HAW HAMBURG

BACHELORTHESIS
Isabelle Maria Dakowitz

Die Kombination von Domain-Driven Design und Function as a Service

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Isabelle Maria Dakowitz

Die Kombination von Domain-Driven Design und Function as a Service

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

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

Eingereicht am: 11. Juli 2020

Isabelle Maria Dakowitz

Thema der Arbeit

Die Kombination von Domain-Driven Design und Function as a Service

Stichworte

Domain-Driven Design, Function as a Service, Fission Framework, Java, Kubernetes, NATS Streaming

Kurzzusammenfassung

In dieser Arbeit wird die Kombination von Domain-Driven Design und Function as a Serivce untersucht und dessen Kompatibilität bewertet. Die Bewertung erfolgt anhand einer Fallstudie auf Grundlage einer Analyse, eines Entwurfs der Softwarearchitektur und einer Implementierung mithilfe des Frameworks Fission. Dabei haben sich insbesondere die Anwendung einer Message Queue und Message Queue Triggern als erfolgreicher Lösungsansatz für die Kombination erwiesen.

Isabelle Maria Dakowitz

Title of Thesis

The combination of Domain-Driven Design and Function as a Service

Keywords

Domain-Driven Design, Function as a Service, Fission Framework, Java, Kubernetes, NATS Streaming

Abstract

In this work the combination of Domain-Driven Design and Function as a Service is examined and evaluated for their compatibility. The assessment is based on a case study, containing an analysis, a software architecture design and an implementation using the framework Fission. In particular, the use of a message queue and message queue triggers have proven to be a successful solution for the combination.

Inhaltsverzeichnis

Abbildungsverzeichnis **vi**

Abkürzungen **viii**

Listings **x**

1 Einleitung **1**

2 Grundlagen **3**
2.1 Domain-Driven Design 3
2.1.1 Strategisches Domain-Driven Design 4
2.1.2 Taktisches Domain-Driven Design 6
2.1.3 Domain Events 7
2.2 Function as a Service 8
2.2.1 Aufbau von Functions 9
2.2.2 Anbieter 10
2.2.3 Vor- und Nachteile 10
2.2.4 Anwendungsfälle 11

3 Anwendungsfall und Analyse des Zusammenspiels von DDD mit FaaS **12**
3.1 Anwendungsfall 12
3.2 Analyse 12
3.2.1 Analyse mit Fokus auf DDD 16

4 Entwurf **20**
4.1 Schichtenmodell 20
4.2 Architektursichten 23
4.2.1 Laufzeitsicht 24
4.2.2 Verteilungssicht 28

5 Implementierung **30**
5.1 Voraussetzungen für die Implementierung 30
5.1.1 Wahl der Programmiersprache 30
5.1.2 Wahl des Frameworks 31
5.1.3 Funktionsweise von Fission 34
5.2 Implementierung des Fallbeispiels mit Fission 36
5.2.1 Domain Events mit NATS Streaming 38

6 Evaluierung und Bewertung **48**
6.1 Bewertung und Evaluierung des Entwurfs und der Implementierung 48
6.2 Bewertung und Evaluierung des gewählten Frameworks für DDD in Kombination mit FaaS 50

7 Fazit und Ausblick **52**
7.1 Quintessenz der Leitfrage 52
7.2 Weitere Forschungsmöglichkeiten 53

Literaturverzeichnis **54**

Selbstständigkeitserklärung **59**

Abbildungsverzeichnis

2.1 Domain Event mit Eventual Consistency [28]. 8

3.1 Anwendungsfalldiagramm für den Bestellprozess bei einem Restaurant. . . 14
3.2 Fachliches Datenmodell für das FoodDeliverySystem. 15
3.3 Context Map der Domain FoodDeliverySystem. 19

4.1 Drei-Schichtenarchitektur. 21
4.2 Beispiel einer Architektur unter Anwendung von Function as a Service (FaaS). 22
4.3 Sequenzdiagramm mit einer Function, welche in zwei Bounded Contexts agiert i.A.a. [15], *(orange: CustomerContext, rot: Trigger, grün: Function, blau: RestaurantContext)*. 26
4.4 Sequenzdiagramm mit zwei Functions unter Anwendung einer Message Queue i.A.a. [15] *(orange: CustomerContext, rot: Message Queue (MQ) und Trigger, grün: Functions, blau: RestaurantContext)*. 27
4.5 Das FoodDeliverySystem mit Nachbarsystem und FaaS-Umgebung. 28

5.1 Poolmanager für das Java Environment i.A.a. [23]. 32
5.2 Pool von Pods für das Java Environment in Fission i.A.a. [23]. 32
5.3 Test-Functions beim Aufrufen der Deployments für Kubeless i.A.a. [23]. . 33
5.4 Ein Pod pro Function bei Kubeless i.A.a. [23]. 33
5.5 Fission Konzepte [17]. 35
5.6 Bei wiederholtem Aufrufen einer Function wird Spring Boot nicht erneut gestartet. 35
5.7 Das CustomerMealOrder-Aggregate mit seinen Bestandteilen. 37
5.8 Der Client sendet ein Event an die NATS MQ und mittels des MQ-Triggers wird die Ausführung der angesprochenen Function bewirkt [21]. 38
5.9 Informationen (Ausschnitt) über den Pod, der über den `nats-streaming` Service erreichbar ist. 40

5.10 Angepasstes Sequenzdiagramm aus Abschnitt 4.2.1 mit HTTP-Triggern und MQ-Triggern für Domain Events *(orange: CustomerContext, rot: Trigger und MQ, grün: Functions, blau: RestaurantContext)*. 41
5.11 In den Logmeldungen ist zu erkennen, dass die Kommunikation zwischen den Bounded Contexts über die MQ gelungen ist. 47

6.1 `ClassNotFoundException` beim Ausführen einer Function am Beispiel der Anwendung des `JpaRepository`. 51

Abkürzungen

API Application Programming Interface.

AWS Amazon Web Services.

BaaS Backend as a Service.

CNCF Cloud Native Computing Foundation.

CQRS Command Query Responsibility Segregation.

DDD Domain-Driven Design.

DNS Domain Name System.

FaaS Function as a Service.

GUI Graphical User Interface.

HTTP Hypertext Transfer Protocol.

IBM International Business Machines Corporation.

JSON JavaScript Object Notation.

MQ Message Queue.

NATS Neural Autonomic Transport System.

SoC Separation of Concerns.

UML Unified Modeling Language.

URL Uniform Resource Locator.

Listings

5.1 Maven Dependency für NATS Streaming. 39
5.2 Methode zum Veröffentlichen eines Events unter Anwendung einer NATS `StreamingConnection` und unter Angabe eines topics. 39
5.3 `CreateCustomerOrder` Klasse als Entrypoint für die `customer-order` Function. 43
5.4 Unbekannte Properties bei Verwenden des `ObjectMapper` ermöglichen. . . 45
5.5 Befehl für das Erstellen der `customer-order` Function für den Customer-Context. 45
5.6 Befehl zur Erstellung eines HTTP-Triggers mit Fission. 46
5.7 Befehl zur Erstellung eines MQ-Triggers mit Fission. 46

1 Einleitung

Angesichts der über Jahre andauernden steigenden Relevanz und Komplexität von Softwaresystemen nehmen ebenfalls Methoden zur Optimierung und Unterstützung des Softwareentwicklungsprozesses sowie der Softwarequalität an Bedeutung zu [12, S. 747 ff.], [49, S. 124].
Eine Herangehensweise für die Entwicklung eines komplexen Softwaresystems bietet unter anderem *Domain-Driven Design (DDD)*. Dieses dient einer verbesserten Kommunikation zwischen Fachexperten und Softwareentwicklern. DDD bildet schon seit geraumer Zeit einen Bestandteil innerhalb von Softwareprojekten und unterstützt die am Projekt beteiligten Akteure sowohl auf eine strategische als auch eine taktische Art und Weise [34, S. 3, S. 11 ff.].
Daneben findet bei der Entwicklung von Softwaresystemen hinsichtlich der darin verwendeten Technologien ein andauernder Wandel statt. Insbesondere der Gebrauch moderner Technologien im Sinne des Cloud Computing nimmt dabei inzwischen einen essenziellen Bestandteil ein. Zu diesen gehört der Einsatz von *FaaS*, welches in den Bereich des *Serverless Computing* einzuordnen ist und bei welchem feingranulare Functions definiert werden [25, S. 1, S. 7 ff.].
Damit ein Softwaresystem hinsichtlich der Organisation bei seinem Entstehungs- und Wartungsprozess sowie bei der Softwareentwicklung selbst erfolgreich ist, bietet es sich an, die Zusammenarbeit zwischen den Akteuren zu verbessern und die strukturelle und taktische Herangehensweise mit dem Einsatz moderner Technologien zu verknüpfen. Insbesondere ein lang bewährtes Konzept wie DDD und eine neuartige Technologie wie FaaS bilden somit eine interessante Basis für moderne Softwaresysteme.
Im Zuge dieser Arbeit soll die Kombination untersucht und bewertet werden. Die Leitfrage, welche es am Ende der Arbeit zu beantworten gilt, lautet: *„Unter welchen Voraussetzungen lassen sich Domain-Driven Design und Function as a Service in einem Softwareprojekt verknüpfen und welche Hürden ergeben sich bei solch einer Kombination?“*

Dabei wird zunächst auf die Grundlagen von DDD und FaaS im Einzelnen eingegangen. Anschließend finden anhand eines zugrundeliegenden Fallbeispiels eine Analyse, ein Entwurf und eine Implementierung statt, sodass am Ende der Erfolg dieser Kombination mithilfe des Anwendungsbeispiels und der verwendeten Technologien evaluiert wird.

2 Grundlagen

Bevor die Kombination analysiert werden kann, müssen DDD und FaaS zunächst einzeln betrachtet und verstanden werden.
In diesem Kapitel wird zunächst auf DDD Bezug genommen und dabei zwischen den Bestandteilen des strategischen und taktischen DDD unterschieden. Anschließend wird auf FaaS eingegangen und es werden die Vor- und Nachteile von Functions dargelegt.

2.1 Domain-Driven Design

Wenn von DDD die Rede ist, geht es nicht nur um eine bestimmte Art und Weise, Software zu entwickeln. Einen Schwerpunkt bildet die Optimierung der Zusammenarbeit von Domain Experten und Softwareentwicklern. Mittels DDD wird eine Entwicklungsphilosophie und Herangehensweise vermittelt, welche besonders in komplexen Softwareprojekten von Vorteil ist [34, S. 3 ff., S. 11 f.].
Unter **Domain** wird beim DDD das Umfeld, in welchem sich ein Unternehmen bewegt und welche Tätigkeit es in diesem übernimmt sowie welches Know-How es mitbringt, verstanden [27]. Bei **Domain Experten** handelt es sich um Personen, welche über eine besondere fachliche Expertise der betrachteten Domain verfügen [48, S. 27]. Bei einer Software für Banken wäre dies bspw. eine Person, welche sich durch Wissen über diesen Fachbereich und über die Prozesse innerhalb einer Bank auszeichnet und dementsprechend das Fachwissen an die Softwareentwickler weitertragen kann.
Innerhalb von DDD wird des Öfteren zwischen strategischem und taktischem Design unterschieden. Hierzu gehören verschiedene Konzepte und Komponenten [27, 34, S. 3], auf welche in den folgenden Abschnitten 2.1.1 und 2.1.2 näher eingegangen wird.

2.1.1 Strategisches Domain-Driven Design

Strategisches DDD bezieht sich in erster Linie auf die analytische Herangehensweise und Planung vor der Implementierung eines Projekts. Mit dessen Hilfe ist es möglich, ein Projekt in verschiedene Teile zu untergliedern. Im Zuge des strategischen DDD liegt der Fokus darauf, welche Gesichtspunkte eine wichtigere bzw. eine weniger wichtige Rolle spielen und welchen Projektteilen dementsprechend besonders viel Beachtung geschenkt werden muss [48, S. 7 f.].
Nachfolgend werden die unterschiedlichen Bestandteilte des strategischen DDD aufgegriffen und erläutert.

Subdomains

Die Domain selbst lässt sich in mehrere Subdomains unterteilen. Wird dabei von der Core Domain gesprochen, liegt der Schwerpunkt auf den Alleinstellungsmerkmalen eines Unternehmens, d.h. durch welche Besonderheiten es sich charakterisieren lässt und von seiner Konkurrenz abweicht. Im Gegensatz dazu existieren außerdem Subdomains, welche die Core Domain zwar unterstützen und eine gewisse Expertise abverlangen, jedoch von geringerer Bedeutung als diese sind. Diese werden als Supporting Subdomains bezeichnet. Schließlich bilden die Generic Subdomains eine weitere Kategorie. Diese sind für ein System unabdingbar, jedoch nicht individuell auf jenes zugeschnitten und benötigen somit weniger Expertise als bspw. die Core Domain [48, S. 46 f.], [27].

Ubiquitous Language

Die Ubiquitous (*dt.: allgegenwärtig*) Language stellt einen allgemein gültigen Sprachraum innerhalb der Domain dar. Sie bildet einen der wichtigsten Aspekte im DDD, da mit dieser sichergestellt werden soll, dass keine Missverständnisse entstehen und Klarheit bei Gesprächen über die Domain besteht, da alle beteiligten Personen mit denselben Begrifflichkeiten hantieren. Die Einbindung von Domain Experten in die Definition eines gemeinsamen Sprachraums ist unabdingbar für eine korrekte Beschreibung der Domain und darf somit nicht unterschätzt werden [14, S. 24 ff.], [27].
Innerhalb sehr kleiner Domains ist es möglich, lediglich über eine übergreifende Ubiquitous Language zu verfügen. [27]. In komplexen Domains ist es wiederum notwendig,

für verschiedene Bereiche auch über eigene Variationen der allgemeinen Ubiquitous Language zu verfügen. Bei diesen Bereichen handelt es sich um die *Bounded Contexts* einer Domain [14, S. 345], [27], auf welche nachfolgend eingegangen wird.

Bounded Context

Bounded Contexts bilden einen separaten Kontextraum innerhalb der Domain. Jeder Bounded Context ist für eine spezielle Aufgabe zuständig [48, S. 11ff.]. Die Auftrennung der Zuständigkeiten in verschiedene Bounded Contexts verhindert, dass ein *Big Ball of Mud* entsteht, da keine festen Grenzen definiert worden sind. Solch ein unübersichtliches und eng gekoppeltes System ist u.a. das Resultat einer schlechten Kommunikation zwischen Domain Experten und Entwicklern und daraus folgend einer unzureichend geplanten Projektstruktur [48, S. 16 ff.].
Im Idealfall lässt sich ein Bounded Context auf eine Subdomain abbilden [48, S. 45 f.]. Zudem können Bounded Contexts jeweils über eigene Datenbanken oder eigene Datenbankschemata verfügen, damit die Kontextabgrenzungen nicht nur auf fachlicher Ebene in der Domain, sondern auch bezüglich ihrer Persistenz erfolgt [34, S. 109 ff.].
Aufgrund ihrer Ähnlichkeiten ließe sich vermuten, ein Bounded Context würde sich in jedem Fall auf einen Microservice beziehen. Zwar ist es durchaus möglich, dies so umzusetzen, jedoch nicht notwendig, da sich die Bounded Context über die Grenzen ihrer zugehörigen Ubiquitous Language und andere fachliche Zusammenhänge definieren, wohingegen es sich bei Microservices um unabhängig deploybare Einheiten handelt [42]. So könnten bspw. auch zwei Microservices innerhalb eines Bounded Contexts angesiedelt sein [48, S. 43].

Problemraum und Lösungsraum

Im DDD wird des Öfteren eine Unterteilung in zwei Bereiche vorgenommen: dem Problemraum (engl.: problem space) und dem Lösungsraum (engl.: solution space). Während der Fokus im Problemraum auf der Analyse der Gegebenheiten im Projekt liegt, wird in dem Lösungsraum die tatsächliche Implementierung und die Vorgehensweisen bei dieser eingeordnet [48, S. 12]. Auf strategischer Ebene sind die Subdomains im Problemraum anzusiedeln, während die Bounded Contexts dem Lösungsraum zugeordnet werden[13, S. 122].

2.1.2 Taktisches Domain-Driven Design

Neben der strategischen Herangehensweise existiert im DDD zusätzlich ein taktischer Ansatz. Beim taktischen DDD liegt der Fokus vermehrt auf der Nähe zur Implementierung. Das Ziel hierbei ist ein höheres Maß an Klarheit bei der Implementierung des zuvor geplanten Projekts zu erlangen und eine genauere Vorstellung davon, wie die verschiedenen Komponenten technisch zu realisieren sind, sodass eine Implementierung ohne Schwierigkeiten gelingt [28].
Nachfolgend werden verschiedene Modellierungshilfen für das Designen einer auf DDD basierenden Software betrachtet.

Entity

Bei einer Entity handelt es sich um ein einzelnes, veränderbares und individuelles Objekt innerhalb der Domain. Entities definieren sich darüber, dass sie beständig sind und dementsprechend über eine Lebenszeit verfügen. Die Entities haben über ihre Lebenszeit hinweg Eigenschaften, welche sich jedoch zwischendurch verändern können. Dennoch handelt es sich weiterhin um dasselbe Objekt. Um diese Individualität der Entities zu gewährleisten, wird ihnen eine einzigartige ID zugeordnet, welche über die gesamte Lebenszeit bestehen bleibt [47, S. 362 ff.], [28].
Ein Beispiel für eine Entity wäre eine Person, welche über ihre Lebenszeit hinweg dieselbe Identität beibehält, auch wenn sich bspw. ihre Adresse aufgrund eines Umzugs ändert.

Value Object

Value Objects repräsentieren unveränderbare, nicht individuelle Objekte. Diese verfügen folglich im Gegensatz zu Entities über keine einzigartige ID. Wie der Name bereits zu erkennen gibt, liegt der Fokus des Value Objects auf seinem Wert. Durch die Relevanz des Wertes eines Value Objects ist dessen Beständigkeit begründet. Mithilfe von Value Objects können u.a. Entities in Form verschiedener Eigenschaften beschreiben werden [47, S. 330 ff.], [28].
Analog zum vorangegangenen Beispiel handelt es sich bei der Adresse um ein Value Object, welches sich durchaus verändern kann und eine Eigenschaft der Entity *Person* darstellt.

