein Gerät, welches Anfragen an die API stellt.
API	Die Schnittstelle einer Software, über die kommuniziert werden kann.
REST	Ein Programmierparadigma für Webservices.
Monitoring	Das Überwachen von Diensten oder Microservices.
Logging	Das Aufzeichnen von Daten zum Nachvollziehen vergangener Aktionen.
Traffic	Datenverkehr, der zwischen der anfragenden und verarbeitenden Instanz entsteht.
Timeout	Entsteht, wenn ein Dienst nicht erreichbar ist.

Fallback	Eine Art Notfallverhalten, wenn das beabsichtige Verhalten nicht eintrifft.
API Route/Route	Pfad über den HTTP-Requests abgesetzt werden können.
Message Broker/Eventbus	Vermittler, um Nachrichten zu versenden.
Subscribe (abonnieren)	Nachrichten, welche beim Eventbus abonniert werden.
Publish	Nachrichten, welche veröffentlicht werden.
Load Balancer	Verteilt die Last/den Traffic auf verschiedene Services, um das Gesamtsystem zu entlasten.
Backend	Anwendung, die für die Logik zuständig ist. Sie hat keine Oberfläche.
Performance	Leistung des Systems. Umso schneller die Performance ist, desto besser.
Latenz	Verzögerungszeit, z.B. die Länge des Zeitintervalls vom Beginn bis zur tatsächlichen Reaktion.
Middleware	Ein Knotenpunkt, den jeder Request durchlaufen muss.
Parsen	Werte aus einem meist speziellen Format entnehmen und weiterverarbeiten.
Enkodieren und Dekodieren	Daten in ein spezielles Format umwandeln oder zurückwandeln.

Tabelle 1: Glossar

2 Einleitung

Wenn ein Onlineshop genutzt wird, wirkt dieser wie eine einzelne Softwareanwendung. Es wird sich durch Artikel geklickt, der Warenkorb befüllt und schließlich eine Bestellung abgesendet. Wächst ein Onlineshop, wird dieser entsprechend komplexer und bietet zahlreiche Funktionalitäten. Auf Softwareebene entstehen dadurch Herausforderungen. Wie wird z.B. mit massiv steigenden Benutzerzahlen oder vermehrt hinterlegten Datensätzen umgegangen? Dieses Phänomen betrifft natürlich nicht nur Onlineshops, sondern alle Softwareanwendungen, die viel genutzt werden, wie z.B. Spiele- oder Streamingplattformen, soziale Netzwerke, Buchungsseiten für Hotels, Autos oder Flüge usw. Zusätzlich entstehen innerhalb des Unternehmens personelle Herausforderungen. An großer Software arbeiten in der Regel viele Mitarbeiter, diese müssen sich entsprechend organisieren und absprechen, damit sie nicht destruktiv arbeiten.

Abhilfe schafft die sogenannte Microservices-Architektur. Die Idee ist, eine große Softwareanwendung (Monolith) in viele kleine, eigenständig und in sich funktionierende Anwendungen (Microservices) zu zerteilen. Viele große Unternehmen wie Netflix, Amazon und Google bauen ihre Dienste bereits so auf. Aber auch deutsche Unternehmen wie Zalando, Otto oder Rewe nutzen diese Art der Architektur.

Für den außenstehenden Benutzer ist die Architektur irrelevant, d.h. die Anwendung hat dasselbe Erscheinungsbild und dieselbe Funktionalität. Auf technischer Ebene bietet die Microservice Architektur dagegen einfache Möglichkeiten der Skalierung, schnelle Updatezyklen und Agilität. Auch auf personeller Ebene kann die Organisation leichter werden, denn kleinere Teams sind verantwortlich für Microservices und können diesen entsprechend weiterentwickeln ohne andere Teams zu blockieren oder womöglich Fehler einzubauen.

In der folgendenden Arbeit wird exemplarisch eine API nach Microservice Gesichtspunkten eruiert. Dafür wird zunächst untersucht, inwiefern sich der Monolith zu den Microservices unterscheidet und wie man Microservices untereinander abgrenzen kann. Ebenfalls wird die Schnittstellenkommunikation zwischen Microservices beleuchtet sowie die Herausforderung einer Authentifizeriung und Autorisierung. Zusätzlich wird die Infrastruktur zum Aufsetzen mittels Docker und die Erreichbarkeit aller Services durch ein API Gateway behandelt.

Anschließend werden Anforderungen, die an die Stirnraten API gestellt werden, mittels User Story Mapping erfasst und herausgearbeitet. Für die Umsetzung werden in der Konzeption verschiedene Technologien und Strategien gegeneinander abgewogen und darauf beruhend in der Implementierung verwendet.

Ziel ist es, eine lauffähige Stirnraten API basierend auf der Microservice-Architektur umzusetzen und der Stirnraten-App die Möglichkeit zur Kommunikation mit einem Backend zu geben.

3 Grundlagen

Der folgende Abschnitt arbeitet eine Abgrenzung zwischen einer Microservice- und monolithischen Architektur heraus. Zusätzlich geht er auf unterschiedliche Kommunikationsarten bei Microservices sowie die Authentifizierung und Autorisierung ein.

3.1 Definition Microservices

Für den Begriff Microservices existiert keine einheitlich anerkannte Definition. Während Wolff unter Microservices unabhängige, deploybare Module versteht[30], spricht Newman von kleinen, autonomen Services, die zusammenarbeiten. Cockcroft verwendet den Begriff Microservice gekoppelt mit einem Architekturbegriff: Eine Microservice Architektur sind gekoppelte Services, welche für einen gewissen Kontextbereich zuständig sind.[9] D.h. jeder Service behandelt gewisse, fachliche Aufgaben und kann genau für diese genutzt werden. Eine Vielzahl von solchen Services bildet dann die gesamte Anwendung.

Amudsen schreibt dem Microservice an sich die Eigenschaft zu, dass er unabhängig zu anderen Microservices sein muss, d.h. ein Microservice kann losgelöst von anderen geupdated (deployed) werden. Weiter ist ein Microservice wie schon bei Cockcroft für einen gewissen Aufgabenbereich zuständig. Eine Microservice-Architektur ist ein zusammenschluss von miteinander kommunizierenden Microservices.[9]

In *Flexible Software Architecture*[31] werden Microservices zu den bisherigen Eigenschaften noch weitere, teils technische zugeschrieben: Microservices sind technologisch unabhängig, d.h. eine Microservice-Architektur ist beispielsweise nicht an eine bestimmte Programmiersprache oder Datenbank gebunden. Weiter müssen Microservices einen privaten Datenspeicher haben und sie kommunizieren mit anderen Services über das Netzwerk (z.B. über REST). Ebenfalls werden Microservices verwendet, um große Programme in kleine Teile zu unterteilen. Diese kleinen Teile lassen sich automatisch bauen und deployen.

Basierend auf den folgenden Definitionen wird der Microservice Begriff wie folgt verwendet: Microservices sind

- klein in der Größe,
- kommunizieren mit anderen Services über Netzwerkschnittstellen (z.B. REST),
- sind unabhängig voneinander deploybar,
- können unabhängig voneinander entwickelt werden (d.h. Microservice A muss nicht auf B,C,D ... warten und/oder umgekehrt),
- eingeschränkt in ihrer Geschäftslogik, d.h. ein Microservice kümmert sich immer um einen speziellen Kontext, der im Vorhinein definiert werden muss,

- dezentral, d.h. sie können auf unterschiedlichsten Plattformen gehosted werden.

Abschließend handelt es sich um eine Microservice-Architektur, wenn viele Microservices nach Definition verwendet werden.

3.2 Monolithische Struktur

Eine monolithische Struktur ist ein einziges Softwareprogramm (Monolith), welches in sich geschlossen ist. Dies bedeutet im Detail, dass ein Monolith aus mehreren Ebenen besteht, auf die über Schnittstellen zugegriffen werden kann. Innerhalb der Ebenen werden Komponenten wie z.B. Frameworks oder selbstgeschriebene Klassen eingebunden und verwendet.[1] Durch die sich aufeinander aufbauenden Ebenen folgt daraus, dass sämtliche Geschäftslogiken, User Interfaces sowie die Datenbank und Datenbankzugriffe Abhängigkeiten haben. All dies ist in einem Programm vereint.[1] Natürlich kann die monolithische Struktur innerhalb noch einmal Modular sein, d.h. in einem Monolithen existierten ggf. mehrere Module, welche verschiedene Geschäftslogiken abbilden. Dennoch können diese Module nicht unabhängig von der gesamten Anwendung deployed werden.[14]

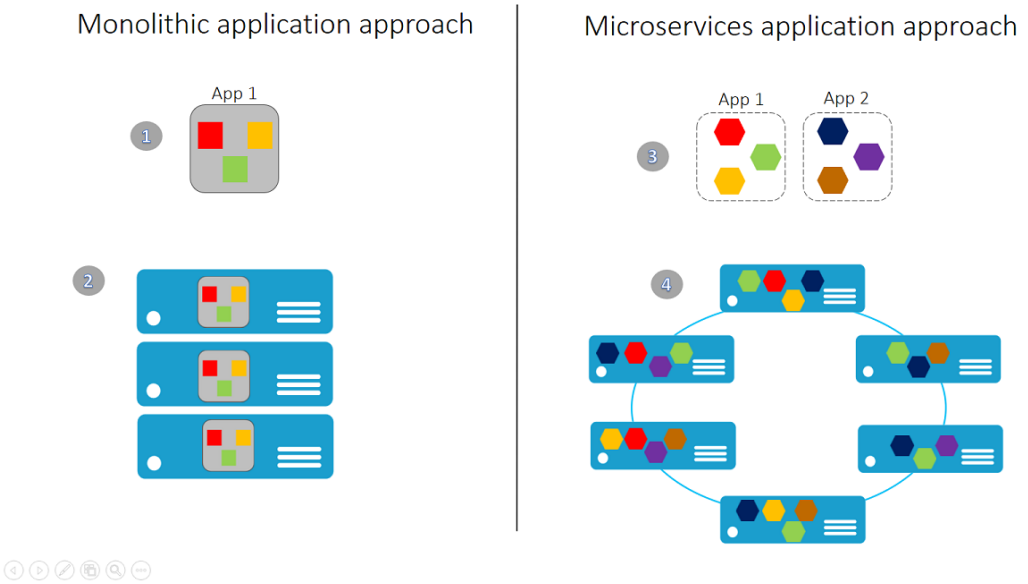

Abbildung 1: Monolith und Microservice-Architektur.[1]

Abbildung 1 zeigt eine vereinfachte Gegenüberstellung der beiden Architekturen. Die App 1 ist in drei klassische Funktionen (Web, Business und Data) unterteilt. Die Skalierung (2) kann durchgeführt werden, indem App 1 über mehrere Server oder VMs geklont wird.

Bei der Microservice-Architektur werden die Funktionen auf unterschiedliche Dienste aufgeteilt. Konkreter könnte dies bedeuten, dass App 1 bei (3) zuständig für eine Be-

nutzerkontoverwaltung ist und App 2 für ein Abrechnungssystem. Die Microservices (4) werden nicht geklont, sondern können unabhängig voneinander bereitgestellt werden.

3.3 Monolith vs. Microservices

Aus den vorherigen Abschnitten sind diverse Unterschiede zwischen den Architekturen erkennbar geworden. Nun gilt es festzustellen, welche Architekturen für welche Problemstellungen, sinnvoller ist. [31] [3]

Aus der Tabelle 2 ergeben sich verschiedene Punkte: Der Monolith eignet sich besonders dann sehr gut, wenn die Projekt- sowie Teamgrößen absehbar und auch die benötigten Technologien klar sind. Zusätzlich kann ein Monolith beim Projektanfang von Vorteil sein, da die Abhängigkeiten innerhalb des Projektes liegen und so die Entwicklungsgeschwindigkeit nicht durch eine kompliziertere Infrastruktur blockiert wird.

Ist die Projektgröße allerdings nicht absehbar, treten früher oder später mehrere Schwierigkeiten auf: Zum einen bindet der am Anfang des Projektes festgelegte Technologiestack, d.h. die Nutzung oder der Austausch neuer Technologien ist in der Regel mit sehr viel Arbeit verbunden. Zum anderen führen die anfangs eingegangen Abhängigkeiten zu Problemen im Deployment (A kann erst updaten, wenn B soweit ist) und einem erhöhten Aufwand in der Kommunikation zwischen den Teams (A kann erst beginnen, wenn B xy erledigt hat).

Die Skalierung von Microservices dagegen ist unabhängiger. Es können sich feingranular Services gesucht werden, welche skaliert werden sollen. Diese benötigen nicht zwingend mehr Hardware (vertikale Skalierung), sondern könnten z.B. auch auf verschiedene Server verteilt werden (horizontale Skalierung). Dies ist bei einem Monolithen natürlich auch möglich, dennoch muss immer der ganze Monolith skaliert werden, welcher zum einen immer mehr Hardware als einzelne Microservices benötigt und zum anderen in der Regel auch durch die Komplexität schwerer zu skalieren ist.[30]

Als abschließender Punkt ist die Robustheit zu erwähnen: Wenn ein Microservice einen Fehler enthält, stürzt dieser im schlechtesten Fall ab. Im besten Fall übernimmt dieser Service eine weniger wichtige Funktion und der Nutzer bemerkt den Ausfall noch nicht einmal. Beim Monolithen dagegen stürzt die gesamte Anwendungen ab. In der Regel startet so eine Anwendung automatisch neu, jedoch betrifft die Downtime alle Nutzer.