Aggregate

Aggregates stellen fachliche Transaktionsgrenzen innerhalb der Bounded Contexts dar [48, S. 75]. Ein Aggregate besteht aus einer oder mehreren Entities. Abgesehen von den Entities können neben diesen zusätzlich auch Value Objects in einem Aggregate auftreten, jedoch ist dies nicht zwingend notwendig. Innerhalb eines Aggregates existiert immer eine Entity, welche als Aggregate Root (*dt.: Wurzel*) fungiert [48, S. 77]. Die Aggregate Root bildet die Wurzel des Aggregates, worüber dieses mithilfe einer ID identifiziert werden kann. Außerhalb des Aggregates ist es lediglich möglich, dieses über die global gültige ID seiner Wurzel zu referenzieren. Die restlichen Entities, welche nicht als Root innerhalb des Aggregates fungieren, verfügen in dessen Inneren über eine lokal gültige ID, über welche sie sich dort identifizieren lassen [28].
Als Beispiel würden die zuvor erwähnte Person und ihre Adresse ein kleines Aggregate bilden, in welchem die Person die Aggregate Root darstellen würde.
Zudem sollen Aggregates immer als ein Ganzes behandelt und gespeichert werden, damit die Transaktionsgrenze eingehalten wird und das Aggregate in einem konsistenten Zustand bleibt [48, S. 78 f.].

2.1.3 Domain Events

Von Domain Events wird gesprochen, wenn Ereignisse vorliegen, welche innerhalb einer Domain auftreten, wie z.B. eine eingehende Bestellung oder das Erstellen eines neuen Kunden. Diese Events können sowohl in einem Bounded Context, als auch kontextübergreifend stattfinden, je nachdem welche Bestandteile der Domain an dem Domain Event interessiert sind [48, S. 99, S. 105 f.]. Dabei wird ein Domain Event von einem Aggregate erzeugt und spricht somit diejenigen Bounded Contexts an, für welche das Ereignis relevant ist. Die Kommunikation mittels Domain Events zwischen den Bounded Contexts ist insbesondere daher von hoher Relevanz, da sie wie in Abschnitt 2.1.1 erwähnt unterschiedliche Datenbanken nutzen. Somit muss gewährleistet werden, dass trotz der verteilten Struktur weiterhin Konsistenz zwischen den verschiedenen Bounded Contexts der Anwendung herrscht [28, 48, S. 14].
Im DDD herrscht die Konvention, die Domain Events so zu benennen, dass sie ein Verb verkörpern über das, wofür das Event zuständig ist und gleichzeitig im Präteritum ausgedrückt, also z.B. CustomerCreated oder OrderReceived [47, S. 289]. Des Weiteren ist darauf zu achten, dass nicht jede einzelne Methode in ein Domain Event ausgelagert

wird, sondern vielmehr nur diejenigen Ereignisse, welche einen relevanten Geschäftsprozess innerhalb der Domain repräsentieren [50].

Eventual Consistency

Im Zusammenhang mit Domain Events spielt **Eventual Consistency** eine wichtige Rolle. Dabei handelt es sich um eine milde Form der Konsistenz. Diese ist häufig in verteilten Systemen anzutreffen und kann ebenfalls in Bezug auf Domain Events im DDD eingesetzt werden. Das bedeutet, dass sich die Daten in den verschiedenen Datenbanken der Bounded Contexts zwischenzeitig unterscheiden können und nicht klar ist, wann genau die Datenbanken alle auf demselben Stand sind, nur dass dieser Fall eintreffen wird. Wann ein Domain Event aus einem Bounded Context auftritt, hängt u.a. von der Auslastung der anderen Bounded Contexts ab, wann sie ihre Datenbank mithilfe der zwischenzeitig eingegangenen Domain Events auf den neusten Stand bringen [28]. Dieser Zusammenhang ist in Abbildung 2.1 zu sehen.

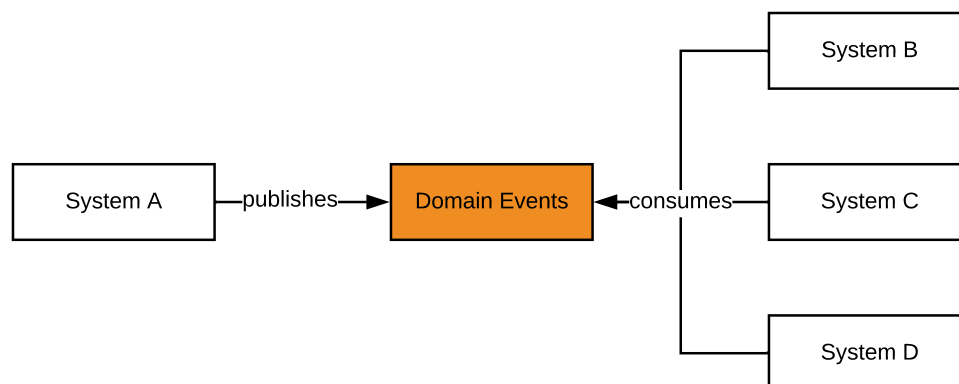

Abbildung 2.1: Domain Event mit Eventual Consistency [28].

2.2 Function as a Service

FaaS bildet einen Teilbereich des serverless Computing. Der Begriff *serverless* bedeutet in dem Zusammenhang nicht etwa, dass kein Server mehr benötigt wird, sondern dass

sich die Entwickler einer Anwendung nicht weiter wie bisher mit dem Server beschäftigen müssen (z.B. bezüglich des Betriebs). Stattdessen existiert dieser für die Entwickler lediglich im Hintergrund bei einem entsprechenden Anbieter [5, 33].
Wird der Begriff *serverless* verwendet, wird für gewöhnlich zwischen *Backend as a Service (BaaS)* und FaaS unterschieden. BaaS steht für serverseitige Funktionalitäten, die von Cloud Anbietern bereitgestellt werden [25, S. 8], [9]. Hierbei handelt es sich bspw. um ein angebotenes Authentifizierungsverfahren, sodass die Logik für diese Standardfunktionalität nicht vom Entwickler selbst implementiert werden muss [41]. Von FaaS wird gesprochen, wenn die Geschäftslogik weiterhin vom Entwickler implementiert wird und in Form kleiner Codeabschnitte als Funktionen gekapselt werden. Anschließend werden die Funktionen in unabhängigen Containern bereitgestellt. Diese existieren nur zum Zeitpunkt ihrer Ausführung [25, S. 9 f.], [10].
Besonders hinsichtlich der Lastverteilung bietet FaaS einen großen Vorteil gegenüber herkömmlichen Anwendungen. So wird bspw. je nach Aufrufhäufigkeit einer Funktion ihre Skalierung automatisch angepasst. Dementsprechend ist es von Vorteil, wenn die Funktionen möglichst feingranular geschrieben werden, damit die Skalierung genaustens erfolgen kann [33, 10], [25, S. 9].
Der Fokus dieser Arbeit liegt auf serverless Computing mit FaaS.

2.2.1 Aufbau von Functions

Nachfolgend werden die FaaS Funktionen in dieser Arbeit als *Functions* bezeichnet. Innerhalb einer Function wird eine bestimmte Aufgabe umgesetzt. Dementsprechend können innerhalb einer Anwendung zahlreiche Functions eingebunden sein. Nach der Implementierung werden die Functions bspw. in Containern auf eine Plattform ausgelagert, welche sich mit der Verwaltung dieser beschäftigt [40]. Solche Plattformen werden von diversen Anbietern gehostet angeboten, jedoch ist es ebenfalls möglich, FaaS Plattformen auf eigener Infrastruktur zu hosten. Diese beiden Möglichkeiten werden in Abschnitt 2.2.2 behandelt.
Um diese Functions dann auszuführen gibt es die Möglichkeit, von ereignisbasierten Triggern Gebrauch zu machen. Mit diesen werden Functions erst beim Eintreffen eines bestimmten Ereignisses ausgeführt [33]. Wird bspw. ein HTTP-Trigger verwendet, wird eine Function mittels eines HTTP Requests aufgerufen [17].

2.2.2 Anbieter

Für die Verwaltung der in Containern bereitgestellten Functions stehen verschiedene Anbieter, meist in Form eines Cloud-Anbieters, zur Verfügung [40]. Für FaaS existiert mittlerweile ein großes Angebot an Plattformen. Zu den bekanntesten und größten gehosteten Plattformen und deren Betreibern zählen u.a.: [8]:

- Amazon Web Services (AWS) Lambda
- Microsoft Azure Functions
- Google Cloud Functions
- IBM Cloud Functions

Neben den genannten kostenpflichtigen Anbietern existieren zusätzlich auch kostenlose selbst gehostete Open Source Angebote für FaaS. Hierzu zählen z.B.: [8]:

- OpenFaaS
- Kubeless
- Fission
- Apache OpenWhisk

Je nachdem, welcher Anwendungsfall vorliegt, ist ggf. ein anderes Framework die bessere Wahl. Somit müssen im Vorhinein die Konditionen und Charakteristiken der unterschiedlichen Anbieter miteinander verglichen werden, um eine optimale Wahl für den eigenen Anwendungsbereich zu treffen [6]. Ist es bspw. gewünscht, ein selbst gehostetes Framework innerhalb eines Kubernetes Clusters zu betreiben, bietet es sich an, auf Frameworks wie Fission oder Kubeless zurückzugreifen [39].

2.2.3 Vor- und Nachteile

Wie bereits in der Einführung zu Functions erwähnt, liegt ein großer Vorteil dieser in ihrer Skalierbarkeit. Je nachdem, welche Function wie oft genutzt wird, passt sich die in Anspruch genommene Kapazität mittels der genutzten FaaS-Plattform automatisch an. Somit entstehen keine überflüssigen Kosten und die Abrechnung kann genaustens anhand der Nutzung erfolgen. Ein weiterer Vorteil entsteht durch die Möglichkeit der

Entwickler, sich mehr auf das Programmieren und den Code selbst zu konzentrieren, als auf den Betrieb und das Verwalten der Infrastruktur [29].
Ein Nachteil bei der Anwendung von Functions ist, dass je nach Plattform unterschiedliche Herangehensweisen existieren, die Functions zu implementieren und zu verwenden. Die Unteschiede können sowohl den Code betreffen als auch Komponenten, welche zum Betrieb verwendet wurden, wie bspw. für das Monitoring, sodass folglich die Wahl des Anbieters im Voraus gut durchdacht werden sollte [41].
Durch diese Unterschiede gestaltet es sich aufwendig, den Anbieter zu wechseln, wenn dessen Plattform bereits seit geraumer Zeit genutzt wird, wodurch ein sog. Vendor Lock-in entsteht. Dies besagt, dass, sobald ein Anbieter gewählt wurde, es immens kompliziert ist, diesen zu wechseln, da die eigene Anwendung sich zu stark an den ursprünglichen anbieterspezifischen Konventionen orientiert [41].

2.2.4 Anwendungsfälle

Es gibt diverse Anwendungsgebiete für FaaS, in welchen die zuvor genannten Vorteile genutzt werden. Gerade das feingranulare Auftrennen von Functions und dessen separate Skalierbarkeit bieten für einige Anwendungsfälle einen besonders großen Vorteil. Dazu gehören bspw. Web-Apps in welchen Functions, wie in Abschnitt 2.2.1 beschrieben, mithilfe von HTTP-Triggern ausgeführt werden können [33, 40]. Folglich können durch die HTTP-Trigger bspw. auch Datenbankabfragen in Folge einer Useranfrage hin getätigt werden, ohne, dass dafür dauerhaft ein Server laufen muss [41].
Ein weiteres Beispiel in welchem auf FaaS zurückgegriffen wird, ist der Sprachdienst Alexa von Amazon. Dabei liefert die vom Anwender angefragte Function ein Ergebnis und geht anschließend wieder in ihrem Ausgangszustand über [33].

3 Anwendungsfall und Analyse des Zusammenspiels von DDD mit FaaS

Anschließend an die vorangehenden Kapitel, in welchen die Aspekte, Anwendungsfälle, sowie die Vor- und Nachteile im Einzelnen beleuchtet wurden, stellt sich in diesem Kapitel die Frage, inwieweit eine Kombination von DDD und FaaS praktikabel ist und in welchen Fällen sich dieses Arrangement lohnen würde.

3.1 Anwendungsfall

Angenommen, ein Restaurant hat den Wunsch, seine Lieferdienste online anzubieten. Hierbei spielen u.a. die Kunden des Restaurants eine Rolle, welche eine Bestellung mit ihren gewünschten Gerichten aufgeben können. Nach der Bestellung soll der Kunde einen Zustellzeitpunkt erhalten. Dieser hängt unter anderem von der Auslastung, also von der Gesamtanzahl an offenen Bestellungen beim Restaurant als auch von der Distanz zum Kunden ab. Sobald das Restaurant die Bestellung verarbeitet, mindert sich der Zutatenbestand. Wird ein kritischer Zutatenbestand des Restaurants erreicht, gibt dieses bei seinem Zulieferer eine Bestellung auf, um sich die Zutaten nachliefern zu lassen. Für diese Bestellung erhält das Restaurant ein Lieferdatum. Die Bestellung, die beim Zulieferer eingegangen ist, mindert wiederum dessen Ressourcenbestand.

3.2 Analyse

Basierend auf dem zugrundeliegenden Fallbeispiel können Anforderungen an ein System, welches die vorgegebenen Funktionalitäten bietet, herausgearbeitet werden.
Zunächst wird aus dem Anwendungsfall ersichtlich, dass sich in dem gewünschten System drei verschiedene Akteure befinden: der Kunde, das Restaurant und der Zulieferer.

Je nachdem, aus welcher Perspektive die Anwendung betrachtet wird, ergeben sich unterschiedliche Anforderungen. So resultieren für den Kunden User Stories wie:

- Als Kunde kann ich eine Mahlzeit auswählen.
- Als Kunde kann ich eine Bestellung aufgeben.
- Als Kunde kann ich den Lieferzeitpunkt meiner Mahlzeit einsehen.

Während auf der Seite des Kunden die Anwendung direkt durch diesen bedient wird, geschieht beim Restaurant vieles automatisch. So muss der Restaurantbetreiber nicht immer den Zutatenbestand prüfen, sondern dieser wird automatisch vermindert, je nachdem welche Mahlzeit vom Kunden bestellt wurde. Außerdem wird ab einem kritischen Zutatenbestand ebenfalls ohne Zutun des Restaurantbetreibers eine Bestellung an dessen Zulieferer getätigt, in welcher die fehlenden Zutaten nachbestellt werden. Auch die Auslastung des Restaurants und der Zustellzeitpunkt, welcher dem Kunden übermittelt wird, sollen vom System berechnet werden. User Stories, welche sich für das Restaurant bzw. dessen Betreiber ergeben, wären dann z.B.:

- Als Restaurantbetreiber kann ich meine Kunden einsehen.
- Als Restaurantbetreiber kann ich meinen Bestand einsehen.

Dem Zulieferer wird in dem Fallbeispiel weniger Beachtung geschenkt. Bei diesem wird ebenfalls automatisch beim Eingehen einer Bestellung des Restaurants lediglich dessen Ressourcenbestand vermindert. In dieser Arbeit werden die weiteren Hintergrundprozesse, wie z.B. dessen Warenlager etc. jedoch nicht betrachtet. Stattdessen liegt der Fokus auf den zwei zuvor genannten Hauptakteuren. Der Zulieferer selbst spielt somit in dem System lediglich für das Restaurant eine Rolle, wird aber nicht weiter als aktiver Akteur der Anwendung behandelt.
Nachdem die Anforderungen des Fallbeispiels analysiert wurden, muss das System mittels einer Spezifikation für den Kunden verständlich dargelegt werden, damit sich die Entwickler mit den Kunden abstimmen können und ggf. Änderungen vorgenommen werden können, wenn das System noch nicht alle gewünschten Anforderungen erfüllt. Der gesamte Projektablauf findet idealerweise in einem iterativen Prozess statt, sodass sich die Entwickler stetig mit den Kunden auseinandersetzen können. So können anhand der andauernden Kommunikation Fehler vermieden werden und die Anforderungen des Kunden stets korrekt geplant und umgesetzt werden [12, S. 759 f.].

Zu dem Fallbeispiel ergibt sich das in Abbildung 3.1 zu sehende Anwendungsfalldiagramm für das Essensliefersystem, welches im Verlauf der Arbeit als *FoodDeliverySystem* bezeichnet wird. Außerhalb des Systems sind die relevanten Akteure *Customer*, *Restaurant manager* und *Supplier* zu erkennen. Außerdem befindet sich dort ein Nachbarsystem *SupplierSystem* welches im Zusammenhang mit dem Zulieferer steht. Innerhalb des FoodDeliverySystems sind verschiedene Anwendungsfälle zu erkennen, welche im Rahmen des Systems stattfinden können. Einige Anwendungsfälle hängen direkt mit dem Stattfinden anderer Ereignisse zusammen. Bspw. muss immer beim Anlegen einer Essensbestellung des Kunden *(Order meal)* der Bestand gemindert werden *(Reduce stock)*. Solche zwingend zusammenhängenden Anwendungsfälle sind über Pfeile miteinander verbunden, welche mit «include» beschriftet sind. Der Anwendungsfall *Order ingredients* findet nicht immer beim Prüfen des Bestands im Sinne von *Check stock* statt, sondern lediglich beim Erreichen eines kritischen Zutatenbestandes. Daher sind diese beiden Fälle über die Bezeichnung «extend» miteinander verknüpft. Mithilfe dieses Diagramms wird deutlich, an welchen Stellen das System zusammenarbeitet und inwiefern die außenstehenden Akteure und Systeme eine Rolle spielen.

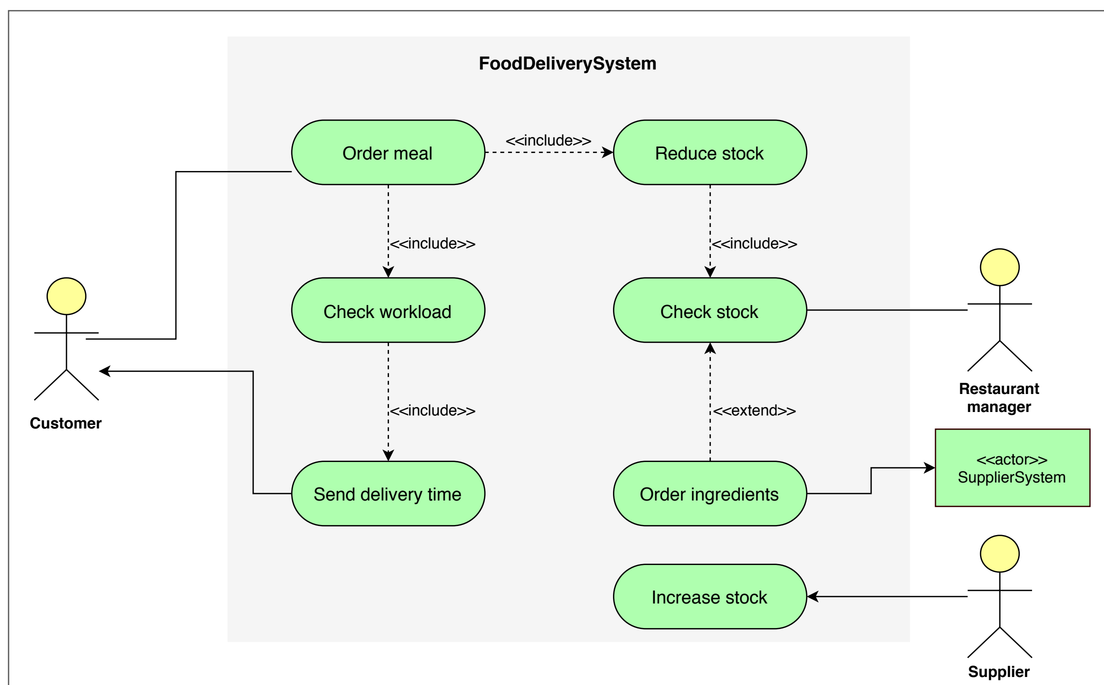

Abbildung 3.1: Anwendungsfalldiagramm für den Bestellprozess bei einem Restaurant.

Im nächsten Schritt wird ein Fachliches Datenmodell mit UML erstellt. Dabei werden

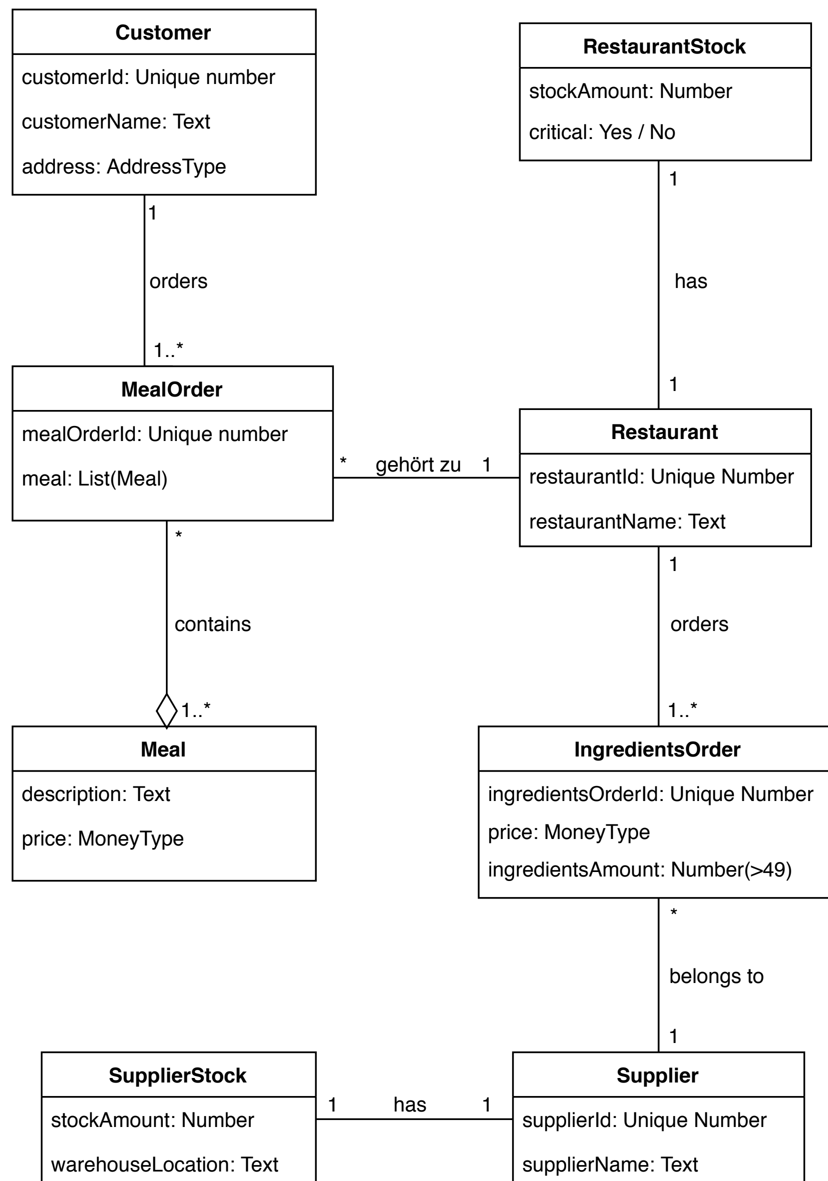

Abbildung 3.2: Fachliches Datenmodell für das FoodDeliverySystem.

noch keine softwaretechnischen Lösungsansätze dargestellt, sondern vielmehr die fachlichen Zusammenhänge erläutert. Das in Abbildung 3.2 zu sehende Fachliche Datenmodell ist dementsprechend keine Darstellung der geplanten Implementierung, sondern dient einem tieferen Verständnis und weiteren Abstimmungen mit dem Kunden. In der Abbildung werden bereits einige aus der Analyse resultierende Beispielsklassen dargestellt und über Assoziationen miteinander in Zusammenhang gebracht. Den zu den fachlichen Bei-

spielklassen zugehörigen Attributen werden an dieser Stelle keine technischen Datentypen zugeordnet. Folglich handelt es sich auch bei diesen um fachliche Datentypen, welche im Zuge einer Implementierung konkretisiert werden.
In dem Diagramm befinden sich neben dem *Restaurant*, dem *Customer* und dem *Supplier* auch andere wesentliche Bestandteile, welche für das Fallbeispiel eine Rolle spielen. Hierzu gehören:

- *MealOrder*: eine Essensbestellung des Kunden beim Restaurant
- *Meal*: die Mahlzeit, welche in der *MealOrder* enthalten ist
- *RestaurantStock*: der Bestand des Restaurants
- *IngredientsOrder*: eine Zutatenbestellung des Restaurants beim Zulieferer
- *SupplierStock*: Der Bestand des Zulieferers

Aus dem Diagramm wird ersichtlich, dass es für einen Kunden möglich ist, beliebig viele Essensbestellungen zu tätigen, jedoch mindestens eine, da es sich ohne getätigte Essensbestellung nicht um einen Kunden handeln würde. Eine Essensbestellung ist wiederum immer lediglich einem Kunden zuzuordnen. Eine Essensbestellung enthält in jedem Fall mindestens eine Mahlzeit, es handelt sich bei der Mahlzeit also um einen bestimmten Teil der Essensbestellung. Somit wird die Beziehung zwischen den beiden als Aggregation aufgeführt. Zudem kann eine Mahlzeit in beliebig vielen Essensbestellungen vorkommen, somit möglicherweise auch nie bestellt werden. Im Diagramm ist außerdem zu erkennen, dass sowohl der Zulieferer als auch das Restaurant jeweils über eine 1:1 Beziehung zu ihrem Bestand verfügen. Der Unterschied zwischen dem Bestand des Restaurants und dem des Zulieferers liegt darin, dass das Restaurant über einen Wahrheitswert verfügt, welcher aussagt, ob der Bestand kritisch ist oder nicht. Außerdem enthält der Ressourcenbestand des Zulieferers eine Angabe über den Lagerort, da der Zulieferer möglicherweise mehrere Lager für seine Ressourcen nutzt. Die weiteren Prozesse des Zulieferers werden in dem Fallbeispiel nicht weiter betrachtet, um das Beispiel anschaulich zu halten und da sie für diese Arbeit keinen Mehrwert liefern würden.

3.2.1 Analyse mit Fokus auf DDD

In Anschluss an die allgemeine Analyse des Fallbeispiels, folgt die Analyse des Sachverhalts gemäß DDD. Vor der Verwirklichung der Anwendung muss zunächst die Domain

festgelegt werden. Wie aus der bereits erfolgten Analyse hervorgegangen ist, wird hier als **Domain** das Essensbestellsystem eines Restaurants angenommen, das *FoodDelivery-System*.
Mit der Festlegung der Domain stellt sich für den Anwendungsfall als nächstes die Frage nach einer sinnvollen Aufteilung in Bounded Contexts. Neben dem Restaurant spielen die Kunden und der Zulieferer des Restaurants eine tragende Rolle in der Domain. Wird der Bestellvorgang des Kunden und der des Restaurants betrachtet (wie in Abbildung 3.2 zu sehen ist) wird deutlich, dass es sich hierbei um verschiedene Arten von Bestellungen handelt. Gibt ein Kunde eine Essensbestellung auf, beinhaltet diese u.a. die Mahlzeit und dessen zugehörige Menge. Die darin enthaltene Mahlzeit enthält außerdem noch einen Preis. Tätigt das Restaurant eine Bestellung, handelt es sich um eine Zutatenbestellung bei seinem Zulieferer. Diese Art der Bestellung umfasst höhere Mengenangaben als eine Essensbestellung des Kunden, da der Zulieferer lediglich Großbestellungen annimmt. Den Begriff *Bestellung* zu verwenden wäre also unvorteilhaft, da es verschiedene Arten von Bestellungen innerhalb der Domain gibt. Selbes gilt für den Bestand des Zulieferers und den des Restaurants.
Anhand dieser Aspekte wird schnell deutlich, welche Aufteilung sich anbietet. Es entstehen innerhalb der Domain **drei Bounded Contexts**: *SupplierContext*, *RestaurantContext* und *CustomerContext*. Zwischen den Kontexten soll eine präzise Aufteilung entstehen. Der gesamte Sachverhalt lässt sich mithilfe einer Context Map, wie in Abbildung 3.3, anschaulich darstellen. In dieser sind ein weiteres Mal die wichtigsten Klassen und dessen Beziehungen zueinander abgebildet. Im Gegensatz zum vorher betrachteten Fachlichen Datenmodell in Abbildung 3.2 wird hier allerdings angesichts der Kontextaufteilung eine weitere Aufteilung der einzelnen Bestandteile vorgenommen. Außerdem existiert ein gemeinsam genutzter Ressourcenpool auf welchen der *CustomerContext* und der *RestaurantContext* gleichermaßen zugreifen dürfen, der sogenannte *Shared Kernel*.
Als Shared Kernel wird im DDD ein von verschiedenen Bounded Contexts gemeinsam genutztes Modell bezeichnet, welches diese sich teilen [48, S. 55]. Darin befindet sich für dieses Fallbeispiel die Klasse *Meal*. Von dieser leiten sich für die beiden genannten Kontexte ein eigenes *CustomerMeal* und ein *RestaurantMeal* ab. Der Grund für diese Unterteilung ist, dass ein *CustomerMeal* andere Bestandteile als ein *RestaurantMeal* hat. Ein Beispiel hierfür ist, dass die Mahlzeit auf der Kundenseite die Information enthält, ob sie derzeit verfügbar ist oder nicht bestellt werden kann, wohingegen diese Information nicht auf Seiten des Restaurants vorhanden ist. Damit zwischen den Bounded Contexts eine klare Trennung besteht, steht folglich auch nicht mehr die *MealOrder* als einzelne Klasse zwischen dem Restaurant und dem Kunden, sondern wird auf zwei Klas-

sen aufgeteilt: *CustomerMealOrder* und *RestaurantMealOrder*. Die Essensbestellung hat je nach Kontext eine eigene Bedeutung und beinhalten unterschiedliche Unterarten der Klasse *Meal*. Ein weiterer Grund für die getrennte Behandlung der Bestellungen in den zwei Bounded Contexts sind die Möglichkeiten, welche dadurch den Enwicklern gegeben werden. Wenn für jeden Bounded Context bspw. je ein Team zuständig ist, können die Entwickler des einen Kontexts problemlos Änderungen an den Klassen durchführen ohne dass der andere Kontext von diesen Änderungen betroffen ist.

Da die *CustomerMealOrder* und die *RestaurantMealOrder* jedoch im weiteren Verlauf fachlich zusammenarbeiten, müssen zwischen diesen Domain Events stattfinden, was in der ContextMap mittels einer gestrichelten Linie angedeutet wird. Jeder der vorliegenden Bounded Contexts hat somit ein anderes Verständnis von einer Bestellung, insbesondere im RestaurantContext, welcher nicht nur eine Art der Bestellung enthält, sondern sowohl eine *CustomerMealOrder*, als auch eine *IgredientsOrder* einbezieht. Dementsprechend ergibt sich für jeden Bounded Context eine eigene Ubiquitous Language. Durch die Eindeutigkeit in den Bezeichnungen wird sowohl unter den Domain Experten als auch unter den Entwicklern verständlicher, wo genau der Fokus des jeweiligen Kontexts liegt und es herrscht mehr Klarheit bei der Kommunikation. Dies unterstreicht nochmals die Relevanz der Ubiquitous Language innerhalb der Bounded Contexts.

In diesem Beispiel wird deutlich, dass die Kommunikation zwischen den verschiedenen Contexts für die Implementierung eines solchen Lieferservices unabdingbar ist. Der Kunde muss seine *CustomerMealOrder* an das Restaurant weiterleiten, dieses übergibt dem Kunden wiederum den voraussichtlichen Zustellzeitpunkt. Durch die *RestaurantMealOrder* mindert sich außerdem der Zutatenbestand des Restaurants, wodurch eine *IngredientsOrder* an den Zulieferer getätigt werden muss. All dies sind Beispiele für die Kommunikation zwischen den Bounded Contexts, welche mithilfe von Domain Events modelliert werden.

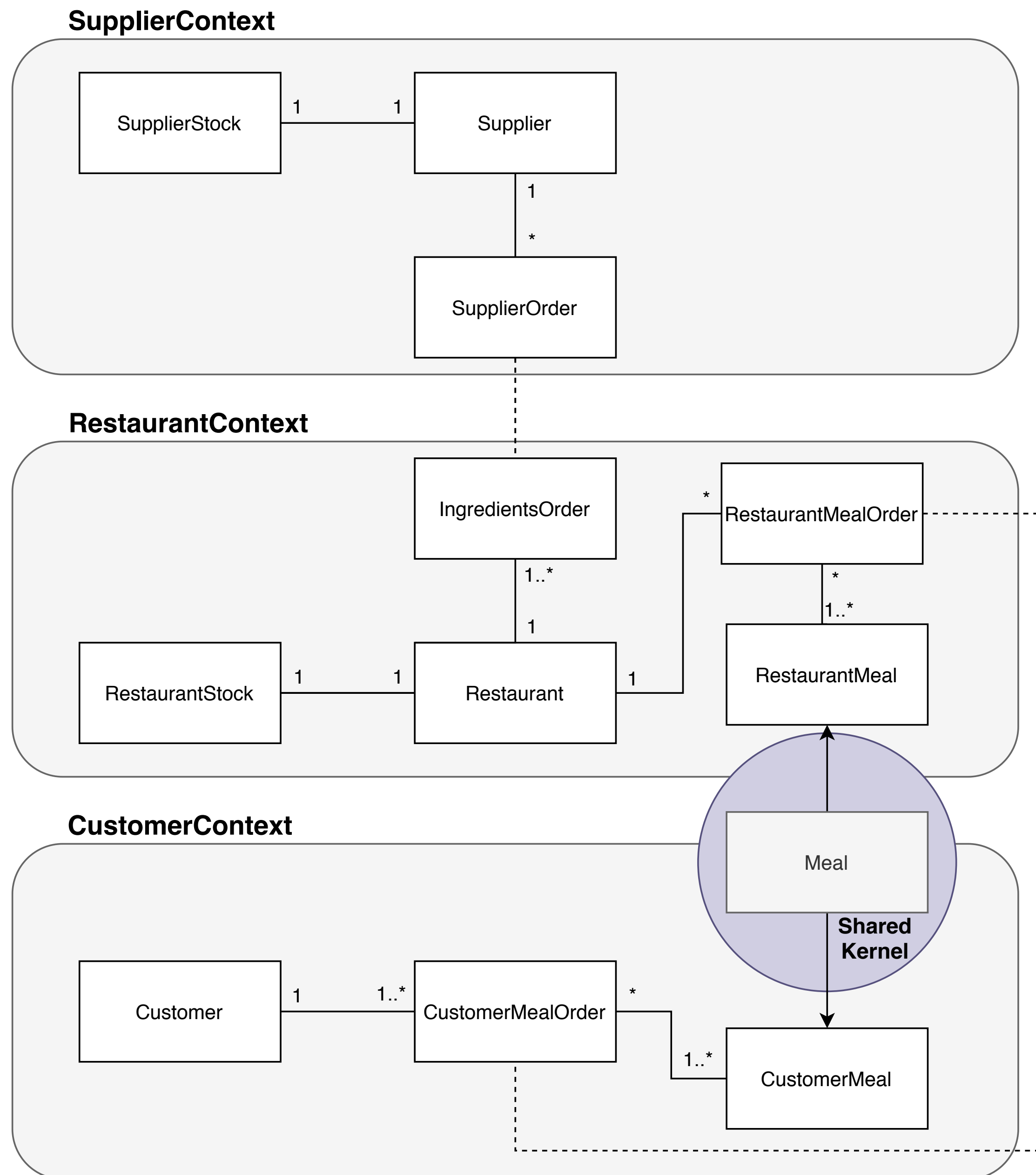

Abbildung 3.3: Context Map der Domain FoodDeliverySystem.

4 Entwurf

Damit die analysierten Elemente des Projekts korrekt implementiert werden bedarf es einem präzisen Entwurf, in welchem die wichtigsten Projektstrukturen dargestellt werden. Hierbei ist es besonders wichtig, FaaS auf eine effiziente Weise in die Elemente des DDD mit einzubringen.
Dabei ist ebenfalls die Wartbarkeit des Systems zu berücksichtigen, da sie ein entscheidender Faktor für den Erfolg und weiteren Verlauf eines Softwareprojekts ist [26, S. 8]. Ist die Wartbarkeit eines Systems mangelhaft, besteht die Gefahr einer dauerhaften Architekturerosion wodurch stetig technische Schulden aufgebaut werden [32, S. 5 f.]. Infolgedessen müssen im Schlimmstfall ganze Teile einer Anwendung erneut programmiert werden, wodurch ein hoher Zeitaufwand und folglich Kosten entstehen, welche hätten vermieden werden können. Somit ist es bereits bei der Erstellung eines Entwurfs von großer Bedeutung, das System so zu konzipieren, dass die Wartungsaufwände möglichst geringgehalten werden und mögliche Änderungen am System ohne Schwierigkeiten vorgenommen werden können [24, S. 3 ff.].
Da DDD hauptsächlich in komplexen Softwareprojekten angewendet wird [48, S. 2], muss für das Fallbeispiel somit ein Entwurf auf Basis der getroffenen Aspekte bezüglich der Wartbarkeit getroffen werden.