Eine generelle Aussage, welche Architektur besser oder schlecht ist, lässt sich dementsprechend nicht treffen. Es kommt immer darauf an, welche Zielsetzung und wie viele Ressourcen für das Projekt festgelegt sind. Eine monolithische Architektur ist schneller umsetzbar und es ist nicht gegeben, dass ein Softwareprodukt überhaupt so viel Nachfrage erzeugt und somit eine Microservice-Architektur notwendig ist. Dementsprechend muss abgewogen werden, welche Architektur für welchen Anwendungsfall besser geeignet ist.[30] REWE Digital beispielsweise hat ihr Produkt zuerst als Monolithen gestartet und ist erst später auf eine Microservice-Architektur umgeschwenkt.[7]

	Monolithische Architektur	Microservice-Architektur
Abhängigkeiten	all1es in einer Anwendung	entkoppelt, da Prinzip von Modularisierung verwendet wird
Größe	linear steigend	einzelne Services sind klein
Geschwindigkeit der Zugriffe	schnell, da alles in einer Anwendung	Zugriffe können länger dauern
Deployment	schwieriger je größer das Projekt, aufgrund von • Abhängigkeiten • Größe	einfach, da Microservices • klein und • modular sind
Organisation	leichter, da alles an einem Ort	schwerer, da mehr Domänenlogik (wer macht was?) beachtet werden muss
Legacy-Systeme ablösen	ggf. schwierig, da System sehr verzahnt mit anderen Technologien ist	leicht, da Microservices durch neue abgelöst werden können
Technologie	beschränkt	vielfältig
Nachhaltige Entwicklung	wartbar mit Einschränkungen	leicht wartbar
Robustheit	weniger, da ganzes System bei schweren Fehlern abstürzt	sehr, da im Zweifel immer nur ein Service abstürzt
Skalierung	horizontale und vertikale Skalierung, Umsetzung kann sehr komplex werden	horizontale und vertikale Skalierung
Betrieb	nur ein System	komplex, da mehr Services verwaltet werden müssen

Tabelle 2: Monolitih vs. Mircoservice-Architektur

3.4 Architektur von Micrsoservices

Wie bereits erwähnt, ist die Entkopplung von Microservices ein großer Vorteil gegenüber dem Monolithen. Dennoch ist es sinnvoll, Richtlinien, Regeln und/oder Festlegungen zu schaffen, damit die Microservices nicht blockierend oder technologisch gegensätzlich arbeiten. Die Entscheidungsebene kann global (Makroarchitektur) oder nur für einen einzelnen Service (Mikroarchitektur) gelten.[31] Welche Festlegungen und mit welcher Strenge diese eingehalten werden müssen, hängt von verschiedenen Faktoren ab, welche technologisch, organisatorisch oder wirtschaftlich motiviert sein können.[7]

In dem folgenden Abschnitt wird das Grundprinzip der Software-Modellierungs-Methodik Domain Driven Design untersucht. Zusätzlich wird erläutert, welche Fälle makro- oder microarchitektonisch einzuordnen sind.

3.4.1 Domain Driven Design

Domain Driven Design (DDD) ist ein Vorgehen, mit dem ein Softwaresystem modelliert werden kann. Im Sinne einer Microservice-Architektur kann dies als Werkzeug genutzt werden, um Microservices fachlich einzuteilen.[20] Beim sogenannten *Strategic Design* wird dafür das Softwaressytem in verschiedene *Bounded Contexts* eingeteilt, welche an ein *Domänenmodell* gebunden sind. Ein Domänenmodell bildet die Geschäftslogik ab, d.h. inwiefern einzelne Objekte innerhalb des Kontextes in Relation zueinander stehen, welche Eigenschaften sie haben und wie sie sich verhalten. Dabei kann ein Domänenmodell - je nach Entwurfsmuster - von einem oder mehreren Bounded Contexts genutzt werden.[30]

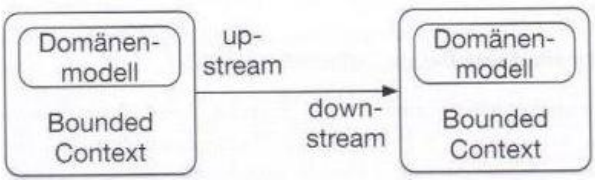

Abbildung 2: Bounded Contexts mit eigenständigem Domänenmodell.[30]

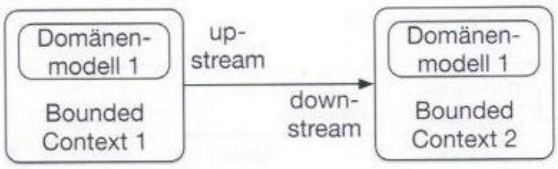

Abbildung 3: Bounded Context 2 adaptiert das Domänenmodel von Bounded Context 1.[30]

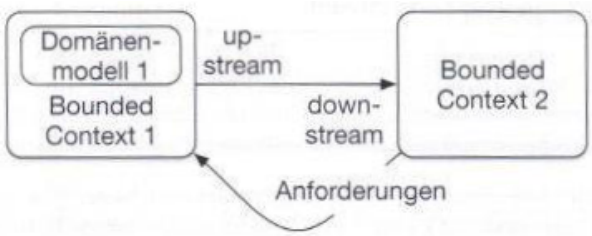

Abbildung 4: Bounded Context 2 erhält ein auf ihn zugeschnittenes Domänenmodel von Bounded Context 1.[30]

Wolff verdeutlicht das Prinzip von Bounded Contexts mit Hilfe von vier Microservices, welche einen Onlineshop repräsentieren: *Suche*, *Check Out*, *Inkasso* und *Lieferung*. Während das Datenmodell der Suche detaillierte Informationen über die Produkte enthält, reicht es im Warenkorb (hier Check Out), wenn ggf. nur der Produktname gespeichert wird. Bei Inkasso ist es ähnlich: An dieser Stelle sind Zahlungsdaten des Benutzers relevant, während bei der Lieferung ggf. nur die Adresse notwendig ist.[30] An diesem Beispiel wird deutlich, dass zwar von denselben Begrifflichkeiten wie Benutzer, Produkt usw. gesprochen wird, allerdings jeder Service sein eigenes Domänenmodell hat. Diese Technik ist unterstützend dabei, eine saubere Microservice-Architecktur abzubilden.[30] [20]

Neben der fachlichen Trennung bildet DDD auch die Kommunikation zwischen Kontexten ab. Dabei wird grundsätzlich vom *up-stream* (vorgeschaltet) und dem *down-stream* (nachgeschaltet) gesprochen.[30] Der up-stream stellt dem down-stream Informationen bereit. Wie dies technisch umgesetzt ist, also ob der down-stream nachfragt oder der up-stream aktiv Daten schickt, ist frei wählbar.

Nach dem Anwenden von DDD sollte die Struktur der Software erkennbar sein: D.h. welche Art Microservices werden benötigt und inwiefern steht sie mit anderen in Abhängigkeiten oder in einer Kommunikation.

3.4.2 Macro- und Mikroarchitektur

Wenn durch das DDD entworfen wird, welche Microservices voraussichtlich benötigt werden, ist es sinnvoll, einen Art Bauplan zu verfassen, welcher impliziert, an welche Regeln sich ein Microservice halten muss. Diese Regeln können wie bereits erwähnt auf globaler Ebene getroffen werden, d.h. sie gelten für alle Services (Makroarchitektur) oder sie gelten nur im Microservice selber (Microarchitektur). REWE Digital unterscheidet dabei zwischen *Must*, *Should*, *Could*. D.h. es gibt Regeln, die Microservices erfüllen müssen, wie z.B. das Kommunzieren über REST oder das Implementieren einer einheitlichen Autorisierung.[7] Andere Regeln dagegen sind viel mehr Richtlinien (should) oder komplett optional (could). Das Ziel ist stets, dass durch die makroarchitektonischen Entscheidungen nicht die Vorteile von Microservices beschnitten werden.[30][9].

Es gibt verschiedene Einflussfaktoren wie sich die Makroarchitektur für ein Unternehmen definiert: Zum einen empfiehlt es sich, ein Gremium zu gründen, welches sich stetig mit den Regeln der Makroarchitektur auseinandersetzt, sie entsprechend erweitert, überarbeitet und die getroffenen Entscheidungen auch immer begründen kann.[30] Zum anderen besteht immer ein technischer Einfluss[30]:

- Gewählte Technologien müssen in die Infrastruktur des Unternehmens passen: Angenommen die Auslastung eines Microservices muss überwacht werden und dies wird firmenweit mit Tool A erledigt, dann wäre es sehr aufwendig, wenn der besagte Service nur eine Schnittstelle für Tool B anbietet und der nächste Service nur für Tool C. Dies würde einerseits sehr unübersichtlich werden und andererseits viel Aufwand bedeuten.

- Technologien sind immer von dem Personal abhängig: Gerade wenn Unternehmen klein bis mittelständig sind, empfiehlt es sich, Technologien zu nutzen, die mehrere Entwickler beherrschen, um Inselwissen zu reduzieren.

- Ebenfalls können gezielt strategische Entscheidungen getroffen werden, z.B. wenn ein Unternehmen die Dateninfrastruktur zu einem Cloudanbieter auslagern möchte, hat dies entsprechende makroarchitekonische Auswirkungen.

Basierend auf Wolff und Nadareishvili ist Tabelle 3 entstanden, welche einen Überblick darüber gibt, wie gängige Themen eingeordnet werden können. [30][9][7]

	Mikroarchitektur	Makroarchitektur
Programmiersprache	x	x
Datenbank	x	x
Look and Feel (UI)	x	x
Dokumentation	x	x
Datenformat		x
Kommunikationsprotokoll		x
Authentifizierung		x
Integrationtests		x
Autorisierung	x	
Unittests	x	
Continuous-Delivery-Pipeline	x	

Tabelle 3: Entscheidungen Micro- und Macroarchitektur

In der Tabelle 3 sieht man, dass gerade die ersten Punkten sehr von der Unternehmenskultur und den technischen sowie personellen Freiheiten abhängen. In der Theorie sollte die Programmiersprache sinnvoll für jeden Microservices gewählt werden, dennoch ergibt es auch Sinn, einen Pool an Programmiersprachen auf Makroebene zu definieren,

um Inselwissen zu reduzieren und nachhaltige Codequalität zu gewährleisten. Ähnliches gilt beispielsweise für die Wahl der Datenbank: Ist bereits eine globale Infrastruktur für Datenbank X geschaffen, sollte diese nicht ohne Weiteres aufgebrochen werden, nur weil es technisch möglich ist.

Bei der Dokumentation sowie beim Look & Feel ist es sinnvoll, globale Richtlinien zu definieren, damit klar ist, wo bei jedem Microservices die Dokumentation zu finden ist oder wie ein User Interface grundsätzlich angeordnet und gestaltet werden soll. Dennoch können diese Punkte im Detail je nach Microservice abweichen.

Ein Kommunikationsprotokoll (z.B. REST) sowie Datenformate (z.B. JSON) sollten festgeschrieben werden.[7] [30] Als Grund wird zum einen das Vermeiden von technischem Mehraufwand angegeben, zum anderen sind Microservices zwar eigenständig deploybare Einheiten, dennoch sollten sie technisch zum Gesamtsystem passen und nicht dagegen arbeiten.

Während die Authentifzierung (um wen handelt es sich) einmalig festgelegt werden sollte, liegt die Überprüfung der Autorisierung (was darf der Benutzer) in jedem Microservice selbst. Die Alternative wäre, dass jede eingehende Anfrage noch einmal geprüft wird, was zu unnötig hohem Traffic und somit zu Verzögerungen führen würde.

Die hier erarbeitete Tabelle 3 ist an dieser Stelle nicht als feststehendes Manifest für alle Unternehmen zu verstehen, sondern zeigt eine mögliche, sinnvolle Einordnung. Natürlich können architektonische Entscheidungen stark vom jeweiligen Anwendungszweck abhängen. Nicht aufgeführt ist beispielsweise der Umgang mit Konfigurationsdateien, Monitoring oder Logging. Beim Monitoring könnte zum Beispiel global entschieden werden, *wo* Metriken abgelegt werden bzw. mit *welcher* Technologie gearbeitet wird. Aus microarchitektonischer Sicht könnten die Services allerdings selbst entscheiden, *was* gemessen wird.

Ebenso beim Deployment: Es gibt zahlreiche Methoden, um neue Updates bereitzustellen wie z.B. mittels Docker, Kubernetes oder indivudelle Installationsskripte.[30] Welche Technologie sich durchsetzt, muss anhand der Anforderung entschieden werden.

Aus den erarbeitenten Punkten lassen sich Vor- und Nachteile ableiten. Vorteile für microarchitektonische Entscheidungen sind ein sehr hohes Maß an Flexibilität und eine hohe Unabhängigkeit im Gesamtsystem, was grundsätzlich das Ziel von Microservices ist. Dies wiederum kann dazu führen, dass Entwicklungsoverhead oder Inselwissen entsteht. Ebenfalls könnten Punkte wie z.B. das *Look & Feel* oder die *Dokumentation* darunter leiden.

Macroarchitektonische Entscheidungen haben zum Vorteil, dass es Regeln gibt, welche die Entwicklung vereinfachen sollen und gegebenfalls die Nachteile der Microarchitektur kompensieren können. Auf der anderen Seite schränken macroarchitekonische Entschei-

dungen ein. Zusätzlich müssen sie organisch, z.B durch ein extra dafür geschaffenes Gremium, durchgesetzt werden.[30] Im Gesamten lässt sich daraus schließen, dass sehr genau abgewogen werden muss, welche Entscheidungen global oder individuell entschieden werden. Grundsätzlich gilt, dass jede Entscheidung begründbar sein muss.

3.5 Kommunikation

Bei einem Monolithen wird eine Abfrage über eine Route gestellt, woraufhin die Anwendung entsprechend mit der Bearbeitung beginnt. Da die gesamte Datenhaltung an einer Stelle ist, sind alle Daten bekannt und abrufbar. Wichtiger noch: Die Daten sind konsistent.

Microservices sind diesbezüglich herausfordernder. Es müssen verschiedene architektonische Entscheidungen getroffen werden, wie z.B. ob es einen zentralen Service gibt, welcher alle Anfragen weiterleitet (**API Gateway**) oder ob jeder Service einzeln erreichbar ist. Ebenfalls sollte auch begründbar entschieden werden, ob eine synchrone, asynchrone oder möglicherweise eine hybride Kommunikation verwendet wird.

In den folgenden Unterkapiteln werden Vor- und Nachteile der verschiedenen Kommunikationsarten für Microservices mit einem Schwerpunkt auf Unabhängigkeit (Entkopplung) und Datenkonsistenz untersucht.

3.5.1 Synchrone Kommunikation

Wenn ein Microservice bei der Bearbeitung einer Anfrage selbst eine weitere Anfrage an einen anderen Microservice stellen muss und auf das Ergebnis wartet, spricht man von synchroner Kommunikation.[30]

Anhand dieser Definition lässt sich folgendes Szenario darstellen (siehe Abbildung 5).

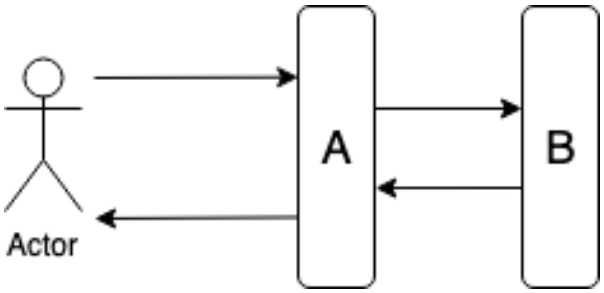

Abbildung 5: Synchrone Kommunikation

Der Actor stellt bei Microservice A eine Anfrage, welche bearbeitet und schließlich an B weiterleitet wird. B verarbeitet diese ebenfalls und antwortet, so dass auch A wieder antworten kann. Aus diesem Ablauf lässt sich festhalten, dass die übertragenden Daten aktuell sind. D.h. der Actor erhält definitiv konsistente Daten, was positiv zu vermerken ist. Problematischer dagegen ist die Abhängigkeit, welche entsteht. Sollte B nicht

erreichbar sein, läuft A in einen Timeout und ist blockiert. Einerseits ließe sich argumentieren, dass genau dies passieren soll, schließlich scheint es einen Fehler zu geben. Aber angenommen A wäre ein Service zum Erstellen von Rechnungen und B ein Service zum Sammeln von Daten für Marktanalysen. A möchte B infomieren, dass eine neue Rechnung erstellt wurde, läuft allerdings in einen Timeout. Die Operation, eine Rechnung zu erstellen, hat allerdings eine höhere Priorität als es statistisch zu erfassen. Nach den aufgestellen Definitionen aus Abschnitt 3.3 für Microservices wird die Entkopplung, Modularität und Robustheit des Systems verletzt, da A nicht weiterarbeiten kann.[30] [12]

Bläht man das Beispiel auf, so dass weitere Services statistische Daten erfassen wollen, würden zahlreiche Microservices ausfallen (siehe Abbildung 6).