4.1 Schichtenmodell

Die Bounded Contexts in einer Domain können, wie bereits in Kapitel 2.1.1 erwähnt, über eigene Datenbanken verfügen. Dies ist ein ergänzender Schritt für den Einsatz des Design-Prinzips *Separation of Concerns (SoC)* [34, S. 109 ff.].
Bei SoC handelt es sich um die Trennung der Zuständigkeiten innerhalb eines großen Softwaresystems auf kleinere Einheiten [32, S. 76 f.]. Wird dieses Prinzip konsequent in ein Softwareprojekt eingebracht, erhöht sich dessen Wartbarkeit [7, S. 237]. Auch eine

klare Trennung zwischen verschiedenen Architekturschichten im Sinne eines Schichtenmodells hilft, das SoC Prinzip mit umzusetzen [11, S. 19].
So ist in Abbildung 4.1 eine klare Trennung zwischen der Anwendungsschicht, der Persistenzschicht und der Präsentationsschicht (*Graphical User Interface (GUI)*) zu sehen, was unabdingbar ist für eine Software, welche im Nachhinein gut lesbar und wartbar sein soll. Eine Trennung nach dem Drei-Schichtenmodell führt somit zu einer losen Kopplung zwischen den Schichten. [11, S. 17 ff.].

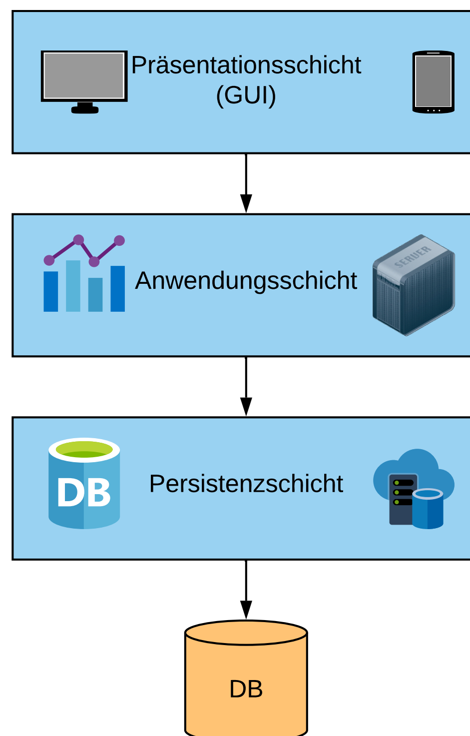

Abbildung 4.1: Drei-Schichtenarchitektur.

Das klassische Drei-Schichtenmodell, wie es in Abbildung 4.1 zu sehen ist, muss bei der Anwendung von FaaS allerdings angepasst werden. Dabei sollte beachtet werden, dass die Functions eine zentrale Rolle in solch einem System einnehmen. Functions können bspw. über ein API Gateway mittels HTTP Methoden wie GET, PUT, POST und DELETE ausgeführt werden. Die Function, welche durch den Aufruf getriggert wurde, kann dann je nach Anfrage lesend und/oder schreibend auf die Datenbank zugreifen [41]. Dieser Zusammenhang ist in Abbildung 4.2 veranschaulicht. Dort wird durch die gestrichelte

Linie eine Grenze zwischen zwei Bounded Contexts symbolisiert, wobei drei Functions dem ersten Bounded Context (*BC1*) und eine dem zweiten Bounded Context (*BC2*) zugeordnet ist.
Interessant wird an dieser Stelle ebenfalls der Zusammenhang zu DDD. Hier muss berücksichtigt werden, dass die Functions nicht willkürlich auf jegliche Datenbanken zugreifen dürfen. Je nachdem, in welchen Bounded Context sie einzuordnen sind, ergeben sich andere Zugriffsrechte und es muss sichergestellt werden, dass gegen diese im Zuge der Implementierung nicht verstoßen wird. Damit das System wartbar bleibt, müssen bei dem Entwurf einer Architektur somit bereits die Zugriffsrechte der Functions strikt beachtet werden.

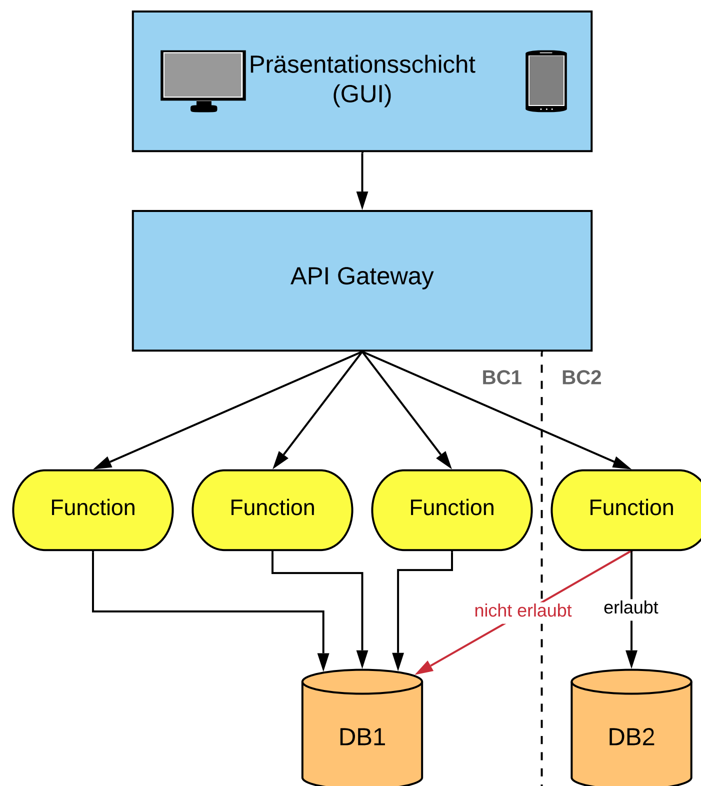

Abbildung 4.2: Beispiel einer Architektur unter Anwendung von FaaS.

4.2 Architektursichten

Im Rahmen dieser Arbeit liegt der Fokus auf einem Systementwurf, welcher sich sowohl mit DDD als auch mit FaaS vereinen lässt. Ein besonderes Augenmerk liegt dabei wie zuvor in 4.1 erwähnt, auf dem Umgang mit Functions.
Um möglichst alle Sichtweisen auf das Projekt zu veranschaulichen, bietet es sich an, die vier Architektursichten nach Starke zu betrachten, welcher zwischen der Kontextsicht, Bausteinsicht, Laufzeitsicht und Verteilungssicht unterscheidet [44, S. 156 f.].
Die benötigten Informationen zur Realisierung eines Softwareprojekts unterscheiden sich je nach der beteiligten Person [44, S. 155 f.]. Für den Kunden spielen z.B. insbesondere fachliche Aspekte eine Rolle sowie die Geschäftsprozesse, welche innerhalb eines Systems stattfinden. Aber auch auf der Ebene der Softwareentwicklung und Wartung sind unterschiedliche Sichtweisen auf das Projekt notwendig. Je nachdem, ob es sich um einen Softwarearchitekten, Entwickler, Administrator oder eine Person, welche den Betrieb des Systems übernimmt, handelt, sind die Informationen, welche sie zur Bewältigung ihrer jeweiligen Aufgabe benötigen unterschiedlich. Folglich sprechen die Architektursichten auch unterschiedliche Akteure an [44, S. 155 ff.].
Zu beachten ist, dass auch der Entwurf verschiedener Architektursichten immer in einem iterativen Prozess stattfinden sollte, da sich u.U. die äußeren Gegebenheiten und Anforderungen ändern können und außerdem Erkenntnisse einer Sicht Auswirkungen auf die Konzeption der anderen Sichten haben können [44, S. 158 f.].
Im Folgenden werden die verschiedenen Architektursichten nach Starke betrachtet. Abhängig von dem zugrundeliegenden Projekt fällt die Relevanz einer Sicht teilweise stärker oder geringer aus [44, S. 156]. Für das Fallbeispiel und die Kombination von DDD und FaaS werden insbesondere die Laufzeitsicht und die Verteilungssicht behandelt. Um Redundanzen in den Diagrammen zu vermeiden, wird im Folgenden nur flüchtig auf die Kontextsicht und die Bausteinsicht eingegangen, diese werden jedoch im Gegensatz zur Laufzeitsicht und Verteilungssicht nicht in einem separaten Unterkapitel betrachtet.

Kontextsicht

In der Kontextsicht (auch als *Kontextabgrenzung* bezeichnet) findet zunächst eine Abgrenzung der Kontexte und der Projektumgebung, in welcher das System sich befindet, statt. Dabei werden nicht die Elemente des Systems selbst betrachtet, sondern das System als Black-Box in Zusammenhang mit seiner Umgebung gesetzt [44, S. 157].

In Kapitel 3 wurde zuvor in Abbildung 3.1 auf ein Anwendungsfalldiagramm verwiesen, welches bereits einen Eindruck über die Umgebung des Systems liefert. In Abschnitt 4.2.2 werden schließlich in der Verteilungssicht weitere Bestandteile des Systems erläutert.

Bausteinsicht

Wie dem Namen dieser Sicht bereits zu entnehmen ist, geht es um die unterschiedlichen Teilstücke, aus welchen sich ein System zusammensetzt. Dabei spielen in erster Linie einzelne Klassen und Komponenten eine Rolle sowie die Art und Weise, auf welche diese miteinander zusammenhängen und kommunizieren [44, S. 157, S. 162 f.]. Als Diagramm für diese Sicht eignet sich bspw. auch ein UML-Klassendiagramm [44, S. 167] ähnlich wie das Fachliche Datenmodell in Abbildung 3.2.

4.2.1 Laufzeitsicht

Für das Fallbeispiel ist insbesondere die Laufzeitsicht von Relevanz. In der Laufzeitsicht wird dargestellt, auf welche Weise sich welche Teile des Systems zur Laufzeit verhalten [44, S. 169]. Somit ist das Interesse an dieser Sicht insbesondere unter Anbetracht der Functions hoch, da diese lediglich zum Zeitpunkt ihrer Ausführung existieren [25, S. 9] und im Anschluss wieder direkt zurück in ihren Ursprungszustand gelangen, in welchem sie jederzeit erneut aufgerufen werden können [33].
Functions möglichst feingranular zu implementieren, um diese unabhängig voneinander zu skalieren ist eines der zentralen Ziele bei FaaS. Viele Functions hängen von der Geschäftslogik her zwar zusammen, sollten aber dennoch unter dem Vorbehalt, auch einzeln deployed und einzeln skaliert werden zu können, implementiert werden [15, 10].
Auf das Fallbeispiel bezogen betrifft dies zum Beispiel das Erhöhen des Bestandes des Restaurants. Zwar sollte ab einem kritischen Bestand dieser prinzipiell immer erhöht werden, jedoch sollte sich der Restaurantbestand auch unabhängig von einem kritischen Bestand aufstocken lassen, bspw. vor Feiertagen, wenn viele Bestellungen und Gäste innerhalb des Restaurants zu erwarten sind. Analog dazu sollte sich auch der Restaurantbestand in dem Fallbeispiel nicht nur in Zusammenhang mit einer Bestellung beim Restaurant mindern, sondern auch die Möglichkeit bestehen, den unabhängig von eingehenden Bestellungen zu reduzieren. Dies wäre bspw. der Fall, wenn ein Teil der Waren verdorben ist.
Somit ist das direkte Aneinanderketten von Functions ebenfalls wie das definieren zu

großer Functions unvorteilhaft und auch für dieses Anwendungsbeispiel ungeeignet. Auf diese Weise entstehen zu viele Abhängigkeiten zwischen den einzelnen Teilen der Anwendung und die Ladezeiten im Fall einer großen Function können sich erhöhen [15]. Dieser Aspekt sollte daher ebenfalls beim Entwurf der Laufzeitsicht berücksichtigt werden.
Unter Berücksichtigung der zuvor dargelegten Aspekte muss analysiert werden, auf welche Weise die Functions optimal in ein Diagramm für die Laufzeitsicht mit eingebracht werden können. Im Folgenden wird außerdem gezeigt, auf welche Weise eine Function ausgelöst wird und sich im Anschluss daran verhält. Mittels eines Sequenzidagramms ist es möglich, diese Zusammenhänge zu veranschaulichen [44, S. 171]. Dabei bieten sich unterschiedliche Varianten an.
In diesem Abschnitt werden zwei Sequenzdiagramme vorgestellt, welche eine beispielhafte Modellierung eines Teilausschnittes des FoodDeliverySystems abbilden. In dem Sequenzdiagramm 4.3 wird der Bestellvorgang des Kunden mittels einer `create-order` Function beschrieben. Laut des Diagramms wird vom Kunden ein HTTP POST Request an einen HTTP-Trigger geschickt, welcher das Ausführen der `create-order` Function auslöst. Durch diese wird anschließend sowohl eine neue `CustomerMealOrder` im Kontext des Kunden erstellt als auch eine neue `RestaurantMealOrder` im Kontext des Restaurants. Dabei wurde zur Übersichtlichkeit eine farbliche Abgrenzung der verschiedenen Bestandteile des Diagramms vorgenommen: die Function grün, der CustomerContext orange, der RestaurantContext blau und der HTTP-Trigger rot.
Eine Anwendung von Functions wie in Abbildung 4.3 wäre jedoch mit DDD nicht konform, da dieselbe Function in zwei verschiedenen Bounded Contexts arbeitet. Eine eindeutige Zuordnung der `create-order` Function zu einem Bounded Context ist auf diese Weise nicht möglich. Nicht nur, dass dies somit gegen die Prinzipien des DDD verstoßen würde, auch eine Zusammenarbeit verschiedener Teams würde erschwert werden. Wären diese bspw. je nach Bounded Context aufgeteilt, würde durch einen solchen Entwurf Unsicherheit darüber entstehen, welches Team die Verantwortung für das Implementieren der Function trägt. Somit ist ein Aufbau nach diesem Sequenzdiagramm ungeeignet.
In Abbildung 4.4 ist ebenfalls ein Sequenzdiagramm zu einem Teilausschnittes des zugrundeliegenden Anwendungsbeispiels dargestellt. In diesem Ausschnitt wird erneut der Bestellvorgang des Kunden und die daraus entstehenden Bestellungen auf Seite des Kunden und des Restaurants beschrieben. Dabei bilden zwei unterschiedliche Functions, welche als Folge bestimmter Events auftreten, einen zentralen Bestandteil des Diagramms. Es handelt sich hierbei um eine Form der ereignisgesteuerten Architektur (*Event-Driven architecture*).
In Zusammenhang mit einer ereignisgesteuerten Architektur eignet sich eine MQ und

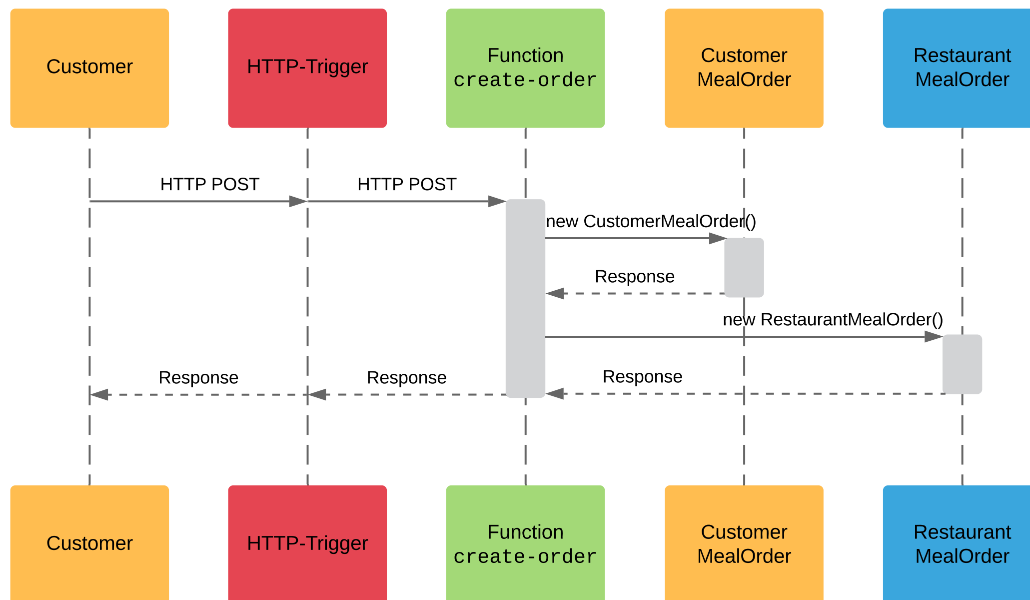

Abbildung 4.3: Sequenzdiagramm mit einer Function, welche in zwei Bounded Contexts agiert i.A.a. [15], *(orange: CustomerContext, rot: Trigger, grün: Function, blau: RestaurantContext)*.

dementsprechend auch ein MQ-Trigger, welche gemeinsam den Umgang mit eingehenden Events koordinieren [15]. Diese sind ebenfalls im Sequenzdiagramm abgebildet.
Im Gegensatz zum vorherigen Beispiel übernimmt nicht eine Function die Aufgaben in zwei Bounded Contexts, sondern der Vorgang wurde auf zwei Functions aufgeteilt. Mithilfe der `customer-order` Function wird eine Essensbestellung auf Seite des Kunden erstellt, die `restaurant-order` Function erstellt wiederum die Essensbestellung im Restaurantkontext.
Auf diese Weise entstehen keine Konflikte zwischen den Bounded Contexts und es besteht Klarheit darüber, in welchen Kontext die Functions einzuordnen sind. Mithilfe der MQ wird der Vorgang koordiniert. Wird das entsprechende Event zum Erstellen einer `CustomerMealOrder` vom `Customer` freigegeben (bspw. durch das Bestätigen des Vorgangs über die Anwendung), gelangt dieses Event asynchron an die MQ, da dieser auf keine Antwort der MQ warten muss, um eine nächste Bestellung aufzugeben. Der MQ-Trigger kann beim Erhalt eines bestimmten Events das Deployen der entsprechenden Function triggern, was bspw. im Fall des FaaS Frameworks Fission (dazu in Kapitel 5 mehr) mittels eines HTTP POST geschehen kann [22]. Ob das

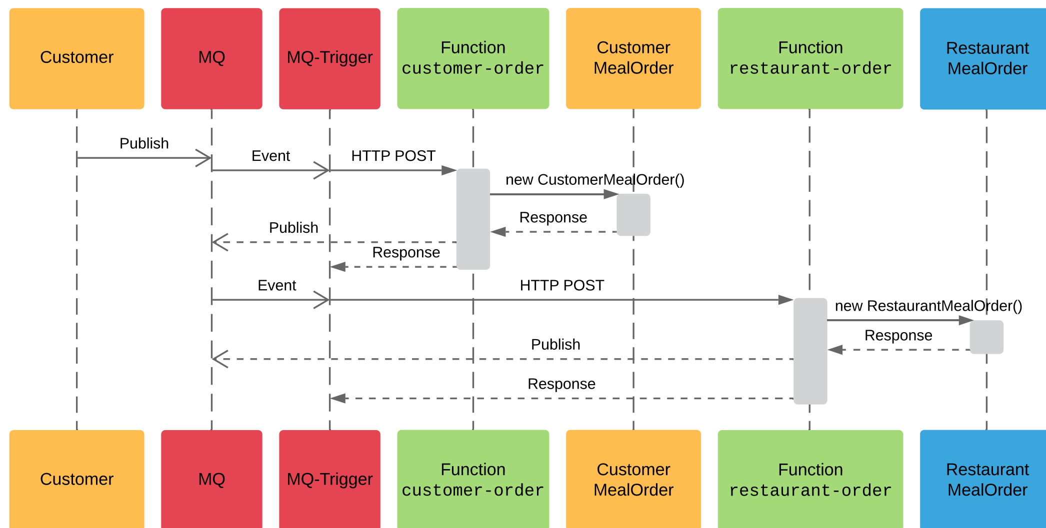

Abbildung 4.4: Sequenzdiagramm mit zwei Functions unter Anwendung einer Message Queue i.A.a. [15] *(orange: CustomerContext, rot: MQ und Trigger, grün: Functions, blau: RestaurantContext)*.