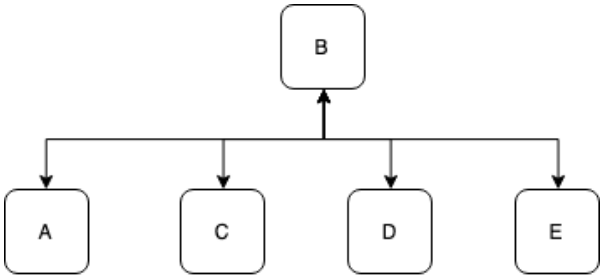

Abbildung 6: Abhängigkeiten in synchroner Kommunikation

Die Microservices können auch nicht auf eine Notfallstrategie (**Fallback**) zurückgreifen. Nehme man an, Service B ist nach x Sekunden nicht erreichbar, so wird die ursprüngliche Abfrage durch A weiter abgearbeitet, um nicht zu blockieren. Nun müsste allerdings eine andere Logik dafür sorgen, dass die Daten, die nicht übertragen werden konnten, zu einem anderen Zeitpunkt nachgetragen werden. In diesem Moment würde keine Datenkonsistenz mehr gewährleistet sein.

Auch nicht zu vernachlässigen, ist die Geschwindigkeit, mit der die Abfragen abgearbeitet werden können. Möglicherweise muss B in einem anderen Szenario noch mit C kommunizieren. Die Anfrage würde sich über drei Services erstrecken, was zusätzliche Latenz mit sich bringt.[30]

Ebenfalls entsteht durch jede Schnittstelle eine fachliche Abhängigkeit.[12] Die Anfragen von den Microservices A, C, D, E müssen der Schnittstellendefinition von B entsprechen. Änderungen führen gegebenenfalls zu Fehlern und weiteren Abhängigkeiten. Wie diese Problematik gelöst werden könnte, wird in Abschnitt 3.5.2 beschrieben.

3.5.2 Asynchrone Kommunikation

Wie bereits beschrieben, wird bei der synchronen Kommunikation auf weiterführende Abfragen gewartet. Die asynchrone Kommunikation wartet nicht auf Antworten von

weiteren Services, sondern trifft Annahmen über etwaige Systemzustände.[30] Um Annahmen zu treffen, existieren je nach Anwendungsfall verschiedene Strategien:

1. Ein Microservice kann replizierte Daten vorhalten. Angenommen ein Artikel soll rausgeschickt werden: Der dafür verantwortliche Service benötigt die Anschrift des Kunden, aber nicht weitere Daten wie Geburtsdatum, Zahlungsmethode oder ähnliches. Dementsprechend werden nur relevante Daten repliziert vorgehalten. Eine Herausforderung ist es, dass diese replizierten Daten stets mit den Originaldaten übereinstimmen. Schließlich kann sich eine Anschrift ändern.[30]

2. Ggf. muss nur ein weiterer Service informiert werden, wie der Service B aus Abschnitt 3.5.1, welcher Statistiken erfasst. In dem Szenario der asynchronen Kommunikation würde die Abfrage gestellt werden, ohne das Ergebnis abzuwarten, da es schlichtweg nicht relevant ist. Die Herausforderung hier ist, zu gewährleisten, dass die Abfrage auch in Fehlerfällen früher oder später zugestellt wird.

Aus den Strategien ergeben sich Anforderungen an die Kommunikationsstruktur: Es muss gewährleisten sein, dass fehlerhafte Abfragen erneut übermittelt werden und ebenfalls wird eine Struktur benötigt, die dafür sorgt, dass replizierte Datensätze stets mit aktuellen Daten befüllt sind. Dies lässt sich durch sogenannte Events erreichen.[12][30]

Die folgende Abbildung 7 verdeutlich das Prinzip von Events und deren Infrastruktur.

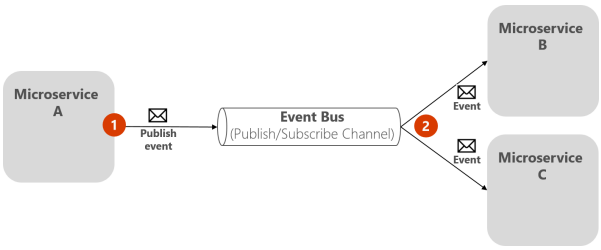

Abbildung 7: Eventbus mit Events[22]

In dieser Abbildung haben Microservice B und C das Event x beim Ergeinisbus (**Event Bus**) abonniert (**subscribed**). D.h. Microservice A veröffentlicht (**published**) eine Änderung, woraufhin B und C informiert werden. B und C können nun ihren Datenbestand aktualisieren und halten so die aktuellen Daten vor. Parallel kann der Microservice A seine Abfrage ganz normal weiterführen. Der Eventbus ist dementsprechend ein Vermittler (**Message Broker**), welcher garantiert, dass die Nachrichten übertragen werden. Dieser sollte mit etwaigen Fehlerfällen (z.B. C ist nicht erreichbar) umgehen können und eine spätere Übertragung garantieren.[22][30]

Aus diesem Modell ergibt sich ein weiterer Vorteil: Die Microservices sind entkoppelt. Es wird keine REST-Schnittstelle definiert, welche eine gewissen Fachlogik vorgibt. Ebenfalls können mehrere Services auf ein Event hören. Wolff warnt allerdings davor

Events unnötig aufgebläht zu gestalten: Zum einen werden schnell Daten übermittelt, die nicht für alle Abonnenten (**Subscriber**) relevant sind und zum anderen entspräche dies nicht dem Prinzip vom DDD.[30]

Zusätzlich sollte beachtet werden, dass Microservices so gestaltet werden, dass sie idempotent sind. In diesem Zusammenhang bedeutet dies, dass falls ein selbes Event zweimal übertragen wird, der Microservice die Aktion nicht zweimal ausführt. D.h. eine Mehrfachausführung führt zu demselben Ergebnis wie eine einzige Ausführung. Wenn z.B. eine Rechnung versendet werden soll, ist garantiert, dass diese nur ein einziges Mal versendet wird.[30]

3.5.3 Abwägung asynchrone vs. synchrone Kommunikation

Aus den zwei vorherigen Abschnitten ergibt sich folgende Aufstellung (siehe Tabelle 4).

Es lässt sich feststellen, dass die Vorteile einer asynchronen Kommunikation für Microservices überwiegen, weshalb diese Kommunikationsart auch empfohlen wird.[30][12] Dennoch ist die Kommunikation nicht dogmatisch zu betrachten, sondern sollte je nach Projekt und Anwendungsfall entschieden werden. Synchrone Kommunikation bietet sich nämlich gerade dann an, wenn der Datenbestand definitiv konsistent sein soll.

Das sogenannte CAP-Theoreom beschreibt die Abwägung, welche man in verteilten Systemen bei der Auswahl der Kommunikation tätigen muss. CAP bedeutet:

- Consistency (Konsistenz): Die Daten in einem verteilten System sind konsistent.
- Availability (Verfügbarkeit): Die Verfügbarkeit für alle Systeme ist gegeben.
- Partition Tolerance (Partitionstoleranz): Das Gesamtsystem arbeitet auch weiter, wenn Teile davon ausfallen.

In einem verteilten System können immer nur zwei von den drei Bedingungen erfüllt sein.[30] Sofern Konsistenz gewährleistet werden soll, müssen alle Dienste stets verfügbar sein. Damit kann der Punkt Partitionstoleranz nicht mehr erfüllt sein. Umgekehrt: Wenn die Partitionstoleranz garantiert ist, z.B. dadurch dass Services ihre eigene Datenhaltung besitzen, ist zwar prinzipiell auch die Verfügbarkeit gegeben, aber nicht die Konsistenz.

Dementsprechend ist es sinnvoll, sich die System-Anforderungen zu überlegen und aufgrund dieser Basis zu entscheiden, welche Kommunikationsart implementiert werden soll.

3.5.4 API Gateway

Umso mehr Services aufgesetzt werden, desto komplexer ist es, die Übersicht über alle zu behalten. Eine Abhilfe im Routing bietet ein sogenanntes *API Gateway*. Ein API

	synchrone Kommunikation	asynchrone Kommunikation
Vorteile	• Jederzeit konsistent • Paradigma ist Entwicklern bekannt[30]	• Entkoppelt durch Events • Flexibilität, da ein Event mehre Services erreichen kann • Nachrichtenempfang garantiert (ggf. mit Verzögerung) • Absicherung gegen Ausfall
Nachteile	• Anfälligkeit durch Abhängigkeiten • Erweiterbarkeit ist schwerer, da fachliche Abhängigkeiten • Ggf. lange Netzwerkzeiten	• nicht jederzeit garantiert konsistent • Idempotenz muss beachtet werden

Tabelle 4: synchrone vs. asynchrone Kommunikation

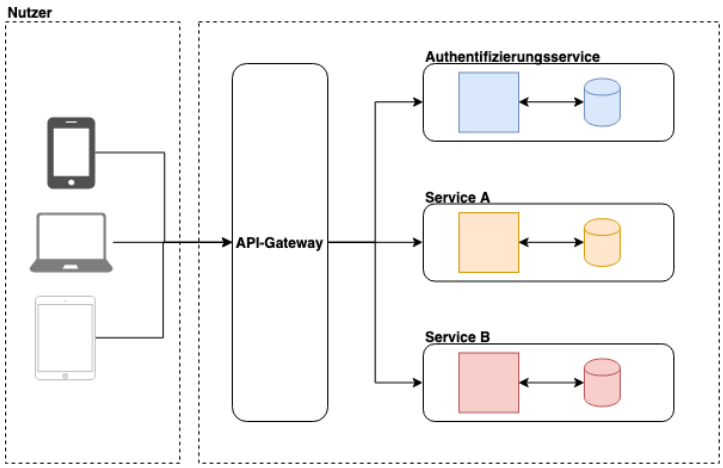

Abbildung 8: Prinzip API Gateway

Gateway ist der einzige Einstiegspunkt für den Nutzer (**Client**). Von dort wird er weitergeleitet, ohne die Routen von einzelnen Services zu kennen.

Abbildung 8 zeigt deutlich, wie die Nutzer über das API Gateway anfragen, welches anschließend an die entsprechenden Services weiterleitet. Wenn APIs eine hohe Auslastung haben, würde man mehrere Instanzen von einem API Gateway erstellen. Ein sogenannter Load Balancer wäre der Einstiegspunkt für die Clients. Dieser würde die Anfragen sinnvoll an die API Gateway Instanzen verteilen, so dass keine Überlastung entsteht.[28]

API Gateways haben neben dem einzelnen Einstiegspunkt noch weitere Vorteile:

- Höhere Sicherheit darüber, dass einzelne Services nicht sichtbar und nur über das Gateway zu erreichen sind.[12]
- Authentifizierung kann bereits im Gateway ausgeführt werden, dies führt zu weniger Last für einzelne Microservices.[30]
- Zentralisiertes Logging, Caching, Monitoring, Mocking sowie eine zentralisierte Dokumentation ist möglich.[30]

Ein Nachtteil in der Struktur des API-Gateways ist, dass die Abfragen länger sind, da sie immer erst über das Gateway gehen.

3.6 Authentifizierung und Autorisierung

Beim Monolithen ist architektonisch klar, dass die Authentifizierung und Autorisierung innerhalb des Monolithen stattfindet. Im Bereich der Microservice-Architektur existieren verschiedene Szenarien, wie man eine Authentifizierung sowie Autorisierung gestalten

kann.[12]

Authentifizierung: Identifiziert, wer jemand ist. Z.B. Nutzer A, der sich durch Benutzername und Passwort registriert hat.[12]

Autorisierung: Bestimmt, wie viel ein Nutzer darf. Nutzer A hat eine Rolle, welche ihn berechtigt, gewisse Aktionen durchzuführen.[12]

Wenn möglich und sinnvoll, kann die Autorisierung auch bereits im Gateway geschehen. Häufig muss allerdings ein Microservice diese selbst durchführen. Teils ist dies begründbar durch den Datenbestand, welcher nur dem Microservice vorliegt.

Um unnötige Last zu verhindern, kann die Validität (handelt es sich um einen gültigen Request) und die Authentifizierung bereits im API Gateway überprüft werden. Der weitere Vorteil ist, dass die Authentifizierung somit an einer zentralen Stelle durchgeführt wird. Dies verhindert Redundanz und fehlerhafte Implementierungen.[7][19]

Die Idee ist, dass der Benutzer nach dem Anmelden ein Security-Token erhält, welches verwendet wird, um sensible Anfragen zu verifizieren. Zum einen gibt es die Möglichkeit, ein Token auszustellen, welches beim Auslesen verschlüsselt ist (opaque Token) und zum anderen auf einen offenen Standard namens *JSON Web Token* (JWT, transparent Token) zu setzen. Das opaque Token hat den großen Nachteil, dass es zusätzliche Performance sowie Latenz verursacht und nur synchron entschlüsselt werden kann.[19]

Das JWT wird beim Ausstellen signiert, um die Echtheit zu gewährleisten. Während die Nachteile des opaque Tokens hier nicht auftreten, ist ein anderes Problem, dass ein JWT nach Ausstellung nicht widerrufen werden kann. Theoretisch wäre es dauerhaft gültig, weshalb Ablauflaufzeiten gesetzt werden. Dies wiederum impliziert, dass der Client dafür sorgen muss, immer rechtzeitig ein neues Token anzuforden. Für solche und weitere Logiken exisitiert bereits ein Sicherheitsstandard namens OAuth2, welcher empfohlen wird zu verwenden.[19]

Ziel bei OAuth2 ist unter anderem, Autorisierungen zwischen verschiedenen Anwendungen zu erlauben. Ursprünglich wurde das Authentifizierungsprotokoll so entworfen, dass Drittanwendungen Zugang zu Informationen erhalten, ohne dass Passwörter weitergeleitet werden müssen.[19] Beispielsweise wird OAuth2 verwendet, wenn Benutzer sich über ihren Facebook-Account bei Drittplattformen anmelden. Die Drittplattformen können natürlich nicht das Facebook-Passwort einsehen, erhalten aber je nach Anwendungsfall Zugriff auf verschiedene Ressourcen (z.B. Lesezugriff auf die E-Mail-Adresse und/oder Konakte, Schreibzugriffe zum Teilen von Nachrichten usw.).

Da OAuth2 ein sehr komplexes und umfangreiches Thema ist, wird im Folgenden nur ein häufig verwendetes Grundprinzip erklärt.

Um die Abbildung 9 besser zu verstehen, sind folgende Definitionen hilfreich:[19]

Authorization Server: Authentifiziert den Benutzer und gibt ein Access sowie Refresh Token raus.

Access Token: Durch ein Access Token erhält man Zugriff auf den Resource Server. Das Format ist abhängig von der jeweiligen Implementierung, eine bereits genannte Möglichkeit wäre JWT. Der Access Token ist zeitlich begrenzt gültig.

Refresh Token: Ein Token, welches langlebig ist, also eine lange Gültigkeit besitzt. Dieses kann allerdings im Gegensatz zum Access Token widerrufen werden. Ebenfalls wird es verwendet, um ein neues Access Token vom Authorization Server anzufordern. Dafür ist keine Übergabe der Benutzerdaten nötig.

Resource Server: Eine Ressource auf die nur zugegriffen werden kann, wenn ein valides Access Token vorliegt. Eine Ressource könnte z.B. ein Microservice sein.

Client: Ein Client möchte Zugriff auf den Resource Server. Clients können beispielsweise Drittanwendungen, Webanwendungen oder mobile Applikationen sein. Sie werden auch Resource Owner genannt.

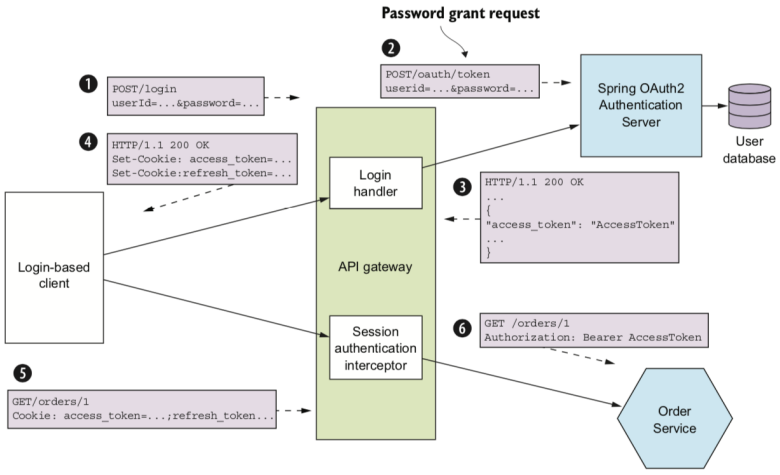

Abbildung 9: Ablauf eines sogenannten ‘password grant‘[19]

Die Abbildung 9 zeigt einen Ablauf, bei dem ein Client seinen Usernamen und sein Passwort übermittelt (1). Diese Anfrage wird von dem API Gateway weitergeleitet an einen Authorization Server (2). Wurde der Nutzer gefunden, wird ein Access und Refresh Token über das API Gateway (3) an den Nutzer übermittelt (4). Mit der entsprechenden Autorisierung ruft der Client seine Bestellungen (5) ab. Dieser Request wird entsprechend von dem API Gateway weitergeleitet (6).

Neben dem *password grant Flow* existiert auch der sogenannte *client credentials grant Flow*. Dieser funktioniert ähnlich, nur dass nicht Username und Passwort übertragen werden, sondern geheime Daten für die Authentifikation zwischen Anwendung und Authorization Server bekannt sind. Den *client credentials grant Flow* sollte man verwenden, wenn eine Applikation Ressourcen aufrufen möchte, die außerhalb eines User-Kontextes liegen, d.h. es handelt sich in der Regel um recht allgemeine Daten.[16][19]

Auf Basis dieses Abschnittes 3.6 wird herausgearbeitet, welche Möglichkeiten zur technischen Umsetzung von OAuth2 bereitstehen (siehe 4.5).

3.7 Docker

Über Docker können Anwendungen innerhalb eines Containers laufen. Ein Container ist vergleichbar mit einer sehr leichtgewichtigen, modularen virtuellen Maschine.[29] Docker in der Tiefe aufzuarbeiten, würde den Umfang dieser Arbeit übersteigen. Dennoch wird im Folgenden erläutert, warum speziell Docker sich sehr gut für Microservices eignet und wie das grundsätzliche Wirkungsprinzip funktioniert.

Anfangs wurden Microservices so definiert, dass sie als möglichst eigenständige, deploybare Einheiten zu betrachten sind. Beim Hosten - also dem Bereitstellen des Services - sollte dies ebenfalls berücksichtigt werden. Nimmt man an, man hostet alle Services auf einer Maschine, läuft man Gefahr, dass die Microservices sich z. B. durch Portkonfigurationen oder dem Zugriff auf selben Ressourcen behindern.[31]

Als bisherige Lösung boten sich an dieser Stelle virtuelle Maschinen (**VM**) an. D.h. auf einem Host könnten mehrere Betriebssysteme laufen, die sich die Hardwareressourcen vom Host teilen. Die angestrebte Isolation zwischen den Microservices wäre erreicht und individuelle Konfigurationsmöglichkeiten könnten sich gegensetitig nicht mehr stören. Durch die VMs entstehen allerdings Leistungseinbußen, welche nicht im Verhältnis zum Nutzen stehen und zusätzlich einen höheren Verbrauch der Hardwareressourcen mit sich bringen.[31] Gesucht ist dementsprechend eine Technologie, die es schafft, Services zu isolieren und dabei gleichzeitig leichtgewichtig zu sein: Docker.

Startet man einen Microservice über Docker, ist dies damit gleichzusetzen, als würde im Betriebssystem der Service als Prozess gestartet werden. Es entsteht kein deutlich sichtbarer Overhead in Bezug auf den Ressourcenverbrauch.[31]

Um den Docker-Aufbau (siehe Abbildung 10) besser zu verstehen, ist folgendes Vokabular nützlich:[31]

Docker-Image: Aus einer Anwendung kann man ein Image erzeugen, so dass es im Docker-Container gestartet werden kann.

Docker-Container: Wenn ein Image ausgeführt wird, läuft es in einem Container, welcher verschiedene Eigenschaften hat, wie z.B. ein eigenes Netzwerk-Interface und ein Dateisystem.

Dockerfile: Ein Dockerfile ist ähnlich einem Bauplan. Er beschreibt, wie das Image gebaut werden muss, damit es in einem Docker-Container laufen kann.

Repository: Ein Repository speichert Images. In der Regel hält ein Repository mehrere Images von derselben Anwendung mit verschiedenen Versionen bereit.[29]

Docker-Registry: Eine Docker-Registry verwaltet mehrere Repositories.

Docker-Host: Ein Docker-Host unterstützt die Docker-Technologie, d.h. es können entsprechend Images ausgeführt werden. Zusätzlich ist es möglich, Images von einem Repository auszuführen. D.h. man muss die Images nicht extra auf den Host laden, um sie auszuführen.

Bei genauerer Betrachtung der Abbildung 10 fällt auf, dass Container in einem Netzwerk sind und sich denselben Kernel teilen. Dateisysteme, Netzwerk-Interface und containereigene Prozesse sind voneinander isoliert. D.h. Portfreigaben oder ähnliche Konfigurationen behindern sich nicht.

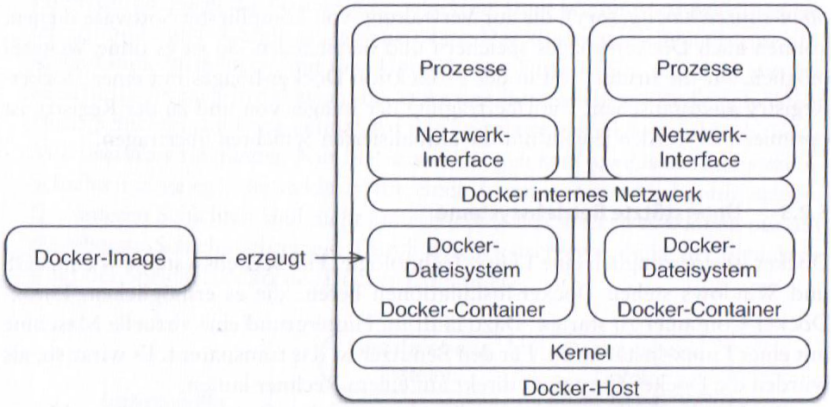

Abbildung 10: Docker Architektur[31]

Zusammenfassend lässt sich sagen, dass sich durch das Nutzen von Docker eine isolierte und ressourcenvertretbare Trennung von Microservices auf einem Host erreichen lässt.

4 Konzept

In dem folgenden Abschnitt werden verschiedene Anforderungen an die Stirnraten API mittels User Story Mapping erfasst. Anschließend werden verschiedene Technologien und Strategien zum Umsetzen der Kommunikation, des Gateways und der Authentifizierung/Autorisierung gegeneinander abgewogen und festgelegt.

4.1 Anforderungen definieren

Um die Anforderungen für das Spiel *Stirnraten* zu erfassen, sollten zwei verschiedene Aspekte berücksichtigt werden:

- Der **Ist-Stand**, d.h. was mindestens erfüllt werden muss und
- welche möglichen **Erweiterungen** durch eine API entstehen.

Um die Anforderungen greifbarer zu gestalten, wird auf das Prinzip von User Story Mapping zurückgegriffen. D.h. jede Anforderung ergibt sich aus einer sogenannten User Story. Diese ist so aufgebaut, dass Folgendes beschrieben wird: **Wer** möchte **was** und **aus welchem Grund**.[17]

Im Folgenden gelten die zwei Definitionen: Ein Nutzer ist eine Person, welche die App spielt. Der Betreiber ist der Besitzer von Stirnraten. Ein User kann ein Nutzer oder Betreiber sein.

Ein Beispiel für eine User Story könnte lauten: Als Nutzer (**wer**) möchte ich mein Spielprofil teilen (**was**), um mich besser mit meinen Freunden messen zu können (**warum**).

Wie diese User Story nun umgesetzt wird, muss abgewogen werden. Zum einen sollten User Stories konkret genug formuliert werden, so dass klar ist, was der User möchte. Zum anderen bleibt bei der Entwicklung ein agiler Handlungsspielraum.[17] Eine Teilen-Funkion beispielweise kann unterschiedlich aufwendig umgesetzt werden. Der Nutzer könnte einen Text teilen, ein extra aufbereitets Bild oder einen Link, welcher auf ein mögliches Online-Profil verweist. All diese Möglichkeiten bedeuten unterschiedliche Aufwände. Alternativ könnte man aus dieser einen User Story drei erstellen, welche entsprechend unterschiedlich priorisiert werden.

Exkurs - Spielprinzip Stirnraten: Die Spieleranzahl muss mindestens zwei betragen. Ein Spieler entscheidet sich für eine oder mehrere Kategorien und hält sich das Telefon an die Stirn. Es erscheint ein Begriff, welchen sein Gegenüber erklären muss. Errät der Spieler den Begriff, neigt er das Telefon nach vorne und ein neuer Begriff erscheint. Weiß er ihn nicht, kann er diesen überspringen, indem er das Telefon nach hinten neigt. Ziel ist es, innerhalb einer frei wählbaren Zeit (z.B. 60 Sekunden) so viele Begriffe wie möglich zu erraten.

4.1.1 Erfassung Stirnratens Ist-Stand

In der folgenden Tabelle 5 wird gezeigt, welche Funktionen die App bereits auf dem Gerät bereitstellt, welche aber zukünftig serverseitig erledigt werden sollen.

Funktion	Beschreibung
Profil/Statistik	Nach jedem Spiel werden verschiedene Daten erfasst, z.B. die Dauer des Spiels oder richtig geratene Wörter. Löscht man die App, ist dieses Profil unwiederbringlich.
Bereitstellung Begriffe	Die über 6000 verschiedenen Begriffe liegen nur offline zur Verfügung. Editieren, Hinzufügen und Löschen geht nur über das Updaten der App.
Zweisprachigkeit	Die App wird für den deutschen sowie den englischen Sprachraum angeboten. Es ist gewährleistet, dass je nach Nutzer, auf die sprachlich richtige Datenbank zugegriffen wird.

Tabelle 5: Bestehende Funktionen in Stirnraten

Aus dem Ist-Stand ergeben sich bereits folgende User Stories:

- Als Betreiber möchte ich neue Begriffe über eine Schnittstelle hinzufügen, editieren und löschen können, um die Datenbank schneller und leichter zu pflegen.
- Als Betreiber möchte ich eine Datenbank nutzen, um nicht den Datenbestand zu pflegen für zwei Apps (iOS und Android).
- Als Betreiber möchte ich entscheiden können, in welcher Sprache (Englisch oder Deutsch) ich Begriffe manipuliere, um sinnvolle Daten zu gewährleisten.
- Als Nutzer möchte ich das Spiel immer offline spielen können, da ich auf Reisen häufiger kein stabiles Internet habe.
- Als Nutzer möchte ich mein Spielerprofil online speichern, um es auf anderen Geräten oder nach einer Neuinstallation abrufen zu können.
- Als Nutzer möchte ich, automatisch die Sprache angezeigt kriegen, welche für mich relevant ist, weil es mir sonst zu kompliziert ist.

4.1.2 Erweiterungen mittels User Story Mapping

Durch das Einführen einer API bieten sich folgende Erweiterungsmöglichkeiten an:

- Als Betreiber möchte ich neue Kategorien hinzufügen, editieren und löschen können, um das Nutzerangebot zu vergrößern.

- Als Betreiber möchte ich eine Kategorie als Premium kennzeichnen können, um Angebotsaktionen zu schalten.
- Als Betreiber möchte ich eine Kategorie (de)aktivieren können, um sie immer zu einem sinnvollen Zeitpunkt anbieten zu können.
- Als Betreiber möchte ich eine Registrierfunktion anbieten, um die Nutzer stärker an mich zu binden.
- Als Betreiber möchte ich die Nutzer abrufen, welche sich bei mir registriert haben, um einen Nutzerstamm aufzubauen.
- Als Betreiber möchte ich Nutzer aus Datenschutzgründen löschen können.
- Als Betreiber möchte ich sehen, wenn ein Begriff bereits in der Kategorie ist, um die Datenqualität zu gewährleisten.
- Als Nutzer möchte ich eigene Begriffe einreichen können, weil mir manche Begriffe oder Kategorien im Spiel fehlen.
- Als Nutzer möchte ich sehen, wenn ein eingereichter Begriff bereits in einer Kategorie existiert, um Bescheid zu wissen.
- Als Betreiber möchte ich, dass Nutzer-Zugangsdaten entsprechend gut verschlüsselt sind, um die Datensicherheit zu gewährleisten.

Die folgende Auflistung sind User Stories, welche auch als Anforderungen entstanden sind, aber im Rahmen der Projektarbeit aufgrund von Aufwänden nicht umgesetzt werden können.