Erstellen der `CustomerMealOrder` erfolgreich war, wird über eine Response zurückgegeben. In diesem Beispiel wird außerdem durch die `customer-order` Function ein weiteres Event an die MQ asynchron gesendet und mithilfe des MQ- Triggers an die `restaurant-order` Function übergeben. Somit wird lediglich eine Essensbestellung auf Seiten des Restaurants erstellt, wenn auch die Essensbestellung beim Kunden erfolgreich war. Sollte aus einem Grund, wie bspw. einer nicht existierenden Adresse, die Essensbestellung beim Kunden abgelehnt werden, wird auch beim Restaurant nicht fälschlicherweise eine `RestaurantMealOrder` angelegt. Dennoch existieren die Functions unabhängig voneinander, sodass die `customer-order` Function mithilfe eines speziellen Events auch ohne Einwirkung der `restaurant-order` Function ausgeführt werden könnte. Das ausgehende Event welches von der der `restaurant-order` Function in dem Sequenzdiagramm an die MQ geschickt wird soll andeuten, dass auch nach dem erfolgreichen Erstellen der `RestaurantMealOrder` eine nachfolgende Function ausgelöst wird, hier aber lediglich ein Teilabschnitt des Systems dargestellt wird. Dieses Event könnte z.B. weiterführend ein Event zur Minderung des Restaurantbestands triggern.

4.2.2 Verteilungssicht

Mithilfe der Verteilungssicht (auch als *Infrastruktursicht* bezeichnet) wird die Ablauf- und Hardwareumgebung eines Systems dargestellt. Dabei können bspw. unterschiedliche Hardware-Bestandteile wie z.B. Speicher aufgezeigt werden, jedoch auch externe Nachbarsysteme [44, S. 172 f.]. Je nach Anwendungsfall variiert die Relevanz bestimmter Bestandteile in dem zugehörigen Diagramm [44, S. 174].

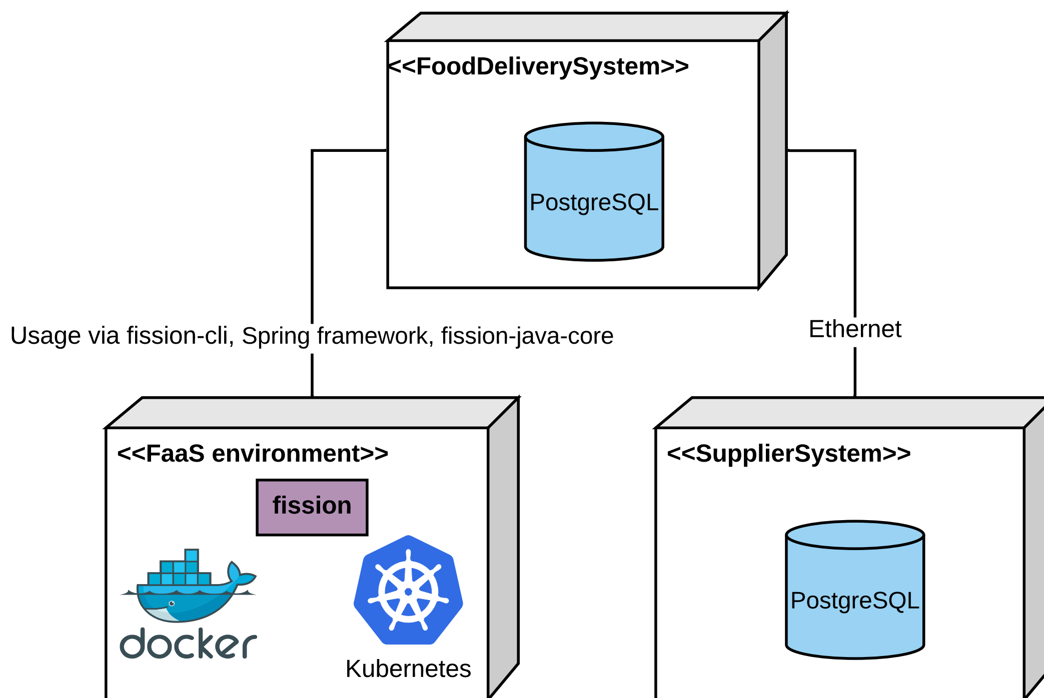

Abbildung 4.5: Das FoodDeliverySystem mit Nachbarsystem und FaaS-Umgebung.

In dem hier aufgezeigten Diagramm (Abbildung 4.5) ist eine vereinfachte Darstellung der Systemumgebung aufgezeigt. Im Zentrum steht das FoodDeliverySystem. Dieses kommuniziert über Ethernet mit dem SupplierSystem. Zudem verfügen beide über eine PostgreSQL Datenbank. Auf Seite des FoodDeliverySystems ist außerdem die Umgebung aufgezeigt, welche benötigt wird, um FaaS mit in das System zu integrieren. Dabei wird unter anderem auf Docker und Kubernetes hingewiesen, da in dem Anwendungsbeispiel mithilfe von Docker Desktop ein single-node Kubernetes Cluster aufgesetzt wird. In diesem Cluster wird das FaaS Framework Fission betrieben und mithilfe dessen die

Functions u.a. verwaltet und deployed. Auf diese Komponenten wird in Kapitel 5 näher eingegangen. Dem Diagramm ist außerdem zu entnehmen, dass das FoodDeliverySystem mithilfe der fission-cli, des Spring frameworks und dem fission-java-core die FaaS Umgebung inkludiert. Auch dies wird im o.g. Kapitel näher beschrieben, weshalb an dieser Stelle nicht weiter darauf eingegangen wird.

5 Implementierung

Im Anschluss an die Analyse und die daraus folgenden entwurfstechnischen Entscheidungen ist es möglich mit der Implementierung des Fallbeispiels zu beginnen. Dabei werden die in der Analyse gezogenen Schlüsse sowie die entworfenen Diagramme als Basis für den Aufbau des FoodDeliverySystems und der Projektstruktur verwendet. Im ersten Abschnitt des Kapitels werden weitere technische Voraussetzungen für die Implementierung getroffen. Anschließend werden die Implementierung und das genaue Vorgehen bei dieser erläutert.

5.1 Voraussetzungen für die Implementierung

Bevor mit der Implementierung begonnen werden kann, müssen weitere technische Entscheidungen getroffen werden. Im ersten Schritt muss die Wahl einer geeigneten Programmiersprache vorgenommen werden. Außerdem muss ein angemessenes und mit der Programmiersprache kompatibles FaaS Framework ausgewählt werden. Im Anschluss an diese beiden technischen Voraussetzungen ist es möglich, mit der eigentlichen Implementierung des FoodDeliverySystems zu beginnen.

5.1.1 Wahl der Programmiersprache

Im Zuge dieser Arbeit ist die Wahl einer geeigneten Programmiersprache wesentlich für die anschließende Implementierung und die daraus resultierenden Schlüsse, da nicht alle Programmiersprachen für eine Kombination von DDD und FaaS geeignet sind.
Wie bereits aus den entworfenen Diagrammen in Kapitel 4 abzulesen ist, wurde als Programmiersprache für diese Arbeit Java gewählt. Die Ergebnisse und Bewertung der Kombination beziehen sich somit ausschließlich auf einen Anwendungsfall mit der hier ausgewählten Programmiersprache.

Java ist eine der gefragtesten objektorientierten Programmiersprachen [45]. Für die Implementierung des FoodDeliveryService mit Java werden die Bounded Contexts auf Java packages abgebildet und darin die zugehörigen Java Klassen implementiert. Das Ziel dessen ist, durch diese strikte Unterteilung in Java Klassen nach Bounded Context einen leicht überschaubaren Projektaufbau zu konstruieren. Demgemäß wäre es für die beteiligten Entwickler am FoodDeliverySystem übersichtlicher, welcher Teil der Anwendung zu welchem Bounded Context gehört und an welcher Stelle neue Teile der Anwendung ergänzt werden müssen, sowie in wessen Zuständigkeitsbereich die Implementierung fällt. In dieser Arbeit besteht zusätzlich das Interesse zu analysieren, wie sich der Aspekt einer objektorientierten Programmiersprache wie Java auf einen Anwendungsfall mit Fokus auf die Kombination von DDD und FaaS auswirkt.

5.1.2 Wahl des Frameworks

Die Wahl des FaaS Frameworks ist eine Entscheidung, welche den weiteren Projektverlauf wesentlich beeinflusst. Je nachdem, welcher Anwendungsfall vorliegt, stellt ein anderes Framework ggf. eine geeignetere Wahl für das Projekt dar. Dementsprechend müssen zunächst die in Frage kommenden FaaS Frameworks analysiert werden, um die beste Entscheidung für das zugrundeliegende Projekt zu treffen [6, 51].
Wie bereits in Kapitel 2.2.2 genannt, existieren zahlreiche FaaS-Anbieter. Im Zuge dieser Arbeit stellte sich die Wahl zwischen den drei kostenlosen Open Source Frameworks *Fission*, *OpenFaaS* und *Kubeless*.
Fission ist ein Open Source Projekt und wird in einem Kubernetes Cluster betrieben [18]. Für Fission haben sich unterschiedliche Programmiersprachen etabliert, zu diesen gehören u.a. NodeJS, Go und Java [19]. Bei der Verwendung von Fission wird ein Pool von Containern genutzt, auf welchen die Functions ausgeführt werden können. Somit kann zum Ausführen einer Function diese in einen beliebigen, laufenden Container geladen und dort weiterverwendet werden. Fission bietet hierdurch einen Performancevorteil, welcher auch bei den etablierten Frameworks zu den wichtigsten Gründen für die Umstellung auf FaaS zählt. In Abbildung 5.1 ist dieser Pool im `fission-function` *Namespace* von Kubernetes zu sehen. Ein Namespace in Kubernetes wird durch ein virtuelles Cluster definiert, welches innerhalb eines physischen Clusters unterstützt wird [46].

```
$ kubectl get deploy -n fission-function
NAME                        READY   UP-TO-DATE   AVAILABLE   AGE
poolmgr-java-default-1346   3/3     3            3           3h18m
```

Abbildung 5.1: Poolmanager für das Java Environment i.A.a. [23].

In Abbildung 5.2 ist zu erkennen, dass ein Pool von Pods für die hier angelegte Java Umgebung existiert.

```
$ kubectl get po -n fission-function
NAME                                         READY   STATUS    RESTARTS   AGE
poolmgr-java-default-1346-56fc8bf68f-5gfhb   2/2     Running   0          3h18m
poolmgr-java-default-1346-56fc8bf68f-n7wk6   2/2     Running   0          3h18m
poolmgr-java-default-1346-56fc8bf68f-ttmg6   2/2     Running   0          3h18m
```

Abbildung 5.2: Pool von Pods für das Java Environment in Fission i.A.a. [23].

Wie bereits aus den Beispielen ersichtlich wurde, wird für Fission ein Kubernetes Cluster benötigt. Existiert lediglich der Bedarf nach einem simplen Cluster, lässt sich dieses u.a. lokal über Docker Desktop erstellen, worauf an dieser Stelle zurückgegriffen wurde. Hiermit wird ein single-node Kubernetes Cluster aufgesetzt [16].
Mithilfe eines Docker-Images für die gewünschte Programmiersprache lässt sich nun ein dazu passendes, auf die Programmiersprache zugeschnittenes *Environment* in Fission aufsetzen [19], worauf in Abschnitt 5.1.3 eingegangen wird. Durch das Integrieren von Fission in einem Kubernetes Cluster ist es möglich, genauere Einblicke in die Anwendung der implementierten Functions zu erlangen, da so wie bereits in Abbildung 5.1 und 5.2 zu sehen war, Details auch innerhalb des lokalen Kubernetes Clusters beobachtet werden können. Dadurch ergibt sich ein Vorteil gegenüber Frameworks, welche weniger Einblicke in die inneren Strukturen gewähren.
OpenFaaS ist ein Open Source Framework welches ebenfalls in einem Kubernetes Cluster betrieben wird und viele Programmiersprachen, wie bspw. Java, Python und Go unterstützt [38]. Für die Functions werden eigene Images gebaut und in eine Container Registry gepusht. Zusätzlich können die Functions in ein Cluster deployed werden [37].
Kubeless ist das dritte Open Source Projekt, das hier vorgestellt wird. Wie bereits bei den zwei anderen vorgestellten Frameworks, handelt es sich auch hier um ein Framework, das in einem Kubernetes Cluster bereitgestellt wird und viele gängige Programmiersprachen unterstützt, neben Python und Node.js auch die für das Anwendungsbeispiel genutzte Programmiersprache Java [30].
Ein wesentlicher Nachteil von Kubeless ist, dass dort im Gegensatz zu Fission, kein Poolmanager verwendet wird. Dies bedeutet für den Anwendungsfall in der Praxis, dass wenn

ein Node im Kubernetes Cluster abstürzt, bei Fission dank dem Poolmanager lediglich auf einen anderen Pod zurückgegriffen werden kann, wohingegen bei Kubeless erst ein neuer Pod erstellt werden muss [23]. Zu diesem Thema bestand bereits 2017 ein Issue bei GitHub, welcher aufgrund der vorangeschrittenen Zeit nach einem Jahr geschlossen wurde [3]. Dabei wurde allerdings diese Auffälligkeit nicht behoben, obwohl gerade der Aspekt der schnellen Verfügbarkeit bei FaaS von hoher Relevanz ist und oftmals überhaupt ein Grund für dessen Anwendung.
Beim Aufrufen der Deployments mittels `kubectl get deploy` werden in Abbildung 5.3 drei Test-Functions sichtbar, welche hier beispielhaft angelegt wurden.

```
$ kubectl get deploy
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
hello         1/1     1            1           9m46s
hellothree    1/1     1            1           5m7s
hellotwo      1/1     1            1           5m27s
```

Abbildung 5.3: Test-Functions beim Aufrufen der Deployments für Kubeless i.A.a. [23].

Werden anschließend mit dem Befehl `kubectl get po` die Pods für die Functions aufgelistet, zeigen sich für die einzelnen Test-Functions in Kubeless in Abbildung 5.4 jeweils ein Pod pro Function.

```
$ kubectl get po
NAME                          READY   STATUS    RESTARTS   AGE
hello-56d6494494-vdpnx        1/1     Running   0          9m52s
hellothree-8c7f77d66-l67lk    1/1     Running   0          5m13s
hellotwo-79c6f754f-c7ccm      1/1     Running   0          5m33s
```

Abbildung 5.4: Ein Pod pro Function bei Kubeless i.A.a. [23].

Entscheidung für ein Framework. Wie gezeigt wurde, lassen sich alle drei Frameworks in Kombination mit gängigen Programmiersprachen verwenden, sodass in Hinsicht auf diesen Aspekt kein Anbieter ausgeschlossen werden konnte.
Die Installation und Anwendung von OpenFaaS hat sich als weniger Intuitiv als bei den anderen hier betrachteten selbst gehosteten Frameworks erwiesen. Das Erstellen und Deployen einer Function mittels der `faas-cli` in Kombination mit Java erwies sich als deutlich aufwendiger im Vergleich zu den anderen Varianten. Zwar bietet die Dokumentation viele Informationen, jedoch ist sie gerade für Neueinsteiger in dem Bereich weniger geeignet, als bspw. die Dokumentation von Fission, da diese insbesondere für den Einstieg

in das Framework verständlicher beschrieben ist. Aus diesem Grund wurde OpenFaaS als mögliche Alternative ausgeschlossen.
Das Framework Kubeless ist ebenfalls wie Fission sehr leicht aufzusetzen und ist somit auch für Einsteiger auf dem FaaS Gebiet gut geeignet. Aufgrund der Tatsache, dass bei Kubeless wie in den Abbildungen 5.3 und 5.4 zu sehen war kein Poolmanager existiert und diese Gegebenheit zum Zeitpunkt der Erstellung dieser Arbeit nicht geändert wurde, wurde Kubeless als mögliches Framework im Rahmen der Implementierung für diese Arbeit ausgeschlossen.
Insgesamt gehen die Meinungen bei der Einschätzung der Usability der drei Frameworks stark auseinander [39, 51, 23]. Dies deutet ebenfalls darauf hin, dass die Wahl des passenden FaaS Frameworks stark vom Anwendungsfall abhängig ist. Je nachdem, welche Gegebenheiten vorliegen, erweist sich ggf. eines der Frameworks als nützlicher oder kompatibler zur jeweiligen Situation als andere. Dementsprechend fällt die Beurteilung der Entwickler sehr subjektiv aus und die Entscheidung sollte nicht nur anhand von Bewertungen anderer Programmierer erfolgen.
Zusammenfassend hat sich **Fission** als das geeignetste selbst gehostete Framework für diese zugrundeliegende Arbeit erwiesen.