- Als Betreiber möchte ich eine Newsletter-Funktion anbieten, um die Nutzer über Neuigkeiten zu informieren
- Als Nutzer möchte ich ein Profilbild hochladen, um mein Profil zu indiviualisieren.
- Als Nutzer möchte ich mein Passwort zurücksetzen können, wenn ich es vergessen habe.
- Als Betreiber möchte ich individuelle Animationen vom Server an den Nutzer weiterreichen können, um die Verspieltheit der App zu unterstreichen.
- Als Betreiber möchte ich Themes und Farbcodes online bereitstellen, um den Nutzern Individualsierungsmöglichkeiten schneller und leichter bereitzustellen.
- Als Betreiber möchte ich automatisiert individuelle Nachrichten senden, um den Nutzer stärker zu binden.
- Als Betreiber möchte ich Bilder pro Kategorie hinzufügen, editieren und löschen können, um ein ansprechendes Bild für die Nutzer zu hinterlegen.

- Als Nutzer möchte ich mich in einer Rangliste mit anderen Nutzern vergleichen können, um zu sehen, wer in dem Spiel besser ist.
- Als Nutzer möchte ich die Spielerprofile von anderen Nutzern detailliert ansehen, um zu sehen, was ihnen gefällt.
- Als Betreiber möchte ich die Ranglisten-Namen der Nutzer manipulieren können, um unflätige Namen/Missbrauch zu verhindern.
- Als Betreiber möchte ich kumulierte Daten aus den Nutzerstatistiken sehen, um Marktentscheidungen besser treffen zu können.
- Als Betreiber möchte ich die Kategorien sortieren können, um die Anordnung für die Nutzer bestmöglich zu gestalten.
- Als Betreiber möchte ich eingereichte Begriffe zulassen oder ablehnen können, um den Datenbestand zu vergrößeren bzw. die Qualität zu gewährleisten.
- Als Betreiber möchte ich sehen, wann meine Nutzer zuletzt online waren, um ggf. Marketingmaßnahmen zu unternehmen.

Aus den User Stories ergeben sich konkrete Abhängigkeiten zwischen den Microservices sowie klare Vorlagen für die Datenhaltung.

4.2 Macroarchitektonische Festlegungen von Technologien

Wie bereits in Abschnitt 3.4.2 erwähnt, können durch makroarchitektonische Entscheidungen gewisse Vorteile erzielt werden, wie z.B. dass die Technologien zur Infrastruktur des Unternehmens und zu den Kompetenzen der Mitarbeiter passen. Ebenfalls können strategische Entscheidungen (z.B. das ausschließliche Nutzen von Cloudtechnologien) die Makroarchitektur beeinflussen. Im Folgenden werden einige Technologien für *Stirnraten* makroarchitektonisch festgelegt.

4.2.1 Wahl der Datenbank

Bei der Wahl zwischen einer relationalen oder schemalosen (NoSQL) Datenbank wird sich für eine relationale entschieden. NoSQL Datenbanken sind häufig für spezielle Anwendungsfälle sinnvoll, z.B. wenn das Datenbankmodell sich häufig ändert oder ein hohes Maß an Skalierung notwendig ist. Diese Fälle sind für Stirnraten nicht absehbar, weshalb auf den etablierten Standard einer relationalen Datenbank gesetzt wird.[11]

Im Bereich der relationalen Datenbanken können verschiedene Technologien zur Umsetzung genutzt werden. Diese vier stehen im Fokus (Stand Mai 2019): Oracle (Rang 1), MSSQL (2), MySQL (Rang 3) und Postgres (Rang 4).[10] Oracle und MSSQL werden für das Projekt ausgeschlossen, da diese kostenpflichtig betrieben werden müssten. Für Postgres und MySQL wird ein sogenanntes Proof of Concept erstellt, d.h. es wird in

einem einfachen Szenario eine Machbarkeit überprüft. Die Grundanforderungen ist, dass MySQL und Postgres in verschiedenen Docker-Containern auf einer Maschine laufen können. Bei dem Proof of Concept zeigt sich, dass es deutlich komplizierter ist, multiple Postgres Instanzen auf einer Maschine zu starten, da zusätzlich individuelle Scripts ausgeführt werden müssen.[18]

Beim Erstellen von multiplen MySQL-Instanzen sind keinerlei Probleme aufgetreten. Aufgrund des Rankings und der einfacheren technischen Implementierung durch das Proof of Concept verwenden die Microservices MySQL.

4.2.2 Programmiersprachen, Darstellungsart, REST und Docker

Programmiersprachen: Anfangs wurde erwähnt, dass aufgrund unternehmensstrategischer Gründe Entscheidungen darüber getroffen werden, welcher Technologiestack verwendet wird. In Hinblick auf die Programmiersprachen wird deshalb festgelegt, dass die Microservices im .NET Framework in c# entwickelt werden. Der Entwickler beherrscht diese Sprache am besten, so dass Wartbarkeit, Nachhaltigkeit und Pflege des Codes langfristig garantiert sind. Als zusätzliche Alternative ist kotlin ebenfalls erlaubt.

Darstellungsart: Das verwendete Datenformat beim Austausch von Daten ist JSON (JavaScript Object Notation). Alternativ wäre auch die Extensible Markup Language (XML) möglich, allerdings ist XML deutlich aufgeblähter und damit weniger leichtgewichtig. Zusätzlich lässt JSON sich besser von den Programmiersprachen weiterverarbeiten.[13]

REST: Die Kommunikation zwischen den Microservices unterliegt Representational State Transfer-Paradigma (REST). Eine Alternative zu REST wäre SOAP (Simple Object Access Protocol) in Kombination mit WSDL (Web Services Description Language). Da WSDL allerdings auf der Basis von XML arbeitet und SOAP deutlich komplexer und schwerer skalierbar als REST ist, wird es nicht verwendet. [2]

Die Prinzipien von REST sind bereits aus dem Modul *Mobile Application Development* bekannt und werden deshalb nicht weiter erwähnt.

Docker: Neben Docker als Containerisierung existieren einige Alternativen wie Podman, Rocket, LXD, Flockport, Windocks oder Boxfuse. Sie unterscheiden sich teils in Sicherheitsaspekten, Preis, Kompatibilität zum Betriebssystem oder der Anbindung zu Kubernetes (Programm zum Bereitstellen, Skalieren und Verwalten von Container-Anwendungen).[4][23] Die Entscheidung fällt auf Docker. Zum einen da dies - wie bereits bei den Programmiersprachen - eine beherrschte Technologie ist. Zum anderen - gemessen am Google Trend - ist Docker die bevorzugt gesuchte Technologie für Containerisierung:

Abbildung 11: Trends bei den Suchworten: Docker, Rocket Container, LXD, Flockport und Podman

4.2.3 Bounded Contexts - Architektur des Projektes

Mit Hilfe des Wissens aus dem Abschnitt 3.4.1 lassen sich folgende Bounded Contexts erstellen, welche in Abbildung 12 dargestellt sind.

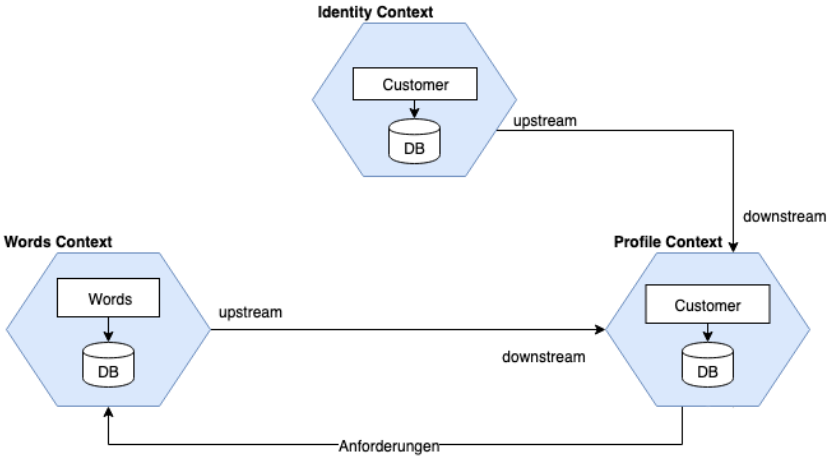

Abbildung 12: Bounded Contexts für die Stirnraten API.

Durch diese verschiedenen Kontexte lassen sich entsprechende Microservices abbilden. Zusätzlich wird definiert, was unter welchen Fachtermini im entsprechenden Kontext zu verstehen ist. Zum Beispiel stellt der *Identity Context* dem Nutzer ein Token aus, mit dem er weitere Aktionen ausführen darf. Dafür speichert der *Identity Context* sich in seinem Customer-Modell (Domänenmodell) einen Namen und ein Passwort. Der Customer im *Profile Context* dagegen enthält noch weitere Informationen wie z.B. Anzahl der gespielten Spiele, geratene Begriffe, Lieblingskategorien und Spielminuten. Der Customer im *Rank Context* benötigt dagegen nur den Customer-Namen und noch zu definierende Parameter, aus denen Customer-Punkte generiert werden können, um eine Bestenliste darzustellen.

Wie bereits in dieser Arbeit erwähnt, stellt der *upstream* dem *downstream* Informationen bereit. Zusätzlich können Anforderungen an Kontexte gestellt werden, damit mit den empfangenen Daten entsprechend gearbeitet werden kann. In der Abbildung 12 fordert der *Profile Context* Informationen vom *Words Context*.

Um die aus den User Stories entstandenen Anforderungen zu erfüllen, werden im Folgenden die Domainmodels konzeptioniert. Diese können während der Implementierungsphase noch abweichen.

Words-Context: Words Domainmodel
id: Dient als eindeutige Identifikation (unique identifier).
category_name: Name der Kategorie.
subtitle: Optionale Beschreibung der Kategorie.
premium_key: Notwendig für die Kaufabwicklung, um zu identifizieren, um welchen In-App-Kauf es sich handelt.
is_premium: Markiert, ob eine Kategorie kostenpflichtig ist oder nicht.
selected: Definiert, ob die Kategorie in der App vorausgewählt ist oder nicht.
words: Die Begriffe pro Kategorie, welche erraten werden können.
sort: Definiert, an welcher Stelle in der Sortierung die Kategorie ist.
is_activ: Definiert, ob die Kategorie in der App angezeigt wird oder nicht.
updated_at: Zeitpunkt, wann die Kategorie das letzte Mal verändert worden ist.

Identitiy-Context: Customer-Domainmodel:
id: Dient als eindeutige Identifikation (unique identifier).
name: Name des Benutzers.
password: Passwort des Benuzters.

Profile-Context: Customer-Domainmodel:
id: Dient als eindeutige Identifikation (unique identifier).
name: Name des Benutzers (kommt aus dem Identitiy-Context).
mail: E-Mail-Adresse des Benutzers (optional).
played_games: Zeigt die Anzahl absolvierter Spiele.
most_right_words: Zeigt die Anzahl der Wörter, die während einer Runde geraten worden sind.
most_skipped_words: Zeigt die Anzahl der Wörter, die während einer Runde übersprungen worden sind.
right_words: Zeigt die Anzahl aller Wörter, die richtig geraten worden sind.
skipped_words: Zeigt die Anzahl aller Wörter, die übersprungen worden sind.
time: Zeigt, wie lange der Benutzer gespielt hat.
top_categories: Speichert, welche Kategorien der Benutzer favorisiert.

Es lässt sich feststellen, dass es Überschneidungen zwischen dem Customer-Domainmodel gibt, je nachdem in welchem Kontext man sich befindet. Natürlich verändert dieses Konzept sich noch im Laufe der Produktentwicklung und muss iterativ an den Stand der Entwicklung angepasst werden. Ebenfalls gilt zu erwähnen, dass es an dieser Stelle im Domain Driven Design nicht zwangsweise ein richtig oder falsch gibt. Dies ist ein Konzeptentwurf, aber natürlich nicht die einzige mögliche Lösung.

4.3 Wahl des API Gateway

Bei der Wahl des API Gateways existieren verschiedene Technologien, die gegeneinander abgewogen werden müssen. Dabei werden unterschiedliche Aspekte betrachtet: Zum einen sollte das API Gateway etabliert und leichtgewichtig sowie gemessen an der Projektgrößte einfach zu implementieren sein. Zusätzlich muss das Gateway ein OAuth2 Token verarbeiten können. Um das Gateway umzusetzen, bieten sich verschiedene Möglichkeiten an, wie z.B. die Eigenentwicklung, das Einbinden einer Library oder das Nutzen von Lösungen, die von Clouddiensten wie AWS, Azure oder Google Cloud bereitsgestellt werden.

Eigenentwicklung: Die eigene Entwicklung eines Gateways ist kritisch zu betrachten, da viele aktuelle, umfangreiche und bereits etablierte Libraries für diesen Einsatzzweck existieren. Schätzungsweise kostet es viel Zeit und Energie die Funktionen, die ein Gateway erfüllen muss, technisch sauber umzusetzen. Beispiele für Funktionen eines Gateways sind: Routing, Caching, Load Balancing, Headers/Query und String/Claims Transformation, Logging ...

Libraries: Eine etablierte und leichtgewichtige Lösung ist das API Gateway Zuul, welches allerdings auf Java basiert und dementsprechend aus macroarchitektonischer Sicht nicht bevorzugt wird.

Eine weitere Möglichkeit ist ein Gateway mit **Istio** in Kombination mit Kubernetes aufzubauen. Das hätte unter anderem den großen Vorteil, dass die Services sich untereinander kennen und die Kommunikation sehr wenig händische Konfigurationen benötigt. Allerdings erfordert Kubernetes sowie Istio jeweils ein hohes Maß an technischem Verständnis und erscheint nicht lohnenswert für ein eher kleines Projekt.[21][26]

Von Microsoft empfohlen wird das Open Source Gateway **Ocelot**. Es ist leichtgewichtig und überschneidet sich mit gesuchten Anforderungen (Routing, Load Balancing, Authorization usw.). Zusätzlich ist es explizit desigend für ASP .NET Core und bietet so eine überschaubare Implementierung vom Aufwand her.[5]

Cloudanbieter: AWS, Azure und Google Cloud bieten alle Gateways an, die auch einen entsprechend großen Funktionsumfang garantieren. Ein weiterer Vorteil ist die Skalierbarkeit, die jeder Cloud-Anbieter verspricht. Ebenfalls wird die Programmiersprache c# im .NET Kontext unterstützt.[25][24] Ein Nachteil dagegen ist die preisliche Komponente. Es ist nicht absehbar, wie sich die Stirnraten API bezüglich der Last entwickelt, weshalb unsicher ist, wie viele Kosten entstehen werden. Auch wenn eine selbstgehoste Lösung andere Nachteile mit sich bringt, ist ein Festpreis garantiert, welcher für eine Projektarbeit bevorzugt wird.

Aufgrund der Recheren wird sich weder für eine Eigenentwicklung (zu aufwendig) noch für eine Cloudlösung (unsichere Kostenfrage) entschieden. Betrachtet man die Vor- und

Nachteile der Libraries, bietet Ocelot den größten Mehrwert, weshalb ein Gateway via Ocelot implementiert wird.

Die Architektur erweitert sich wie folgt:

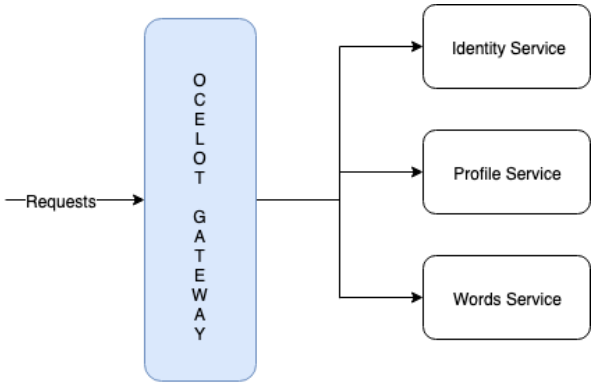

Abbildung 13: Struktur mit dem API Gateway Ocelot

4.4 Wahl der Kommunikation

Um die im Abschnitt 3.5.2 ausgearbeitete asynchrone Kommunikation zu gewährleisten, werden sogenannte Message Broker (kurz: Broker) verwendet. Diese empfangen Nachrichten und senden diese ggf. an mehrere Empfänger weiter. Die folgenden Message Broker sind in der Selbstbeschreibung schnell, robust, zuverlässig und einfach zu implementieren: *RabbitMQ*, *Kafka*, *RocketMQ*, *Artemis* oder *NSQ*.

Bevor die Message Broker jedoch im Detail gegeneinander abgewogen werden, wird untersucht, ob es bereits vorherige Ausschlusskriterien gibt. Zum Beispiel weist die *RocketMQ* (Apache Projekt) noch über 130 Github Issues auf, weshalb davon auszugehen ist, dass dieser Message Broker sich noch in der Entwicklung befindet. *Artemis* verwendet noch ältere Technologien wie XML und *NSQ* bietet nicht den Funktionsumfang wie andere Broker (z.B. Haltbarkeit der Nachrichten oder Clustering).[6] Deshalb werden diese Message Broker von vorneherein ausgeschlossen.

Im Folgenden wird der von LinkedIn entwickelte Broker *Kafka* mit dem *RabbitMQ* Broker verglichen. Auch wenn beide Broker in unterschiedlichen Sprachen entwickelt worden sind, können sie mittels c# verwendet werden und sind Open Source.

Architektonisch arbeitet die RabbitMQ mit einer Entkopplung, da die Produzenten ihre Nachrichten in eine sogenannte Börse (exchange) übermitteln (publish). Die Konsumenten entnehmen die Nachrichten aus einer Queue. So liegt das Routing zwischen Exchange und Queue nicht bei den Produzenten bzw. Konsumenten. Kafka dagegen ist

für ein höheres Volumen auslegt. Im Gegensatz zur RabbitMQ merken die Consumer sich, ob sie bereits eine Nachricht gelesen haben oder nicht. D.h. die Nachrichten werden in einem Kafka Cluster zeitlich begrenzt gespeichert, unabhängig davon, ob sie schon gelesen oder ungelesen sind. Kafka könnte diesbezüglich sinnvoll für Event Sourcing sein, also in einem System, wo der Zustand eines Systems durch Sequenzen von Events abgebildet werden kann.[19] Kafka benötigt im Gegensatz zur RabbitMQ einen externen Dienst (häufig Zookeper verwendet), durch den vereinfacht ausgedrückt mit Kafka kommuniziert werden kann.[8][27] Typische Anwendungsfälle werden bei Kafka beim Messaging, Webseiten-Aktivitäts-Tracking, erfassen von Metriken, Log Aggregationen und Event Sourcing gesehen.[27] RabbitMQ setzt dagegen mehr auf sehr zuverlässige Zustellung der Nachrichten und unterstützt eine Vielzahl von Kommunikationsprotokollen. Zusätzlich lassen sich viele Kafka-Anwendungsfälle (z.B. Event Sourcing) mit Hilfe von Drittsoftware (z.B. Cassandra) in Kombination mit der RabbitMQ abbilden.[8]

Es ist schwierig zu entscheiden, welcher Broker der besser geeinete ist. Beide Technologien sind sehr umfangreich und decken gerade in Kombination mit Drittsoftware viele gleiche Anwendungsfälle ab. Ebenfalls sind die in *Stirnraten* zu erwartetenden Anwedungsfälle im Gegensatz zu den Möglichkeiten, die die Broker bieten, eher trivial. Da Kafka allerdings wie bereits erwähnt ein zusätzliches Programm für die Verwaltung benötigt und im Gesamten größer sowie umfangreicher erscheint, wird auf eine schlankere Implementierung mittels RabbitMQ gesetzt.

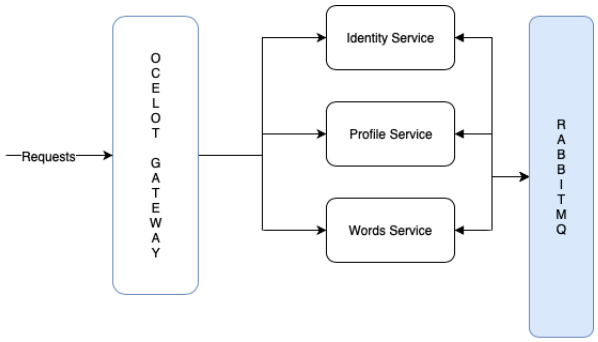

Abbildung 14: Architektur mit Hilfe der RabbitMQ

4.5 Wahl der Authenfizierung und Autorisierung

In Abschnitt 3.6 wurde bereits argumentiert, dass OAuth2 für die Authentifizierung und Autorisierung verwendet wird. Ebenfalls wurde festgelegt, dass jeder Microservices die Autorisierung durchführt, die Authentifizierung allerdings in einem eigenen Service liegen muss. Damit ein Microservice die Autorisierung durchführen kann, muss ein Request vom API Gateway entsprechend aufbereitet werden. Im Detail sieht dies wie folgt aus[19]:

1. Ein Request mit JWT wird vom Gateway empfangen.
2. Das Gateway prüft die Signatur. Ist der Request valide, werden Informationen (z.B. Schreibrecht um neues Wort zu hinterlegen) in den Header vom Request geparst.
3. Das Gateway leitet den Request angereichert mit entpackten Daten im Header weiter an den Microservice.
4. Der Microservice liest die Daten aus und überprüft, ob die nötigen Rechte für die entsprechende Aktion vorliegen.

Während der strukturelle Ablauf nun festgelegt ist, stellt sich die Frage, wie OAuth2 implementiert wird. Dafür stehen verschiedene Möglichkeiten zur Verfügung: Auth0 (Drittanbieter), IdentityServer4, Owin (Framework für ASP .NET) oder cloudbasierte Lösungen.

Auth0: Bei Auth0 (http://auth0.com) handelt sich um einen Drittanbieter, welcher laut eigenen Angaben alle gängigen Authenifizierungsmöglichkeiten abbildet. Die Dienste, welche bereitgestellt werden (z.B. Social Login, Zwei-Faktor-Authentifizierung, E-Mail-Verfication, Forget Password usw.) würden laut eigenen Angaben ca. 90 Tage Eigenentwicklung beanspruchen. Mit Auth0 benötigt man zum Implementieren wenige Stunden und hat Zugriff auf gut gepflegte Dokumenationen. Der Nachteil ist, dass Auth0 ab einer gewissen Last (über 7000 aktive Nutzer im Monat) kostenpflichtig wird. Zusätzlich ist das Angebot, z.B. Zwei-Faktor-Authentifizierung für Stirnraten, nicht notwendig.

.NET OWIN: Allgemein ist OWIN von Microsoft für ASP .NET Core entwickelt und hat den Vorteil, dass Webanwendung und Webserver voneinander entkoppelt sind. Durch OWIN sitzt eine Middleware vor dem Webserver. Dort bietet OWIN die Möglichkeit, Autorisierungsserver basierend auf OAuth2 zu implementieren. Der Dienst ist kostenfrei und auf ASP .NET Core zugeschnitten, leider ist die Dokumentation sehr rudimentär. Es ist schwer herauszufinden, welche Möglichkeiten OWIN genau bietet und wie man diese implementiert.[15]

IdentityServer4: Der IdentityServer4 ist ein OAuth2 Framework für ASP.NET Core, welches in ASP.NET Core 3.0 (derzeitige produktivversion ist 2.2 - Stand 2.6.2019) künftig vorhanden sein soll. Derzeit muss es noch über Libraries installiert werden. Es vereinfacht die Handhabung mit OAuth2, bietet eine umfangreiche Dokumenation und arbeitet unterstützend mit Ocelot zusammen. Es bietet natürlich nicht so eine leichte Implementierung an wie Auth0, ist allerdings kostenfrei.

Clouds: Die Clouddienste Google, AWS und Azure bieten ebenfalls eigene Lösungen an. Auf diese wird allerdings nicht weiter eingegangen, da bei dem Gateway und Hosting bereits auf eine Cloudlösung verzichtet worden ist.

Aus den genannten Möglichkeiten wird sich für das Frsamework IdentityServer4 entschieden, da es kostenlos ist (im Gegensatz zu Auth0), besser dokumentiert als OWIN und mit Ocelot kompatibel ist. Zusätzlich spart man sich gegenüber der kompletten Eigentlichentwicklung Zeit.

5 Implementierung

In dem folgenden Abschnitt wird gezeigt, wie man technisch die Authentifizierung und Autorisierung umsetzt, ein API Gateway mit Hilfe von Ocelot verwendet, eine RabbitMQ zur Verfügung stellt und die unterschiedlichen Services miteinander in einem Netzwerk verknüpft und deployed.

5.1 Implementieren der Authentifizierung und Autorisierung

Ein zentraler Teil bei Microservices ist die Authentifizierung sowie die Autorisierung. Es wurde sich für eine Identity Server mit OAuth2 entschieden. Die MySQL-Datenstruktur ergibt sich aus dem Datenmodell:

```
public class User
{
    [Required] public int Id { set; get; }
    public string Name { set; get; }
    public string Password { set; get; }
    public bool Active { set; get; }
    public string Role { set; get; }
}
```

Als eindeutiger Identifier ist eine ‘Id‘ notwendig. Zusätzlich besitzt jeder User einen Namen, ein Passwort, ist im Standardfall aktiviert und es wird zwischen zwei Rollen unterschieden: ‘customer‘ und ‘admin‘. Wie zu erwarten,. hat die ‘admin‘-Rolle mehr Berechtigung als ein ‘customer‘.

Um den Identity Server einzurichten, müssen sogenannte NuGet-Pakete heruntergeladen werden. NuGet ist ein System, mit welchem Softwarebibliotheken bereitgestellt werden können. Diese Verweise werden zur Projektdatei hinzugefügt.

```
<PackageReference Include="IdentityServer4" Version="2.3.0" />
<PackageReference Include="IdentityServer4.AccessTokenValidation" Version="2.7.0" />
```

In der ‘Startup.cs‘, welche standardmäßig bei C# .NET Projekten vorhanden ist, werden grundsätzlich Serverkonfigurationen initialisiert. Dies muss für den Identity Server ebenfalls durchgeführt werden.

```
(1)
var builder = services.AddIdentityServer()
.AddInMemoryIdentityResources(Config.GetIdentityResources())
.AddInMemoryApiResources(Config.GetApis())
.AddInMemoryClients(Config.GetClients())
.AddProfileService<ProfileService>();
```

```
services.AddTransient<IResourceOwnerPasswordValidator, ResourceOwnerPasswordValidator
services.AddTransient<IProfileService, ProfileService>();

(2)
services.AddAuthentication("Bearer")
.AddJwtBearer("Bearer", options =>
{
   options.Authority = "http://identity_server_service";
   [...]
   options.Audience = "srapi";
});
```

Wie man feststellen kann, sind diese Konfigurationen schon recht umfangreich, obwohl es sich in dieser Version schon um eine sehr leichtgewichtige Implementierung von OAuth2 handelt. In (1) wird der Identity Server zum Projekt hinzugefügt. Zusätzlich wird auf den ‘ProfileService‘ sowie den ‘ResourceOwnerPasswordValidator‘ verwiesen, welche anschließend initialisiert werden müssen. In (2) wird noch die ‘Authority‘ und die ‘Audience‘ gesetzt. Die ‘Authority‘ garantiert zusätzlich, dass das Token nicht von einem anderen Identity Server ausgestellt wird. Die ‘Audience‘ gibt noch einmal an, dass das ausgestellte Token nur Zugriff auf eine entsprechende ‘srapi‘ (Stirnraten-API) Ressource hat.

```
new ApiResource("srapi", "Stirnraten API")
{
    ApiSecrets = new List<Secret>()
    {
    new Secret(CustomClientSecret.Sha256())
    }
}
```

Zusätzlich müssen noch die beiden in den Grundlagen erwähnten Flows (‘ClientCredentials‘ und ‘ResourceOwnerPassword‘) implementiert werden.

```
{
[...]
new Client
{
    ClientId = CustomClientId,
    AllowedGrantTypes = GrantTypes.ClientCredentials,
    ClientSecrets =
    {
    new Secret(CustomClientSecret.Sha256())
    },
    AllowedScopes =
```

```
    {
    "srapi"
    }
},

new Client
{
    ClientId = "sr.client",
    AllowedGrantTypes = GrantTypes.ResourceOwnerPassword,
    ClientSecrets =
    {
    new Secret("secret".Sha256())
    },
    AllowedScopes =
    {
    "srapi"
    }
}
```

Durch diese Flows ist garantiert, dass die API grundsätzlich geschützt ist, d.h. dass nur die Stirnraten Apps auf basale Funktionen der API zugreifen dürfen. Wenn Benutzer in den Apps Benutzerdaten hinterlegt haben, erhalten diese logischerweise noch weitere Zugriffsrechte. Ein Konto mit der Rolle ‘admin‘ hat dementsprechend noch mehr Rechte. Dies wird im Abschnitt 5.2 deutlich.

Wenn eine Anfrage mit Benutzernamen und Passwort gestellt wird, ruft der Identity Server in seinem Abarbeitungszyklus folgende Methode ab:

```
public async Task ValidateAsync(ResourceOwnerPasswordValidationContext context)
{
(1)
var user = await _unitOfWork.UserRepository.
           GetUserByNameAndPasswordAsync(context.UserName, context.Password);

(2)
if (user == null)
{
context.Result = new GrantValidationResult(
TokenRequestErrors.InvalidGrant,
"invalid custom credential");
return;
}
```

```
if (!user.Active)
{
context.Result = new GrantValidationResult(
TokenRequestErrors.InvalidClient,
"User was deactivated by admin");
return;
}

(3)
context.Result = new GrantValidationResult(
subject: user.Id.ToString(),
authenticationMethod: "custom",
claims: GetUserClaims(user)); //get user claims
}
```

In (1) wird ein Benutzer gesucht, welcher mit dem übergebenen Namen und Passwort übereinstimmt. In (2) wird dieser validiert und in (3) das Token anhand der ‘Claims‘ generiert. ‘Claims‘ sind vereinfacht ausgedrückt, zusätzliche Informationen, welche man in dem Token übergeben möchte.

```
return new[]
{
new Claim("user_id", user.Id.ToString() ?? ""),
new Claim(JwtClaimTypes.Name, user.Name),
new Claim(JwtClaimTypes.Role, user.Role)
};
```

In der umgesetzten API wird eine ‘user_id‘, der ‘Name‘ sowie die ‘Role‘ übergeben. Alles wird ggf. von anderen Microservices benötigt, um eine entsprechende Autorisierung zu gewährleisten.