5.1.3 Funktionsweise von Fission

In Fission wird zwischen Functions, Environments und Triggern unterschieden. Bei den *Environments* (*dt.: Umgebungen*) handelt es sich um verschiedene Umgebungen, welche auf die jeweils gewünschte Programmiersprache zugeschnitten sind. Im Fall dieser Arbeit handelt es sich entsprechend um eine Java Umgebung. Mithilfe dieser können die Functions ausgeführt werden. Insgesamt lassen sich die von Fission bereitgestellten Environments in ihrer vorgegebenen Form verwenden, jedoch besteht die Option, die Environments zu verändern, auf die gewünschten Funktionalitäten zu erweitern oder sogar ganz eigene Umgebungen zu kreieren.[17]

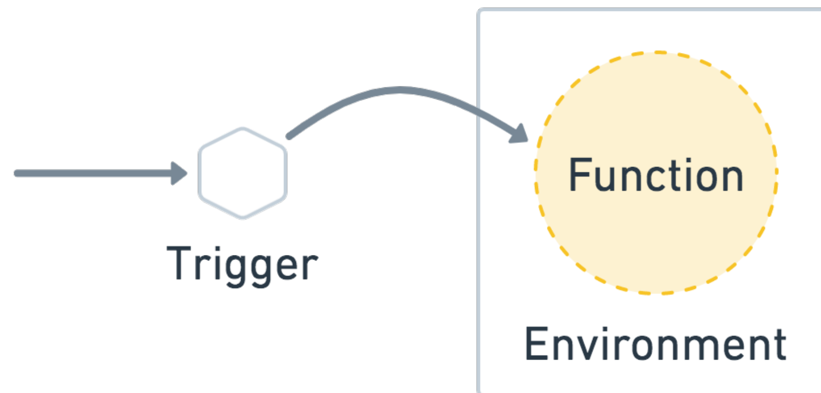

Abbildung 5.5: Fission Konzepte [17].

Wie bereits erwähnt, lassen sich Functions durch Trigger anstoßen, damit sie ausgeführt werden. Fission unterstützt hierbei HTTP-Trigger, Timer-Trigger, MQ- Trigger und Kubernetes Watch Trigger [17]. Im Zuge dieser Arbeit wird der Fokus auf MQ-Triggern liegen, jedoch wird die Anwendung von HTTP-Triggern mit Fission ebenfalls kurz aufgegriffen.
Insgesamt lässt sich für die wesentlichen Bestandteile von Fission, auf welche in dieser Arbeit eingegangen wird, zusammenfassen, dass die definierten Trigger das Ausführen einer Function in einem bestimmten Environment anstoßen.
Neben den Bestandteilen innerhalb von Fission ist ebenfalls zu erwähnen, dass für das Java Environment in Fission das Spring Boot Framework verwendet wird [19]. Dadurch kann zunächst der Anschein erweckt werden, dass das Deployen der Functions wenig performant wäre. Allerdings wird Spring Boot nicht bei jedem Ausführen einer Function erneut hochgefahren, sondern läuft dauerhaft im Pool, wie in dem Auszug aus den Logs in Abbildung 5.6 zu sehen ist. Aus diesem Grund besteht in dieser Hinsicht kein Nachteil bei der Verwendung von Fission.

```
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8888 (http) with context path ''
io.fission.Server                        : Started Server in 3.022 seconds (JVM running for 4.189)
io.fission.HelloWorld                    : Hello World!
io.fission.HelloWorldAgain               : Hello World from another function!
io.fission.HelloWorld                    : Hello World!
io.fission.HelloWorldAgain               : Hello World from another function!
```

Abbildung 5.6: Bei wiederholtem Aufrufen einer Function wird Spring Boot nicht erneut gestartet.

5.2 Implementierung des Fallbeispiels mit Fission

Anschließend an die getroffenen Voraussetzungen für eine Implementierung, folgt die Implementierung des Beispiels unter den zuvor definierten Gegebenheiten.
Für die Implementierung ist zunächst die Entscheidung zu treffen, wie der Aufbau der Projektstruktur erfolgen soll. Anhand der Context Map aus Kapitel 3 ist es möglich eine passende zugrundeliegende Projektstruktur zu konzipieren. Am geeignetsten stellt sich für das Fallbeispiel eine Aufteilung hinsichtlich der Bounded Contexts heraus. Die Java packages können nach dieser Logik angeordnet werden und die Überführung in Quellcode der zugehörigen Klassen und Methoden erfolgt analog zu ihrem Kontext im jeweiligen package. Da es sich hier um ein kleines Fallbeispiel handelt, ist die Abbildung von Bounded Contexts auf Java packages übersichtlich, allerdings ist es in größeren Projekten bspw. möglich, die Bounded Contexts als eigenständige Module und somit unabhängig auszuführende Artefakte zu implementieren [35, S. 80].
Zu beachten ist, dass die Kommunikation sowohl innerhalb als auch zwischen den Bounded Contexts gemäß DDD erfolgt, also bspw. innerhalb eines Aggregates lediglich die Aggregate Root nach außen hin kommuniziert (siehe Kapitel 2.1.2). Ein Beispiel für solch ein Aggregate innerhalb des Fallbeispiels bilden bspw. die `CustomerMealOrder`, das `CustomerMeal` und der darin enthaltene `MealPrice`, welche zusammen als ein *CustomerMealOrder-Aggregate* implementiert werden (siehe Abbildung 5.7).
Innerhalb dieses Aggregates fungiert die `CustomerMealOrder` als Aggregate Root, wohingegen das `CustomerMeal` eine Entity darstellt und der `MealPrice` des `CustomerMeals` als Value Object implementiert wird. Soll etwas an einem Bestandteil des Aggregates geändert werden, muss dies über die Aggregate Root erfolgen [28]. So könnte bspw. der Kunde seine bestellte Mahlzeit lediglich über seine Essensbestellung verändern. Eine Änderung an einem Bestandteil innerhalb des Aggregates lässt sich mittels einer simplen Function implementieren, die durch ein eingehendes Event getriggert wird, mit welchem bei der `CustomerMealOrder` eine Änderung angekündigt wird.
Das Ausführen solch einer Function kann bspw. durch einen HTTP-Trigger erfolgen. So ist denkbar, dass bei dem Aufruf einer bestimmten URL die Function ausgeführt wird, mit welcher der Kunde seine Mahlzeit verändern möchte. Dies gibt er in seiner Essensbestellung, also der `CustomerMealOrder`, über eine Website an. Indem er die Änderung anschließend mit einem Button auf der Website bestätigt, wird der Kunde auf eine Folgeseite weitergeleitet und durch diese der HTTP-Trigger ausgelöst, welcher wiederum den Request an die Function weiterleitet.
Ein Vorgang dieser Art stellt für das FoodDeliverySystem durchaus eine Möglichkeit

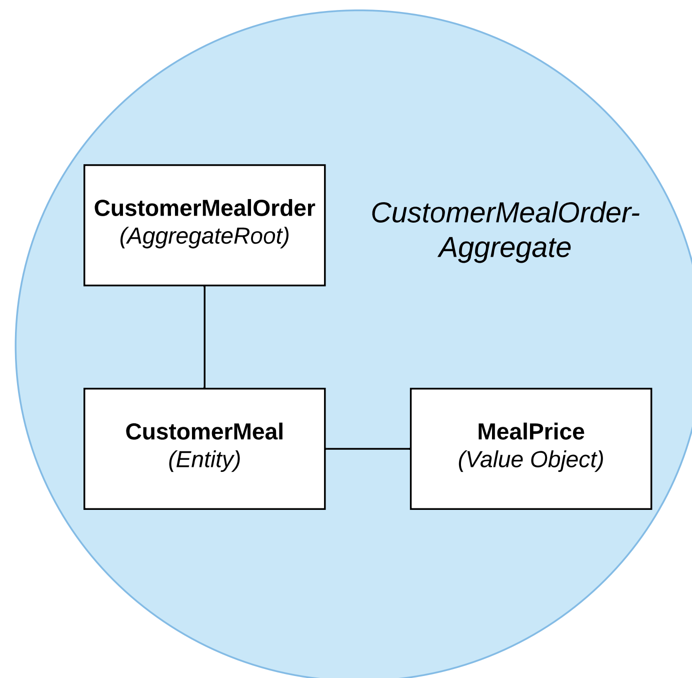

Abbildung 5.7: Das CustomerMealOrder-Aggregate mit seinen Bestandteilen.

dar. Allerdings sind insbesondere für dieses Fallbeispiel unter den Prinzipien von DDD für die eingehenden Events die MQ-Trigger deutlich vorteilhafter, da unter deren Anwendung eine klarere Struktur und stärkere Trennung der Zuständigkeiten innerhalb der verschiedenen Teile der Domäne stattfinden kann. Würde lediglich auf HTTP-Trigger zurückgegriffen werden, könnte unter Umständen schnell eine unübersichtliche Struktur erfolgen, da es nicht einen zentralen Vermittler insbesondere bei Events zwischen unterschiedlichen Bounded Contexts geben würde, sondern lediglich verschiedene URLs über welche die Functions getriggert werden. Im Gegensatz dazu stellt die MQ einen zentralen Bestandteil zwischen den Bounded Contexts dar und agiert als Brücke für die Domain Events. Dennoch werden im FoodDeliverySystem zusätzlich zu MQ-Triggern auch HTTP-Trigger zur Unterstützung mit eingebaut, wie im weiteren Verlauf der Arbeit ersichtlich wird.

Wie bereits im Sequenzdiagramm in Abbildung 4.4 deutlich wurde, ist es für die Implementierung des Fallbeispiels unabdingbar eine MQ einzubauen, welche eingehende Events koordiniert. Dabei muss es Methoden oder Functions geben, welche Events an die MQ schicken. Zusätzlich ist ein Trigger für die MQ erforderlich, welcher bei Ankunft eines

bestimmten Events die Ausführung der gewünschten Function einleitet.
Hierfür bietet Fission bereits integrierte Lösungen u.a. mit den Messaging-Systemen Kafka und NATS Streaming. Letzteres wurde für diese Arbeit ausgewählt, da sich NATS Streaming sehr einfach mit Fission anwenden lässt, weil dort alle benötigten Voraussetzungen für dessen Verwendung bereits bei der Installation von Fission integriert sind. Dahingegen bringt die Installation von Fission für Kafka lediglich die Kafka MQ Integration mit, jedoch muss Kafka zusätzlich noch installiert und so konfiguriert werden, dass es für das Kubernetes Cluster von Fission erreichbar ist [22, 20].

5.2.1 Domain Events mit NATS Streaming

NATS (*Neural Autonomic Transport System*) ist ein Projekt der Cloud Native Computing Foundation (CNCF). Hierbei handelt es sich um ein Open Source Messaging-System. NATS Streaming ist ein Datenstreaming System welches im Gegensatz zum herkömmlichen NATS u.a. bspw. die Fähigkeit hat, dass Nachrichten mit Sicherheit übermittelt worden sind und zudem persistiert werden. [36]
Mithilfe von Fission in Verbindung mit NATS Streaming wird dem Nutzer des Frameworks somit ermöglicht, Events zu einem bestimmten Anliegen, sogenannten *topics* über den `nats-streaming-server` an die MQ zu senden. Mittels eines vorher definierten MQ-Triggers, welcher auf ein spezielles topic lauscht, können dann im Voraus definierte Functions ausgelöst werden.[22] Dieser Zusammenhang ist in Abbildung 5.8 dargestellt.

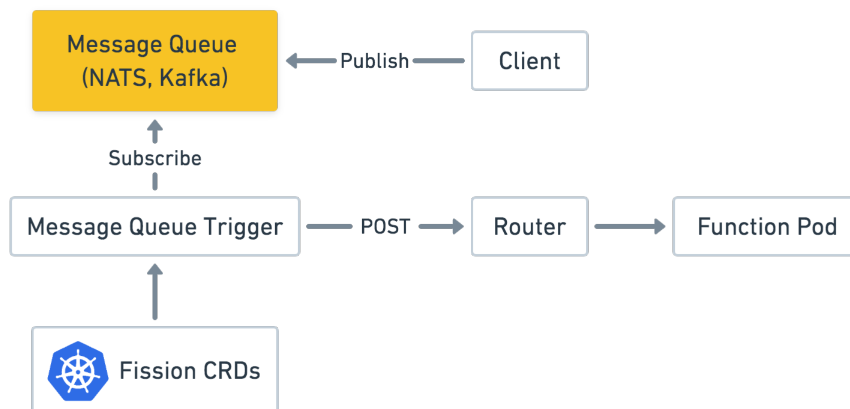

Abbildung 5.8: Der Client sendet ein Event an die NATS MQ und mittels des MQ-Triggers wird die Ausführung der angesprochenen Function bewirkt [21].

Damit dies möglich ist, muss die Anwendung mit demselben `nats-streaming-server` verbunden sein, wie Fission. Zur Integration in Java und der darin aufzubauenden Verbindung mit NATS Streaming musste die Dependency aus dem Listing 5.1 mit in das Maven-Projekt des FoodDeliverySystems mit eingebunden werden.

```
<dependency>
    <groupId>io.nats</groupId>
    <artifactId>java-nats-streaming</artifactId>
    <version>2.2.3</version>
</dependency>
```

Listing 5.1: Maven Dependency für NATS Streaming.

In dem Codeabschnitt 5.2 ist die Methode `publishEvent(String topic, String message)` aus der für die Fallstudie relevante Klasse `NATSEventPublisher.java` abgebildet. Die Methode bekommt als Parameter sowohl das topic als auch eine Nachricht

```java
public void publishEvent(String topic, String message) {
    Options o = new Options.Builder()
            .clusterId("fissionMQTrigger")
            .clientId("FoodDeliverySystem")
            .natsUrl("nats://defaultFissionAuthToken" +
                    "@nats-streaming.fission.svc.cluster.local:4222")
            .build();
    try (StreamingConnection nc = new StreamingConnectionFactory(o)
            .createConnection()) {
        nc.publish(topic, message
                .getBytes(StandardCharsets.UTF_8));
    }
    catch (IOException | InterruptedException | TimeoutException ) {
        e.printStackTrace();
    }
}
```

Listing 5.2: Methode zum Veröffentlichen eines Events unter Anwendung einer NATS `StreamingConnection` und unter Angabe eines topics.

mit übergeben, welche dem Event zugeordnet werden. So kann die Methode für alle beliebigen Events mit unterschiedlichen topics wiederverwendet werden und muss nicht für jedes topic angepasst werden. Innerhalb der Methode findet ein beispielhafter Verbindungsaufbau mit dem NATS Streaming Server statt. Dies erfolgt in den Zeilen 8-9 über die Klasse `StreamingConnectionFactory.java`. Für den Verbindungsaufbau werden

verschiedene Optionen benötigt, welche in Zeile 2 in der Variablen `o` gespeichert und anschließend beim Verbindungsaufbau mitgegeben werden. Damit hier die Verbindung zu NATS Streaming aus dem Kubernetes Cluster im Namespace *fission* genutzt wird, müssen die ClusterID und die korrekte NATS URL als Optionen übergeben werden.
Laut Kubernetes Konventionen setzt sich DNS Name für die NATS URL aus `servicename.namespacename.svc-cluster.local` zusammen [31]. Der Name des Services, `nats-streaming`, lässt sich bereits der Dokumentation von NATS Streaming in Verbindung mit Fission entnehmen [22]. Nachfolgend kann über diesen Namen wiederum der Name des Pods, der über den Service erreichbar ist, ermittelt werden. Dies geschieht mittels `kubectl -n fission get pod -l svc=nats-streaming`. Unter Verwendung des Namens des Pods ist es schließlich möglich, die fehlenden Informationen zum Verbinden mit dem NATS Streaming Server zu erhalten.
In Abbildung 5.9 ist ein Ausschnitt der Kommandozeile nach dem Aufruf des Befehls für die Beschreibung des Pods zu sehen. Hierhin finden sich die ClusterId `fissionMQTrigger` unter dem Punkt `Args: --cluster_id` sowie das ebenfalls benötigte Token zur Autorisierung `defaultFissionAuthToken` an der Stelle `Args: --auth`, ohne welches der Zugriff auf NATS-Streaming verweigert wird.

```
$ kubectl describe pods -n fission nats-streaming-74584c7d6d-66lnk
Name:         nats-streaming-74584c7d6d-66lnk
Namespace:    fission
Priority:     0
Node:         docker-desktop/192.168.65.3
Start Time:   Tue, 12 May 2020 09:54:23 +0200
Labels:       pod-template-hash=74584c7d6d
              svc=nats-streaming
Annotations:  <none>
Status:       Running
IP:           10.1.0.199
IPs:
  IP:           10.1.0.199
Controlled By:  ReplicaSet/nats-streaming-74584c7d6d
Containers:
  nats-streaming:
    Container ID:  docker://481134ca3e95e2e9b8b9a096d7b15ce8efe28ac3b5d0824aa468
e035614e4809
    Image:         nats-streaming
    Image ID:      docker-pullable://nats-streaming@sha256:4f980959f67ef9f39f6b2
515e7fa3109b06af5b091c2e171180b91e2ff8e9163
    Ports:         4222/TCP, 4223/TCP
    Host Ports:    0/TCP, 0/TCP
    Args:
      --cluster_id
      fissionMQTrigger
      --auth
      defaultFissionAuthToken
```

Abbildung 5.9: Informationen (Ausschnitt) über den Pod, der über den `nats-streaming` Service erreichbar ist.

Die Klasse `NATSEventPublisher.java`, in welcher die in dem Listing 5.2 aufgezeigte `publishEvent(String topic, String message)` Methode implementiert ist, ist für alle Bestandteile der Domain zugänglich, sodass mithilfe dieser die Kommunikation innerhalb des FoodDeliverySystems erfolgen kann.
Im Anschluss an eine erfolgreiche Verbindung mit NATS Streaming aus dem Kubernetes Cluster im *fission* Namespace können unter Anwendung dieser Methode Domain Events innerhalb und zwischen den Bounded Contexts an die MQ gesendet werden und die entsprechenden Functions werden passend zu den angesprochenen topics getriggert. Damit das Triggern durch Domain Events funktioniert, müssen MQ-Trigger für die Functions erstellt werden.
Als Beispiel wird der Anwendungsfall aus dem Sequenzdiagramm 4.4 aufgegriffen und für die Implementierung ein wenig abgewandelt. Zu Beginn des genannten Sequenzdiagramms löst der Kunde ein Event aus, welches an die MQ weitergegeben wird. Unter Anwendung des MQ-Triggers wird im Anschluss die `customer-order` Function aufgerufen. Dieser Teil wird so abgewandelt, dass das Ausführen der `customer-order` Function mittels eines HTTP-Triggers ermöglicht wird. Eine aktualisierte Version dieses Diagramms wird in Abbildung 5.10 dargestellt.

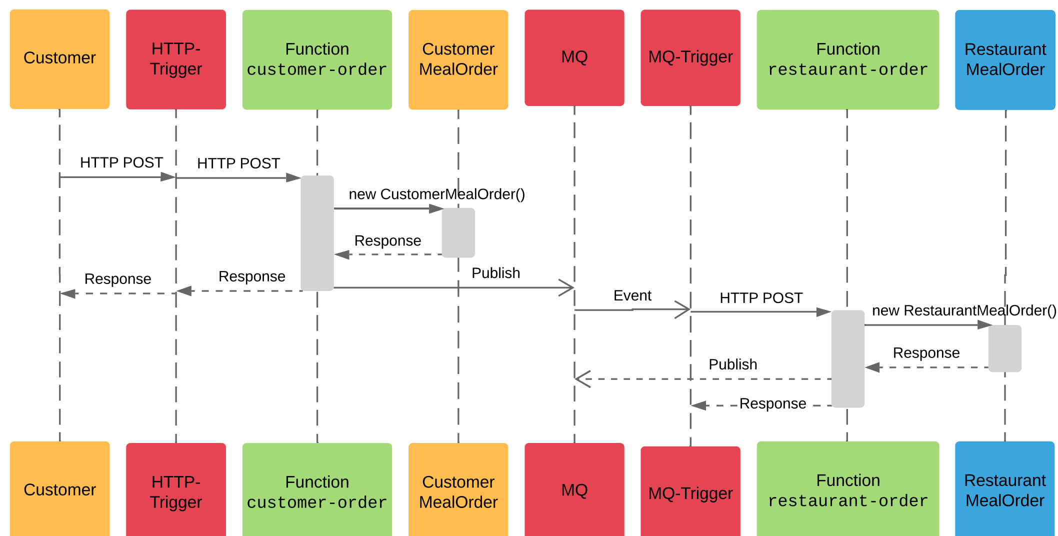

Abbildung 5.10: Angepasstes Sequenzdiagramm aus Abschnitt 4.2.1 mit HTTP-Triggern und MQ-Triggern für Domain Events *(orange: Customer-Context, rot: Trigger und MQ, grün: Functions, blau: RestaurantContext).*

Wie dem Diagramm zu entnehmen ist, wird zur Kommunikation innerhalb eines Bounded Contexts, in dem Fall dem CustomerContext, ein HTTP-Trigger verwendet und zur Kommunikation zwischen zwei Bounded Contexts, hier Customer- und RestaurantContext, wird auf einen MQ-Trigger zurückgegriffen. Dementsprechend wird die Anwendung eines HTTP-Triggers und des MQ-Triggers für unterschiedliche Domain Events kombiniert. Der Wechsel zwischen zwei Bounded Contexts verläuft mittels des verschickten Events an die MQ asynchron, weil die Function keine Antwort diesbezüglich erwartet. Der `Customer` bekommt somit die Response zurück ohne darauf warten zu müssen, dass die `RestaurantMealOrder` erstellt wurde.
Dieser Zusammenhang soll nun in Code überführt werden. Das Ziel ist, wie im Sequenzdiagramm beschrieben, am Ende zwei Functions aus verschiedenen Bounded Contexts auf unterschiedliche Art und Weisen zu triggern. Dabei sollen diese jedoch lediglich unter Zuhilfenahme von NATS Streaming miteinander kommunizieren, sodass die klare Trennung der Bounded Contexts erhalten bleibt.
In dem Codebeispiel 5.3 ist die Java Methode dargestellt, welche im weiteren Verlauf als Entrypoint (Einstiegspunkt für die Ausführung der Function) für die `customer-order` Function verwendet werden soll. Das Ziel ist hierbei, diese Methode mittels eines HTTP-Triggers unter Anwendung der HTTP-Methode POST zu deployen. Diesem POST wird ein JSON Objekt mitgegeben, welches die Bestellung des `Customer` repräsentiert.
Um dies mit Fission zu ermöglichen, muss das Interface `Function` implementiert werden und damit die Methode `call(RequestEntity req, Context context)`. Da der exemplarische Code über keine GUI verfügt, wird in Zeile 3 eine beispielhafte ID für den `Customer` übergeben. In einem realen Anwendungsfall würde diese Information auf Basis eines im System eingeloggten Users ermittelt werden.
Aus dem Body der RequestEntity wird in den Zeilen 6 bis 7 das o.g. JSON Objekt ermittelt und in der Variablen `jsonString` als `String` gespeichert. Mithilfe eines ObjectMappers wird anschließend in den Zeilen 10-11 aus dem String zu dem `JSONObject` die `CustomerMealOrder` gewonnen. Sollte dieser Vorgang fehlschlagen, wird in den Zeilen 13 bis 15 die `call` Methode beendet und eine Fehlermeldung als `ResponseEntity` zurückgegeben, in welcher steht, dass der HTTP POST, über welchen die Function getriggert wurde, keine valide `CustomerMealOrder` enthält.

```java
public ResponseEntity call(RequestEntity req, Context context) {
    log.info("Begin mealOrder...");
    int customerId = 22;
    ObjectMapper oM = new ObjectMapper();
    LinkedHashMap<String, Object> mealMap;
    mealMap = (LinkedHashMap<String, Object>) req.getBody();
    String jsonString = new JSONObject(mealMap).toString();
    CustomerMealOrder customerMealOrder;
    try {
        customerMealOrder =
                oM.readValue(jsonString, CustomerMealOrder.class);
    } catch (IOException e) {
        log.error("",e);
        return ResponseEntity.badRequest()
                .body("POST contains invalid CustomerMealOrder");
    }
    customerMealOrder.setId(4);
    customerMealOrder.setCustomerId(customerId);
    checkAvailableMeal(customerMealOrder);
    String mealOrderString =
            new JSONObject(customerMealOrder).toString();
    new NATSEventPublisher()
            .publishEvent("customerOrderCreated", mealOrderString);
    return ResponseEntity.ok("Customer ordered Meal successfully.");
}
```

Listing 5.3: `CreateCustomerOrder` Klasse als Entrypoint für die `customer-order` Function.

In den Zeilen 17 und 18 werden die benötigten IDs für die `CustomerMealOrder` manuell gesetzt.

Für gewöhnlich würde die `CustomerMealOrder` persistiert werden und infolgedessen diese inklusive einer automatisch generierten ID von der Datenbank zurückgegeben werden. Da es im Zuge der Implementierung mit dem Framework jedoch bezüglich der Datenbankoperationen Probleme gab (mehr dazu in Kapitel 6.2) wurde dies nicht umgesetzt und dieser Teil stattdessen exemplarisch gelöst.

Durch das Persistieren der `CustomerMealOrder` wird dem User auf Kundenseite des FoodDeliverySystems ermöglicht, seine Bestellhistorie einzusehen. Da die später angelegte `RestaurantMealOrder` zum RestaurantContext zugehörig ist, soll es dem `Customer` aufgrund der unterschiedlichen Bounded Contexts nicht möglich sein, diese einzusehen.

In Zeile 19 des Codebeispiels 5.3 wird zudem eine `checkAvailableMeal(CustomerMealOrder customerMealOrder)` Methode aufgeru-

fen. Mit dieser soll bei einer eingehenden Essensbestellung des Kunden geprüft werden, ob die bestellten Mahlzeiten alle jeweils verfügbar sind. Sollte dies nicht zutreffen, kann an dieser Stelle über eine mögliche Fehlerbehandlung nachgedacht werden. Eine Option wäre bspw. mittels einer weiteren Function eine error Nachricht mit dem topic `customerOrderError` an die MQ zu übergeben, wodurch eine Mail an dem Kunden verschickt wird, in welcher diesem mitgeteilt wird, dass seine Bestellung nicht erfolgreich war.
Als letztes wird schließlich die Methode des `NATSEventPublisher` aus dem Codebeispiel 5.2 aufgerufen, welcher als topic `customerOrderCreated` mitgegeben wird und als Nachricht der JSON String zu der `CustomerMealOrder`. Zu beachten sei hierbei, dass im CustomerContext kein Wissen darüber besteht, an welcher Stelle diese Information für welchen Kontext relevant ist. Da also nicht bekannt ist, welche Informationen evtl. in anderen Kontexten benötigt werden, die an diesem Event interessiert sind, wird die gesamte JSON Repräsentation der `CustomerMealOrder` mitgegeben.
Ein weiterer Grund hierfür ist, dass es sich bei der `CustomerMealOrder`, wie in Abbildung 5.7 dargestellt, um die Aggregate Root eines Aggregates handelt. Da Aggregates immer als ein Ganzes betrachtet werden sollen, ist dies ebenfalls ein Argument, aus Sichtweise des DDD, die vollständige Darstellung des Aggregates zu verschicken. Wie außerdem zu erkennen war, wurden die Bestandteile der `CustomerMealOrder`, wie bspw. das `Meal`, nicht etwa einzeln übergeben, sondern das Aggregate wurde auch schon bei seiner Erstellung als Ganzes betrachtet und direkt aus dem JSON Objekt in die `CustomerMealOrder` überführt. Die Benennung des topics zum Publishen des Events erfolgt außerdem im Stil von Domain Events im DDD, also folglich die Bezeichnung für ein bestimmtes Verb als Ereignis in Präteritum ausgedrückt [47, S. 289].
Im Zuge des `customerOrderCreated` topics soll ein MQ-Trigger auf dieses topic reagieren und das Ausführen einer weiteren Function auslösen, die `restaurant-order` Function. Da die Nachricht, welche mittels NATS Streaming in dem Codeausschnitt 5.3 übergeben wurde, die vollständige Repräsentation der `CustomerMealOrder` enthält und die beiden Methoden ähnliche Ziele haben, sieht die `call` Methode für den Entrypoint der `restaurant-order` Function in der Klasse `CreateRestaurantOrder` sehr ähnlich aus. Darin wird jedoch dem `ObjectMapper` ein Feature mitgegeben, welches ihm ermöglicht nicht fehlzuschlagen, wenn unbekannte Properties beim Umwandeln des JSON Strings vorhanden sind. Der Code hierzu ist in Beispiel 5.4 zu sehen.

```
ObjectMapper objectMapper = new ObjectMapper();
objectMapper.configure(DeserializationFeature
        .FAIL_ON_UNKNOWN_PROPERTIES, false);
```

Listing 5.4: Unbekannte Properties bei Verwenden des `ObjectMapper` ermöglichen.

Diese Konfiguration ist notwendig, da ein `RestaurantMeal` über andere Klassenvariablen verfügt als ein `CustomerMeal`. Letzteres beinhaltet bspw. einen Wahrheitswert `private boolean isAvailable`, welcher angibt, ob eine Mahlzeit vom `Customer` bestellt werden kann, oder nicht. Diese Property wird somit vom `ObjectMapper` lediglich ignoriert und der Code kann problemlos weiter ausgeführt werden.
Analog zur neu angelegten `CustomerMealOrder` müsste auch die `RestaurantMealOrder` persistiert werden, damit das Restaurant die Bestellhistorie all seiner Kunden einsehen kann. Dabei sind die IDs der beiden Essensbestellungen in den zwei Kontexten äquivalent zueinander. Genauso wie am Ende der `customer-order` Function ein NATS Streaming Event gepublished wurde, kann dies auch am Abschluss der `restaurant-order` Function geschehen. Dies wird auch bereits im Sequenzdiagramm 5.10 angedeutet. Hierfür würde ein Event mit dem topic `restaurantOrderCreated` an die MQ geschickt werden und alle interessierten Functions würden auf das Event horchen. Zu den angesprochenen Functions würde bspw. eine zur Bestandsminderung im selben Bounded Context gehören. Dementsprechend würde die MQ ebenfalls für Domain Events innerhalb eines Bounded Contexts genutzt werden.
Nach der Implementierung der Ausgangssituation wird die `customer-order` Function unter Verwendung von Fission erstellt. Dies erfolgt über die `fission-cli` mittels des Befehls in Listing 5.5.

```
fission fn create --name customer-order --deploy \
target/FoodDeliverySystem-1.0-SNAPSHOT-jar-with-dependencies.jar \
--env java \
--entrypoint de.dakowitz.customer.functions.CustomerMealOrder
```

Listing 5.5: Befehl für das Erstellen der `customer-order` Function für den Customer-Context.

Mit `--deploy` wird die gebaute jar mitgegeben, `--env` spezifiziert das Environment, in diesem Fall `java`. Mit `--entrypoint` wird der Einstiegspunkt für die Function geliefert, hier also die Java Klasse mit angegeben, in welcher die Funktionalität der Function umgesetzt wurde.

Nachdem die Function erstellt wurde, ist es anschließend möglich, für diese einen HTTP-Trigger zu definieren.

```
fission httptrigger create --method POST \
--url /customer/order --function customer-order
```

Listing 5.6: Befehl zur Erstellung eines HTTP-Triggers mit Fission.

Hierbei wird dem HTTP-Trigger als HTTP-Methode POST zum Auslösen der Function übergeben. Als URL wird `/customer/order` mitgegeben. Durch das `/customer` soll verdeutlicht werden, dass es sich hierbei um eine Function handelt, welche dem CustomerContext zuzuordnen ist. Als letzter Parameter wird die Function angegeben, welche mithilfe des Triggers deployed werden soll.
Die zweite zu erstellende Function ist die `restaurant-order` Function, welche die RestaurantMealOrder im RestaurantContext erstellt. Diese wird analog zum vorherigen Beispiel 5.5 erstellt. Anschließend muss ein MQ-Trigger für die `restaurant-order` Function erstellt werden, sodass diese sich anschließend unter Angabe eines bestimmten topics deployen lässt.

```
fission mqt create --name restaurant-order \
--function restaurant-order --topic customerOrderCreated
```

Listing 5.7: Befehl zur Erstellung eines MQ-Triggers mit Fission.

Dem Befehl zum Erstellen eines MQ-Triggers wird ein Name mitgegeben, welcher hier äquivalent zu dem Namen der zu triggernden Function ist. Als topic für den Trigger wird `customerOrderCreated` mitgegeben. Wird also ein Event an die MQ geschickt, welches das topic `customerOrderCreated` trägt, soll die Ausführung der hier zugewiesenen Function getriggert werden.
Im Anschluss an eine ausgeführte Function wird im Log eine Meldung darüber ausgegeben, ob das Erstellen einer CustomerMealOrder bzw. einer RestaurantMealOrder erfolgreich war. Dafür müssen die Logmeldungen für die Functions in allen Pods gesammelt betrachtet werden, sodass überprüft werden kann, ob die Functions wie erwartet ausgeführt worden sind.
In Abbildung 5.11 ist ein Teilausschnitt aus den Logmeldungen zu sehen. Ein HTTP POST wurde an den hier als Beispiel verwendeten Pfad einer URL `/customer/order` geschickt und diesem wurde ein exemplarisches JSON-Objekt der Bestellung des `Customer`

mit übergeben. Dies triggert die Ausführung der `customer-order` Function, sodass anhand der Logmeldung zu erkennen ist, dass der Entrypoint der Function aufgerufen wurde. Anschließend ist zu sehen, dass der `NATSEventPublisher` das Event mit der in JSON dargestellten Nachricht der `CustomerMealOrder` erhalten hat. Wie geplant ist auch im Log nachzuverfolgen, wie anschließend als Entrypoint der `restaurant-order` Function die `CreateRestaurantOrder` Klasse aufgerufen wurde, innerhalb welcher das übermittelte JSON erfolgreich verarbeitet wurde und somit eine `RestaurantMealOrder` im RestaurantContext erstellt werden konnte.

```
t].[localhost].[/]        : Initializing Spring FrameworkServlet 'dispatcherServlet'
DispatcherServlet         : FrameworkServlet 'dispatcherServlet': initialization started
DispatcherServlet         : FrameworkServlet 'dispatcherServlet': initialization completed in 52 ms
                          : Entrypoint class:de.dakowitz.customer.functions.CreateCustomerOrder
                          : Specialize call done in: 535 ms
CreateCustomerOrder       : Begin mealOrder...
CreateCustomerOrder       : CustomerMealOrder in CustomerContext successfully created!
Publisher                 : Start publishing Event...
Publisher                 : Event with message {"mealList":[{"meal":"pizza","available":true,"priceInCent":899,"id":1},{"m
gress
t].[localhost].[/]        : Initializing Spring FrameworkServlet 'dispatcherServlet'
DispatcherServlet         : FrameworkServlet 'dispatcherServlet': initialization started
DispatcherServlet         : FrameworkServlet 'dispatcherServlet': initialization completed in 82 ms
                          : Entrypoint class:de.dakowitz.restaurant.functions.CreateRestaurantOrder
                          : Specialize call done in: 526 ms
CreateRestaurantOrder     : CustomerRequest in Restaurant: {mealList=[{meal=pizza, available=true, priceInCent=899, id=1},

CreateRestaurantOrder     : RestaurantMealOrder(restaurantMealList=[RestaurantMeal(id=1, meal=pizza, priceInCent=899), Res

CreateRestaurantOrder     : RestaurantMealOrder in RestaurantContext successfully created!
```

Abbildung 5.11: In den Logmeldungen ist zu erkennen, dass die Kommunikation zwischen den Bounded Contexts über die MQ gelungen ist.

Auf diese Weise wurde also gezeigt, wie Domain Events innerhalb und zwischen den Bounded Context agieren können, ohne dass eine Function direkt zwei Bounded Contexts anspricht. Somit bleiben die Prinzipien von DDD trotz der Integration von FaaS erhalten.