Abbildung 15 und 16 zeigen, wie mit dem Programm Postman über eine POST Abfrage die entsprechenden Token ausgestellt werden können.

Enkodiert sieht so ein ausgestellter Benutzer-Token wie folgt aus:

```
[...]
"iss": "http://identity_server_service",
"aud": [
"http://identity_server_service/resources",
"srapi"
],
"client_id": "sr.client",
```

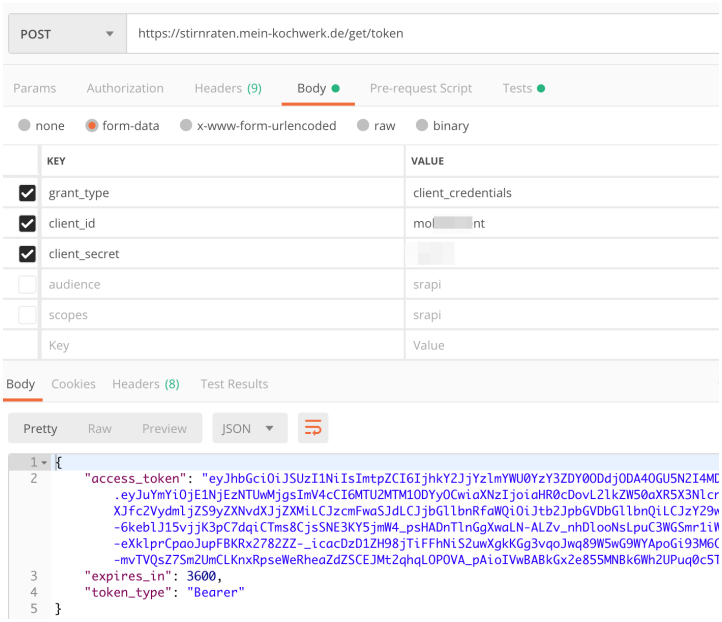

Abbildung 15: Über Postman gesendeter Client Credential Flow. Mit Hilfe dieses Tokens erhält man für 3600 Sekunden (eine Stunde) beschränkten Zugang zur API, um z.B. Kategorien abzurufen, neue Wörter einzusenden oder ein Benutzerkonto anzulegen.

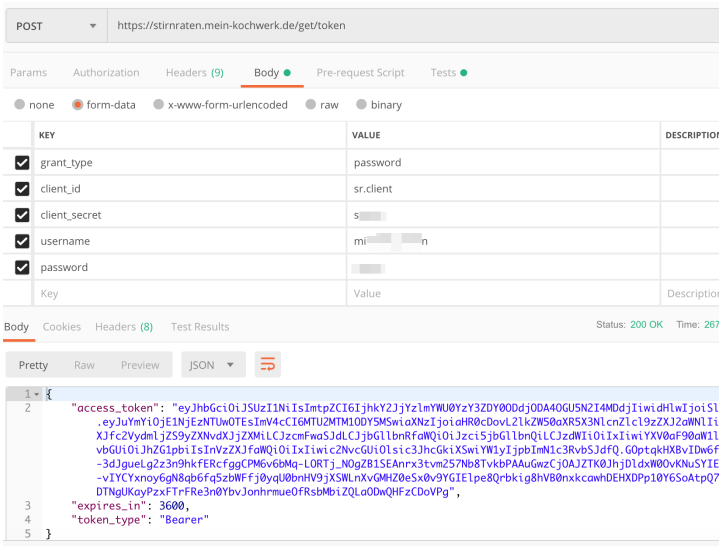

Abbildung 16: Über Postman gesendeter Password Flow. Er ist ebenfalls 3600 Sekunden lang gültig und hat mehr Berechtigung, wie z.B. das Speichern eines Benutzerprofils.

```
[...]
"role": "admin",
"user_id": "1",
```

```
"scope": [
"srapi"
],
"amr": [
"custom"
]
[...]
```

Durch diese Informationen kann das API Gateway bereits eine Autorisierung vornehmen. Wie dies im Detail funktioniert, wird im Abschnitt 5.2 deutlich.

5.2 Implementierung des API Gateway mit Ocelot

Ähnlich wie bei dem Identity Server müssen für Ocelot auch gewisse Bibliotheken über NuGet hinterlegt werden.

```
 <PackageReference Include="IdentityServer4.AccessTokenValidation"
 Version="3.0.0-preview.2" />
<PackageReference Include="IdentityServer4" Version="2.4.0" />
<PackageReference Include="Ocelot" Version="11.0.2" />
<PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" />
```

Durch diese Pakete wird zum einen Ocelot eingebunden, zum anderen wird der Authentifizierungsmechanismus zum Identity Server vereinfacht. Des Weiteren müssen Verbindungsdaten hinterlegt werden, die garantieren, dass das Gateway den Identity Server erreichen kann. Diese werden wieder in der ‘Startup.cs‘ hinterlegt.

```
(1)
var authenticationProviderKey = "NTT9N7MXLJN9";

(2)
void Options(IdentityServerAuthenticationOptions o)
{
   o.Authority = "http://identity_server_service";
   o.ApiName = "srapi";
   o.RequireHttpsMetadata = false;
   o.SupportedTokens = SupportedTokens.Both;
   o.ApiSecret = "xxxxxx";
}
services.AddAuthentication()
.AddIdentityServerAuthentication(authenticationProviderKey, Options);

[...]
(3)
await app.UseOcelot();
[...]
```

In (1) muss ein sogenannter ‘Authentication Provider Key‘ gesetzt werden. Dies ist obligatorisch vorgegeben von Ocelot und notwendig, um weitergeleitete Requests eindeutig zuzuordnen. In (2) werden die Optionen für die Verbindung zum Identity Server definiert. Diese Konfigurationen müssen mit den Einstellungen aus dem zuvor erwähnten Identity Server übereinstimmen. In (3) wird noch einmal explizit ausgedrückt, dass Ocelot auch verwendet werden soll.

Um eingehende Request zu verarbeiten, verwendet Ocelot eine JSON, in der sogenannte ReRoutes (Weiterleitungen) hinterlegt werden.

```
"ReRoutes": [
 {
 (1)
 "DownstreamPathTemplate": "/api/customers",
 (2)
 "DownstreamScheme": "http",
 "DownstreamHostAndPorts": [
    {
    (3)
    "Host": "customer_service",
    "Port": 80
    }
  ],
  (4)
  "UpstreamPathTemplate": "/api/stats",
  (5)
  "AuthenticationOptions": {
  "AuthenticationProviderKey": "NTT9N7MXLJN9",
  "AllowedScopes": [
    "srapi"
  ]
  },
  (6)
  "RouteClaimsRequirement": {
  "role": "admin"
  }
 },
[...]
]
```

Diese dargestellte ReRoute zeigt bereits viele Vorteile von Ocelot. Durch (1) wird das ‘DownstreamPathTemplate‘ definiert. D.h. das Gateway präsentiert über das ‘UpstreamPathTemplate‘ (5) die öffentlichen Schnittstellen, welche man als Client anspricht. Dadurch lässt sich das Weiterleiten flexibel gestalten. Zusätzlich ist es nicht zwingend

notwendig, dass wenn das ‘DownstreamPathTemplate‘ seine Route ändert, die Clienten davon betroffen sind, da das ‘UpstreamPathTemplate‘ gleichgeblieben ist.

Das ‘DownstreamScheme‘ (2) ist http. Https ist nicht nötig, da die Microservices sich nur in einem lokalen Netz befinden. Das hat den Vorteil, dass die Microservices nur über das Gateway zu erreichen sind. So wird garantiert, dass kein Dritter Zugriff auf die Services hat, ohne über das Gateway zu gehen.

In (3) wird definiert, an welchen Microservice weitergeleitet werden soll. In diesem Beispiel der ‘customer_service‘.

In (5) findet die Authentifizierung statt: Jeder Request, welcher eingeht, muss im Token als Scope die ‘srapi‘ enthalten. In diesem Beispiel wird bereits auch autorisiert. Denn es wird die Rolle ‘admin‘ erwartet (6). D.h. wenn ein Request kein gültiges Token oder die entsprechende Rolle hat, wird der Request abgelehnt.

Ocelot bietet zusätzlich an, Informationen aus dem Token direkt in den Request-Header zu parsen. Das kann sinnvoll sein, wenn der verarbeitende Service mit Daten aus dem Token arbeiten muss. Zum Beispiel darf ein Benutzer nur seine eigene Statistik sehen und verändern oder wenn ein neuer Benutzer angelegt wird, darf dieser nur die Rolle ‘admin‘ von einem anderen ‘admin‘ erhalten.

```
[...]
(7)
"AddHeadersToRequest": {
 "claims_user_id": "Claims[user_id] > value > |",
 "claims_role": "Claims[role] > value > |"
},
(8)
"UpstreamHttpMethod": [
 "Get",
 "Delete",
 "Patch"
]
[...]
```

Ebenfalls lassen sich die HTTP Methoden über die ‘UpstreamHttpMethod‘ einschränken (8). Dies bietet zusätzliche Sicherheit und Einschränkungen.

Zusammenfassend lässt sich sagen, dass Ocelot sehr mächtig über die JSON konfigurierbar ist. Ocelot bietet noch weitere Funktionen an, die allerdings nicht in der Stirnraten API verwendet worden sind.

5.3 Aufsetzen einer RabbitMQ

Um wie geplant eine asynchrone Kommunikation zu erreichen, wird auf eine RabbitMQ zurückgegriffen. Diese läuft als eigenständiger Dienst auf dem Server. Um dies ohne viel Aufwand zu erreichen, wird die RabbitMQ als Container gestartet. Die RabbitMQ stellt eine Oberfläche bereit. Dies vereinfacht die Handhabung, ist aus technischer Sicht allerdings nicht notwendig. Wie die Containerisierung im Detail verwendet wird, ist in dem Abschnitt 'Bauen und Hochladen der Microservices' nachzulesen.

Wenn die RabbitMQ gestartet ist, können die Services sich mit dieser verbinden. Dafür benötigen sie folgende Bibliotheken:

```
<PackageReference Include="MassTransit" Version="5.5.1" />
<PackageReference Include="MassTransit.Autofac" Version="5.5.1" />
<PackageReference Include="MassTransit.RabbitMQ" Version="5.5.1" />
```

Es muss ein sogenannter 'Contract' erzeugt werden, dieser ist im Prinzip die Nachricht, die übertragen werden soll. In diesem Fall werden die Daten ('UserId' und 'Name') beim Erzeugen eines neuen Benutzers erstellt und durch die RabbitMQ übertragen.

```
public class UserCreated
{
public string UserId { get; set; }
public string Name { get; set; }
}
```

Anschließend muss die 'Startup.cs' mit folgenden Informationen angereichert werden:

```
private void ConfigureBus(ContainerBuilder builder)
 {
 builder.Register(context =>
 {
  return Bus.Factory.CreateUsingRabbitMq(config =>
  {
   (1)
   var host = config.Host(new Uri("rabbitmq://" + _rabbitMqPath), h =>
   {
   (2)
   h.Username("xxxxx");
   h.Password("xxxxx");
   });
  });
 }).As<IBus, IBusControl, IPublishEndpoint>().SingleInstance();
}
```

Jeder Service, der mit der RabbitMQ verbunden sein möchte, benötigt diese Konfiguration. Über die URI wird dem Service mitgeteilt, über welche Adresse er sich mit der RabbitMQ verbinden kann. Anschließend werden Benutzernamen und Passwort benötigt (2). Daraufhin muss die RabbitMQ gestartet werden, dies geschieht ebenfalls noch in der 'Startup.cs'.

```
busControl.Start();
```

Wenn ein neuer Benutzer erstellt wird, wird folgendes Kommando abgesetzt.

```
await _busControl.Publish(newUserCreated);
```

Alle Services, die sich auf diese Nachricht abonniert haben, erhalten nun die Informationen. Dies impliziert das 'IConsumer' Interface.

```
public class UserCreatedConsumer : IConsumer<UserCreated>
{
 public Task Consume(ConsumeContext<UserCreated> context)
 {
 (1)
 var user = context.Message;
 return TaskUtil.Completed;
 }
}
```

Bei (1) liegt der Benutzer vor, so dass der Service die Daten verarbeiten kann, wie er es benötigt.

5.4 Bauen und Hochladen der Microservices

Die Architektur für den Livebetrieb hat sich ein wenig erweitert, da ein Proxy (siehe Abbildung 17 'nginx') hinzugekommen ist. Diese Möglichkeit wird in der Regel von Hostingdiensten angeboten oder kann z.B. mittels Docker selbst erstellt werden. Es wird nur als eine Weiterleitung genutzt, hat aber z.B. den Vorteil, dass Nutzer von außen nicht sehen, welche Technologien oder Systeme für die Stirnraten API verwendet werden. Dies schützt vor möglichen Angreifern.

Die Einstellung für den 'nginx' beinhalten gerade einmal folgende Zeilen:

```
<Location "/>
ProxyPass "http://127.0.0.1:6200/"
ProxyPassReverse "http://127.0.0.1:6200/"
</Location>
```

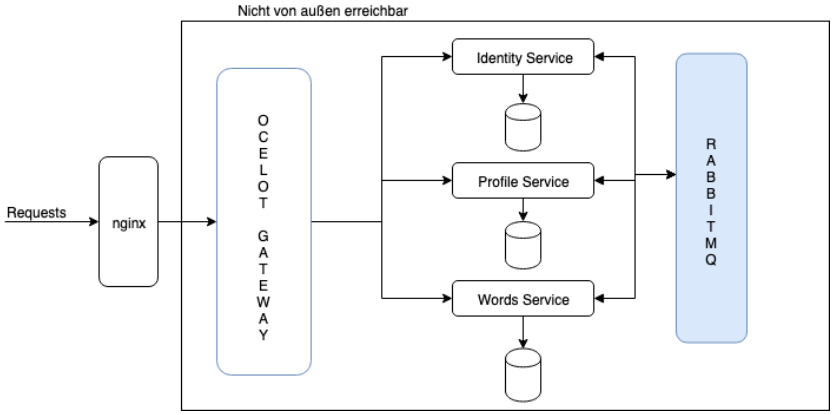

Abbildung 17: Abbildung der Live-Architektur von der Stirnraten API

Sobald ein Request eingeht, wird dieser auf an den Port 6200 weitergeleitet, wo der Ocelot Service läuft. Die Abbildung 17 zeigt insgesamt 8 Microservices. Das umfasst drei MySQL-Datenbanken, drei Services mit Geschäftslogik (Words, Customer, Identity Server), das Gateway und die RabbitMQ. Jeder dieser Services muss auf seinem eigenen Port laufen, um erreichbar zu sein und muss eigenständig deployed werden können.

Im Konzept wurde sich darauf festgelegt, dass die Microservices eigene Container sind. D.h. immer wenn ein Service gebaut wird, wird ein Docker-Image erzeugt. Das Bauen des Images wird über die Dockerfile beschrieben:

```
(1)
FROM microsoft/dotnet:2.2-sdk
WORKDIR /app

(2)
# copy csproj and restore as distinct layers
COPY *.csproj ./
RUN dotnet restore
COPY . ./
RUN dotnet publish -c Release -o out
ENTRYPOINT ["dotnet", "out/GatewayService.dll"]
```

In (1) wird beschrieben, welches .NET notwendig ist, in (2) wird in das Image eine ausführbare ‘dll‘ generiert. Dieses Image muss in eine Docker Registry. Für Unternehmen bietet es sich an, eine eigene Registry zu betreiben, der Einfachheit halber, wurde für Stirnraten ein öffentlicher Dienst namens Dockerhub verwendet. Das Build-Script sieht exemplarisch wie folgt aus:

```
docker build --tag kuzdu/stirnraten:gateway_service_live ../GatewayService
docker push kuzdu/stirnraten:gateway_service_live
```

Über die Konsole mittels ‘build‘ wird ein Image erstellt und mit ‘gateway_service_live‘ getagged. Tags sind zum Identifizieren notwendig. Anschließend wird es nach Dockerhub übertragen. Hinweis: Um mit dem eigenen Dockerhub Account zu kommunizieren, muss man sich einmalig pro Gerät über die Konsole authentifizieren.

Wenn das Image nun in der Registry bereitliegt, muss der Live-Server es nur noch ausführen. Dieses Script sieht wie folgt aus:

```
#!/bin/bash
(1)
ssh root@mein-kochwerk.de << EOF
(2)
docker stop stirnraten_gateway_service_live
docker rm stirnraten_gateway_service_live

(3)
docker run --restart unless-stopped --name stirnraten_gateway_service_live -p 127.0.0
exit
EOF #end of file
```

Mittels ‘ssh‘ wird sich mit dem Server (hier mein-kochwerk.de heißend) verbunden (1). Es wird überprüft, ob bereits ein Microservice läuft, wenn ja, wird dieser gestoppt und anschließend entfernt (2). In (3) wird sich über ‘kuzdu/stirnraten:gateway_service_live‘ das entsprechende Image aus dem Dockerhub-Repository geladen und ausgeführt. Beim Ausführen vergeben wir einen Namen, lassen das Gateway auf dem Port 6200 laufen und teilen mit, dass die Anwendung sich neustarten soll, falls sie abstürzen sollte. Sehr wichtig ist auch der ‘network‘ Parameter. Dieser fasst die Microservices in einem Netzwerk zusammen. Dies bedeutet, dass die Services innerhalb dieses Netzwerks problemlos kommunzieren können. Falls auf dem Server noch andere Container laufen, die gar nicht zur Anwendung gehören, sehen diese Container sich nicht.

Um eine bessere Übersicht zu behalten, wurde auf dem Server ein Dienst namens ‘Portainer‘ installiert. Dieser bietet zur Docker-Konsole eine grafische Oberfläche. In der Abbildung 18 sieht man alle Stirnraten-Container.

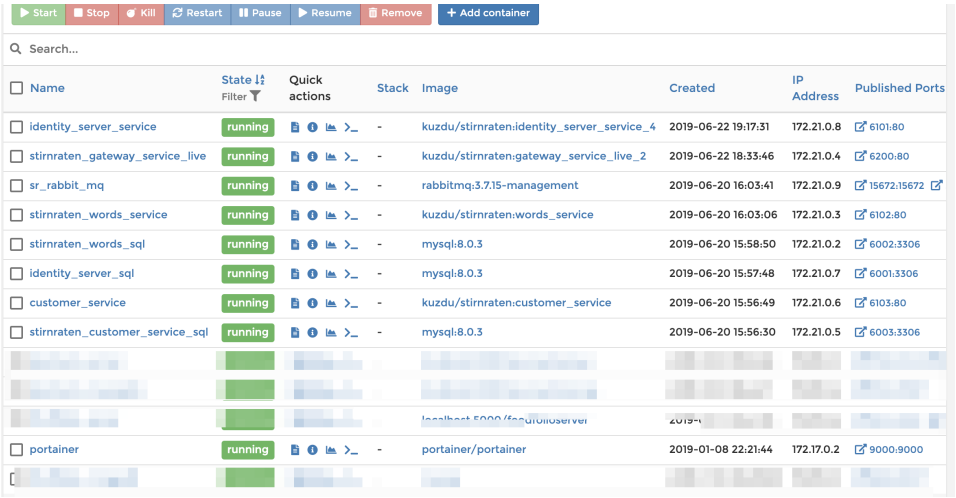

Abbildung 18: Alle laufenden Microservices

6 Fazit

Das Ziel, den Aufbau einer API basierend auf der Microservice Architektur anhand des Spieles Stirnraten, betrachte ich meiner Meinung nach als gelungen und abgeschlossen. Mittels Domain Driven Design konnten verschiedene Microservices detektiert und abgegrenzt werden. Unterstüzend diente User Story Mapping als Werkzeug zum Erfassen von Anforderungen. Diese wurden nach Priorität sortiert und es ist ein Produkt entstanden, welches aktiv eingesetzt werden kann. Ebenfalls werden weitere Ausbaustufen aufgezeigt, da selbstverständlich nicht alle erfassten Anforderungen aus Kapazitätsgründen umgesetzt wurden.

Weiter wurde eine Microservice Architektur erschaffen, die den definierten Bewertungskriterien für Microservices entspricht. Zusätzlich wurden empfohlene Pattern wie das Verwenden eines Gateways und das Nutzen der asynchronen Kommunikation angewandt. Die Authentifizierung sowie Autorisierung bietet ein hohes Maß an Flexiblität. Dies bedeutet, dass weitere Microservices, aber auch etwaige Drittnutzer, einen kontrollierten Zugriff auf die API erhalten könnten.

Trotz der erfolgreichen Umsetzung, gilt darauf hinzuweisen, dass der Umfang der Anforderungen an dieses Projekt derzeit keine Microservice Architektur rechtfertigen würde und deshalb eher prototypisch zu betrachten ist. Durch die vielen verschiedenen Microservices ist Mehrarbeit entstanden. Beispielsweise hätte die Datenstruktur mühelos auf eine Datenbank abgebildet werden können, stattdessen wurden drei Datenbanken verwendet.

6.1 Ausblick

Diese Projektarbeit bietet noch zahlreiche, technische Erweiterungen an. Die Portvergabe und das Zusammenspiel der Microservices war aufwändig und teilweise sehr fehleranfällig, da es manuell und per Hand betrieben worden ist. Wenn die Serviceanzahl wächst, ist dies tendenziell eine große Fehlerquelle, da es ein hoher Arbeitsaufwand ist, nicht den Überblick zu verlieren. Dafür gibt es bereits mögliche Lösungen. Zum einen wäre Kubernetes als Orchestrierungsprogramm möglich. Zum anderen könnte ein sogenannter Service-Mesh erforscht werden, bei dem die Microservices sehr vereinfacht ausgedrückt, sich selbst ins System einpflegen.

Zusätzlich sollte eine automatisierte Build- und Deploymentstruktur (CI/CD) erschaffen werden. Open Source Tools wie z.B. Jenkins oder kostenpflichtige Tools wie z.B. Bamboo bieten zahlreiche Automatisierungsoptionen. Automatisierte Build- und Deploymentscripts sparen viel Zeit und Aufwand.

Wenn die Last wächst, sollten Microservices skalierbar sein. Das Verwenden von mehreren Servern und einem Loadbalancer, der die Last entsprechend verteilt, wären mögliche Lösungsansätze. Natürlich müssen dabei zusätzliche Aspekte beachtet werden, wie z.B. dass keine Inkonsistenzen in den Datenbanken entstehen.

Der letzte und möglicherweise wichtigste Punkt ist, die Microservices zu überwachen. Dies umfasst, entstandene Errors zu erkennen (Logging). Dafür gibt es bereits etablierte Tools wie z.B. den ELK Stack (Elasticsearch, Logstash, Kibana). Ebenfalls sollten Strategien überlegt werden, was passiert, wenn ein Service ausfällt und wie man diesen Zustand bereits frühzeitig bemerkt (Health Check).

Natürlich sind die genannten Ausblicke auch für eine monolithische Anwendung wichtig, allerdings bewerte ich die Umsetzung der genannten Punkte in einer Microservice Architektur als komplexer und umfangreicher. Umso größer die Anwendung wird, desto eher sollte man eine Microservice Architektur verwenden.

Tabellenverzeichnis

Abbildungsverzeichnis

Literatur

[1] Mark Fussel (msfussell). *Einführung in Microservices in Azure - Microsoft-Dokumentation*. abgerufen am 9.4.2019. 2017. URL: `https://docs.microsoft.com/de-de/azure/service-fabric/service-fabric-overview-microservices`.

[2] Amine El Ayadi. *Webservices: REST vs. SOAP*. Hochschule für Angewandte Wissenschaften Hamburg. 2008.

[3] Ferdinand Birk. „Microservices - Eine State-of-the-Art Bestandsaufnahme und Abgrenzung zu SOA“. Universität Ulm, 2016.

[4] Björn Bohn. *Podman*. abgerufen am 27.5.2019. 2019. URL: `https://www.heise.de/developer/meldung/Die-Docker-Alternative-Podman-erreicht-Version-1-0-4281333.html`.

[5] Cesar. *Designing and implementing API Gateways with Ocelot in .NET Core containers and microservices architectures*. abgerufen am 29.5.2019. 2018. URL: https://devblogs.microsoft.com/cesardelatorre/designing-and-implementing-api-gateways-with-ocelot-in-a-microservices-and-container-based-architecture/.

[6] Antoine Duprat. *How to choose a Message Queue*. abgerufen am 2.6.2019. 2018. URL: https://medium.com/linagora-engineering/how-to-choose-a-message-queue-247dde46e66c.

[7] Sebastian Gauder. „A competitive food retail architecture with microservices“. 2019.

[8] PIETER HUMPHREY. *Understanding When to use RabbitMQ or Apache Kafka*. abgerufen am 2.6.2019. 2017. URL: https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka.

[9] Matt McLarty Irakli Nadareishvili Ronnie Mitra und Mike Amundsen. *Microservice Architecture - Aligning Principels, Practices, and Culture*. O Reilly, 2016.

[10] solid IT gmbh. *DB-Engines Ranking*. abgerufen am 21.5.2019. 2019. URL: https://db-engines.com/de/ranking.

[11] Kevin Klöckner. *Im Vergleich: NoSQL vs. relationale Datenbanken*. Universität Siegen. 2015.

[12] Paulo A. Pereira Morgan Bruce. *Microservices in Action*. Manning, 2019.

[13] Sven Neuhaus. *Betont schlank*. abgerufen am 21.5.2019. 2016. URL: https://www.heise.de/ix/artikel/Betont-schlank-506574.html.

[14] Dennis Peuser Nhiem Lu Gert Glatz. *Moving mountains – practical approaches for moving monolithic applications to Microservices*. University of Applied Science und Arts, Dortmund, Germany, adesso AG, Dortmund, Germany. 2017.

[15] olprod hongyes olprod. *OWIN-OAuth 2.0-Autorisierungsserver*. abgerufen am 2.6.2019. 2019. URL: https://docs.microsoft.com/de-de/aspnet/aspnet/overview/owin-and-katana/owin-oauth-20-authorization-server.

[16] Aaron Parecki. *OAuth 2.0 Client Credentials Grant*. abgerufen am 11.5.2019. o. J. URL: https://oauth.net/2/grant-types/client-credentials/.

[17] Jeff Patton. *User Story Mapping - Die Technik für besseres Nutzerverständnis in der agilen Produktentwicklung*. O Reilly, 2015.

[18] phlegx. *Create multiple users and databases*. abgerufen am 21.5.2019. 2016. URL: https://github.com/docker-library/postgres/issues/151.

[19] Chris Richardson. *Microservice Pattern*. Manning, 2019.

[20] Golo Roden. *Domain-driven Design erklärt*. abgerufen am 15.3.2019. 2016. URL: https://www.heise.de/developer/artikel/Domain-driven-Design-erklaert-3130720.html?seite=all.

[21] Heiko Rupp. *Das Service Mesh für verteilte Systeme*. abgerufen am 29.5.2019. 2018. URL: https://www.heise.de/developer/artikel/Istio-Das-Service-Mesh-fuer-verteilte-Systeme-4153426.html.

[22] Cesar de la Torre (cesardelatorre). *Implementieren ereignisbasierter Kommunikation zwischen Microservices*. abgerufen am 16.4.2019. 2018. URL: https://docs.microsoft.com/de-de/dotnet/standard/microservices-architecture/multi-container-microservice-net-applications/integration-event-based-microservice-communications.

[23] o. V. *App-Container: 5 professionelle Docker-Alternativen im Überblick*. abgerufen am 27.5.2019. 2017. URL: https://t3n.de/news/docker-alternativen-container-783741/.

[24] o. V. *AWS SDK für .NET*. abgerufen am 28.5.2019. o. J. URL: https://aws.amazon.com/de/sdk-for-net/.

[25] o. V. *Erste Schritte mit Endpoints in einer flexiblen App Engine-Umgebung (.NET)*. abgerufen am 28.5.2019. o. J. URL: https://cloud.google.com/endpoints/docs/openapi/get-started-app-engine-dotnet.

[26] o. V. *Istio Quickstart*. abgerufen am 29.5.2019. o. J. URL: https://istio.io/docs/setup/kubernetes/install/kubernetes/.

[27] o. V. *Kafka - A distributed stream platform*. abgerufen am 2.6.2019. o. J. URL: https://kafka.apache.org/uses.

[28] o. V. *Plan an API Gateway system*. abgerufen am 1.5.2019. o. J. URL: https://docs.oracle.com/cd/E55956_01/doc.11123/administrator_guide/content/admin_planning.html.

[29] o. V. *Was ist Docker?* abgerufen am 11.5.2019. o. J. URL: https://www.redhat.com/de/topics/containers/what-is-docker.

[30] Eberhard Wolff. *Das Microservice Praxisbuch - Grundlagen, Konzepte und Rezepte*. dpunkt.verlag, 2018.

[31] Eberhard Wolff. *Flexible Software Architectures*. Leanpub, 2016.