6 Evaluierung und Bewertung

In diesem Kapitel erfolgt eine Auswertung der gewonnen Erkenntnisse. Dabei wird zwischen zwei verschiedenen Perspektiven unterschieden. Auf der einen Seite erfolgt eine Bewertung des gewählten Entwurfs und daraus folgenden Architektur und Implementierung. Auf der anderen Seite wird das gewählte Framework in Anwendung auf die zugrundeliegende Fallstudie beurteilt.

6.1 Bewertung und Evaluierung des Entwurfs und der Implementierung

Im Entwurf wurden zwei verschiedene Sequenzdiagramme dargestellt. Der Fokus bei diesen lag auf ihrer Kompatibilität mit den Prinzipien des DDD unter Einbezug von FaaS mittels Functions. Das Sequenzdiagramm 4.4 wurde dabei als optimale Lösung ausgewählt, da dort die Trennung zwischen den Bounded Contexts erhalten bleibt.
Bezüglich der Projektstruktur wurde der Ansatz gewählt, diese nach Bounded Contexts aufzugliedern. Auf den Anwendungsfall in der Praxis übertragen führt das ebenfalls zu einer Teamorganisation innerhalb des Projekts mit einer Aufteilung für ein Team je Bounded Context. Auf diese Weise wird bei der Implementierung ein stärkerer Fokus auf die Trennung der Zuständigkeiten gelegt. Zusätzlich ist insbesondere diese Aufteilung für die Organisation eines Projekts mit DDD von Vorteil, da so innerhalb der Teams von der Ubiquitous Language Gebrauch gemacht werden kann.
Für die Einbringung von FaaS bedeutet dies, dass Domain Events zwischen den fachlich zusammenhängenden Functions in unterschiedlichen Bounded Contexts zur Kommunikation verwendet werden. Die Domain Events werden durch Functions ausgelöst und über eine zentrale MQ weiter zum anderen Bounded Context gesendet. Alternativ dazu wäre eine Aufteilung der Implementierungsstruktur sowie der daraus folgenden Teamaufteilung nach fachlichem Zusammenhang möglich. Hierbei würde jedoch im Zuge der Implementierung ein Team an mehreren Bounded Contexts arbeiten bzw. die Grenzen

der Kontexte könnten viel leichter vermischt werden. Speziell bei feingranular definierten Functions könnten Unstimmigkeiten darüber entstehen, welche Functions sich direkt ansprechen dürfen und auf welche Bereiche eine Function aufgrund der Kontextabgrenzungen nicht zugreifen darf. Daher wurde der zweite Ansatz nicht gewählt.
Für die Implementierung wurde das entworfene Sequenzdiagramm in Abbildung 4.4 wie in Abbildung 5.10 zu sehen abgewandelt, um sowohl den Einsatz von HTTP-Triggern als auch jenen von MQ-Triggern zu erforschen.
Im Verlauf der Implementierung erwies sich der Lösungsansatz als durchaus zufriedenstellend. So war es möglich die im Rahmen von Kapitel 5 erstellte Function `customer-order` und die darin enthaltenen Zusammenhänge im CustomerContext für das Fallbeispiel auf die vorher entworfene Art und Weise umzusetzen. Dieser Zusammenhang konnte zudem erfolgreich getestet werden.
Das entworfene Beispiel ließe sich auch auf die Praxis mit verschiedenen Entwicklerteams je Bounded Context übertragen. So könnte die oben genannte Funktionalität von einem Entwicklerteam übernommen werden. Da als Brücke zwischen den Bounded Contexts die MQ fungiert, können die Entwickler ihr Event unter Angabe eines spezifischen topics bequem an diese schicken. Zwischen den Teams muss lediglich eine Abstimmung über die topics erfolgen. Weiterführend ist es dann für das Team, welches sich um den Restaurant-Context kümmert, möglich, eine Function zu kreieren, welche auf das topic des Customer-Contexts reagiert. Nachfolgend werden im CustomerContext die `CustomerMealOrder` erstellt und die daraus resultierenden Aufgaben für den RestaurantContext ausgelöst. Die Verwendung eine Ubiquitous Language ist auf diese Weise reibungslos möglich und unterstützt somit zusätzlich die Kommunikation innerhalb der Teams. Auch Aggregates können in Zusammenhang mit Functions als Ganzes gehandhabt werden, wobei stets beachtet werden muss, welcher Teil dieser Function als Aggregate Root dient. Insgesamt hat sich für die gesamte Kommunikation innerhalb des FoodDeliverySystems eine MQ als erfolgsversprechend herausgestellt. Die Anwendbarkeit dieser sollte somit in Projekten mit Kombination von DDD und FaaS auf jeden Fall überprüft werden, jedoch sei stets zu beachten, dass der Erfolg immer von der zugrundeliegenden Domain abhängen.
Zusammenfassend ist unter Anbetracht der Fallstudie die Anwendung der MQ u.a. als Kommunikationsmittel zwischen den Bounded Contexts gelungen und die Idee aus dem Entwurf konnte erfolgreich in der Implementierung umgesetzt werden. Da neben den MQ-Triggern jedoch auch andere Arten von Triggern für Functions existieren, könnte in anderen Fällen auf diese zurückgegriffen werden. Insgesamt fällt im Rahmen dieser Arbeit jedoch die Bewertung bezüglich der Anwendbarkeit von MQ-Triggern für Domain Events, welche mithilfe von Functions gesteuert werden, positiv aus.

6.2 Bewertung und Evaluierung des gewählten Frameworks für DDD in Kombination mit FaaS

Die Implementierung im Rahmen dieser Arbeit hat zudem herausgestellt, dass es noch durchaus Schwierigkeiten mit den selbst gehostetes FaaS-Frameworks gibt.
Das für die Implementierung zugrunde liegende Framework Fission verfügt über eine umfangreiche Dokumentation mit Anwendungsbeispielen, welche die Anwendung dieses Frameworks besonders für Einsteiger in dem Bereich FaaS attraktiv macht. Außerdem gab es auch bei dem Aufsetzen des Frameworks keine Schwierigkeiten und die Installation aller zum Betrieb notwendigen Komponenten verläuft reibungslos und schnell. Zudem ist eine problemfreie Installation auf allen gängigen Betriebssystemen möglich, wodurch ein breiteres Spektrum an Anwendern angesprochen wird [18]. In der Dokumentation sind nicht nur Beispiele zum Nutzen der `fission-cli` vorhanden, sondern auch für die unterschiedlichen Environments, wie bspw. das `jvm-env` für Java [19]. Zusätzlich werden dort die Konzepte und Funktionsweisen innerhalb von Fission nahegelegt, wodurch es möglich ist, ein klares Bild darüber zu erhalten, ob dieses Framework für den eigenen Anwendungsfall zu gebrauchen ist [17].
Im weiteren Verlauf der Implementierung und beim Testen der geschriebenen Functions wurden jedoch einige Probleme ersichtlich. Eines davon ist, dass die `fission-cli` des Öfteren abstürzt. Ferner machte sich die Tatsache bemerkbar, dass es nicht möglich war, mit gewissen Java-Interfaces wie z.B. `JpaRepository` oder `CrudRepository` zu arbeiten. So trat z.B. bei dem Testen einer beliebigen Function, bspw: `fission fn test --name customer-order` sobald an einer willkürlichen Stelle im Projekt das `JpaRepository` mit einbezogen war die in Abbildung 6.1 aufgezeigt Exception auf.
Beim Deployen der Functions kam es zu unterschiedlichen Exceptions. Insgesamt erwies sich die Arbeit mit unterschiedlichen Java-Bibliotheken als schwierig und es kam gehäuft zu Konflikten, je mehr Abhängigkeiten hinzugefügt wurden. In Bezug auf diese Komplikationen existierten auch Issues. Von diesen wurde bspw. ein Issue bezüglich doppelter Dependencies in Java [4] als gelöst markiert, jedoch ist das Problem damit nicht behoben worden, sondern ein Workaround für den Umgang mit diesen vorgeschlagen. Allerdings werden auch durch diese die Probleme mit dem `JpaRepository` nicht behoben.
Das Problem wurde zum Zeitpunkt der Erstellung dieser Arbeit weiterhin nicht gelöst und ein weitere Issue zu der Thematik ist weiterhin offen [2]. Diese Probleme erschweren den Usern des Frameworks mit Java insbesondere die Arbeit mit Datenbankoperationen. Insbesondere dies wäre aber u.a. für DDD interessant. Da es sich bei auf DDD basieren-

```
java.lang.ClassNotFoundException: org.springframework.data.jpa.repository.JpaRepository
        at java.net.URLClassLoader.findClass(URLClassLoader.java:382) ~[na:1.8.0_212]
        at java.lang.ClassLoader.loadClass(ClassLoader.java:424) ~[na:1.8.0_212]
        at java.net.FactoryURLClassLoader.loadClass(URLClassLoader.java:817) ~[na:1.8.0_212]
        at java.lang.ClassLoader.loadClass(ClassLoader.java:357) ~[na:1.8.0_212]
        at java.lang.ClassLoader.defineClass1(Native Method) ~[na:1.8.0_212]
        at java.lang.ClassLoader.defineClass(ClassLoader.java:763) ~[na:1.8.0_212]
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142) ~[na:1.8.0_212]
        at java.net.URLClassLoader.defineClass(URLClassLoader.java:468) ~[na:1.8.0_212]
        at java.net.URLClassLoader.access$100(URLClassLoader.java:74) ~[na:1.8.0_212]
        at java.net.URLClassLoader$1.run(URLClassLoader.java:369) ~[na:1.8.0_212]
        at java.net.URLClassLoader$1.run(URLClassLoader.java:363) ~[na:1.8.0_212]
        at java.security.AccessController.doPrivileged(Native Method) ~[na:1.8.0_212]
        at java.net.URLClassLoader.findClass(URLClassLoader.java:362) ~[na:1.8.0_212]
        at java.lang.ClassLoader.loadClass(ClassLoader.java:424) ~[na:1.8.0_212]
        at java.net.FactoryURLClassLoader.loadClass(URLClassLoader.java:817) ~[na:1.8.0_212]
        at java.lang.ClassLoader.loadClass(ClassLoader.java:357) ~[na:1.8.0_212]
        at io.fission.Server.specialize(Server.java:84) ~[classes!/:0.0.1-SNAPSHOT]
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.8.0_212]
```

Abbildung 6.1: `ClassNotFoundException` beim Ausführen einer Function am Beispiel der Anwendung des `JpaRepository`.

den Anwendungen meist um große Projekte handelt, ist somit das Framework Fission und das dort vorhandene Java Environment derzeit noch nicht ausgereift genug, um sich nur auf dieses zu verlassen. Gerade das Auftreten der Exceptions bei dem Deployen der Functions deutet darauf hin, dass zu viel Zeit auf das Organisieren der maven Dependencies verloren geht, was insbesondere bei zeitkritischen Projekten ein schwerwiegendes Problem darstellt.

Als positiv erwies sich jedoch die Anwendung von NATS Streaming. Die Einbindung des Dienstes und die damit Verbundene Erstellung von MQ-Triggern ist simpel gestaltet und funktionierte im Rahmen des Anwendungsbeispiels ohne Probleme. Die Erfahrungen, welche im Laufe der Arbeit mit Fission in Verbindung mit NATS Streaming gesammelt wurden, waren sehr zufriedenstellend. Somit ist die Anwendung von MQ mit dem Framework Fission empfehlenswert.

Schlussendlich konnten die Grundideen von FaaS mithilfe von Fission gut und simpel nachvollzogen werden und lokale Tests ließen sich ebenfalls ohne Schwierigkeiten durchführen. Somit ist die Verwendung von Fission für kleinere Systeme durchaus empfehlenswert. Problematisch ist das derzeit nicht allzu große Entwicklerteam, welches hinter dem Großteil der Entwicklung des Frameworks steht. Es wird den Anwendern zwar erleichtert, die Entwickler von Fission über Slack etc. zu kontaktieren, jedoch dauert es auch länger, bis Issues angegangen werden [51, 1]. Die genannten Probleme lassen sich allerdings im Verlauf der Zeit durch den Zuwachs an Entwicklern lösen. Wenn gegenwärtig der Wunsch besteht, DDD mit FaaS zu kombinieren und der Anwendungsfall keine zu großen Ausmaße annimmt, ist Fission ein empfehlenswertes Framework. Handelt es sich dagegen um ein sehr komplexes Projekt, sollte momentan noch auf eines der größeren und bereits etablierten Frameworks zurückgegriffen werden.

7 Fazit und Ausblick

Im Zuge der Arbeit wurden unter Zuhilfenahme eines Fallbeispiels die Kombination von DDD und FaaS untersucht. Dabei bildete die zugrundeliegende Domain ein FoodDeliverySystem.
Unter Anbetracht des Anwendungsfalls wurde eine Analyse, ein Entwurf und auf Basis derer Resultate eine Implementierung vorgenommen. Im Rahmen der Implementierung lag der Fokus auf der Kommunikation zwischen unterschiedlichen Bounded Contexts in der Domain und es wurde die Einbringung von Functions auf dieser Ebene untersucht. Hierbei wurde insbesondere dem Einsatz einer MQ für Domain Events Beachtung geschenkt und für das Triggern der Functions wurden entsprechend MQ-Trigger und zusätzlich HTTP-Trigger betrachtet.
Für die Implementierung des Anwendungsfalls wurde das FaaS Open Source Framework Fission und als Programmiersprache Java verwendet. Fission erwies sich bezüglich der Anwendung auf Domain Events mit Unterstützung von NATS Streaming als sehr nützlich, trotz einiger anderer Probleme, welche auf die geringe Größe des Projekts zurückzuführen waren und folglich voraussichtlich in der Zukunft bei einer Weiterführung des Frameworks behoben sein werden. Grundsätzlich hat sich unter den hier dargestellten Rahmenvoraussetzung herausgestellt, dass DDD und FaaS nicht im Widerspruch zueinanderstehen.

7.1 Quintessenz der Leitfrage

In Bezug auf die Leitfrage : *„Unter welchen Voraussetzungen lassen sich Domain-Driven Design und Function as a Service in einem Softwareprojekt verknüpfen und welche Hürden ergeben sich bei solch einer Kombination?“* hat sich herausgestellt, dass ein Softwareprojekt in dieser Art durchaus realisierbar ist. Die Hürde bilden die feingranularen Functions, welche den Bounded Contexts korrekt zugordnet werden müssen sowie die adäquate Kommunikation mittels Functions zwischen den Kontexten, ohne dass dabei gegen

die entsprechenden DDD Prinzipien verstoßen wird. Jedoch kann diesen Problemen auf Basis einer präzisen Analyse- und Entwurfsphase vorgebeugt werden. Einer Kombination von DDD und FaaS steht demnach unter Voraussetzung einer guten Planung und der Wahl eines für das System passenden Frameworks nichts im Weg.

7.2 Weitere Forschungsmöglichkeiten

In dieser Arbeit wurde die Kombination von DDD mit FaaS im Allgemeinen analysiert und bewertet. Jedoch existieren zahlreiche Abwandlungen und weitere Konzepte, welche sowohl in Zusammenhang mit DDD als auch mit FaaS von Interesse sind.
So ist denkbar, eine genauere Analyse der Kombination unter Anwendung von **CQRS** (*Command Query Responsibility Segregation*) vorzunehmen, bei welchem es um eine genaue Aufteilung von Schreib- und Lesezugriffen innerhalb eines Systems handelt. Bei CQRS werden ebenfalls wie im DDD fachliche Ereignisse behandelt [43]. Als weitere Forschungsmöglichkeit kann das Anwenden von CQRS in einer DDD Anwendung analysiert werden und FaaS zusätzlich unterstützend eingesetzt werden.
Zusätzlich könnte in der Entwurfsphase für DDD eine **Hexagonale Architektur** gewählt werden [47, S. 129], durch welche sich womöglich neue Art und Weisen zum Einbringen der Functions ergeben würden.
Als Programmiersprache wurde hier auf Java zurückgegriffen. Ein Vergleich zwischen verschiedenen Programmiersprachen und den daraus folgenden Auswirkungen auf die Kombination von DDD und FaaS wäre ebenfalls für dieses Themengebiet von Interesse.
Außerdem wurde im Rahmen dieser Arbeit lediglich das Framework Fission für FaaS aufgegriffen. Allerdings existieren, wie gezeigt wurde, zahlreiche weitere FaaS Frameworks, welche sowohl gehostet als auch selbst gehostet sind. Dementsprechend könnte die Implementierung mit dem Fokus auf einen **Vergleich unterschiedlicher FaaS Frameworks** vorgenommen werden und eine Evaluation erfolgen, welches Framework für eine Kombination mit DDD am geeignetsten wäre.

Literaturverzeichnis

[1] : *Fission Contributors.* – URL https://github.com/fission/fission/graphs/contributors. – Zugriffsdatum: 2020-06-20

[2] : *Handle duplicate dependencies gracefully in JVM environment.* – URL https://github.com/fission/fission/issues/842. – Zugriffsdatum: 2020-05-21

[3] : *Kubeless is creating one pod per function - is this normal for serverless?.* – URL https://github.com/kubeless/kubeless/issues/148. – Zugriffsdatum: 2020-06-02

[4] : *Specific dependencies cause Java function to fail.* – URL https://github.com/fission/fission/issues/841. – Zugriffsdatum: 2020-05-21

[5] Amazon Web Services, Inc. oder Tochterfirmen: *Entwickeln von Anwendungen mit serverlosen Architekturen.* – URL https://aws.amazon.com/de/lambda/serverless-architectures-learn-more/. – Zugriffsdatum: 2020-06-14

[6] Blommaerts, Bart: *Die Serverless Cloud – eine Einführung (Teil 2): Der FaaS-Anbieter-Vergleich.* – URL https://jaxenter.de/serverless-cloud-teil-2-48401. – Zugriffsdatum: 2020-06-18

[7] Bommer, Christoph ; Spindler, Markus ; Barr, Volkert: *Softwarewartung : Grundlagen, Management und Wartungstechniken.* dpunkt.verlag GmbH, 2008. – ISBN 9783864919664

[8] Cloud Native Computing Foundation: *CNCF Cloud Native Interactive Landscape.* – URL https://landscape.cncf.io/format=serverless. – Zugriffsdatum: 2019-12-03

[9] Cloudflare, Inc.: *Backend-as-a-Service.* – URL https://www.cloudflare.com/learning/serverless/glossary/backend-as-a-service-baas/. – Zugriffsdatum: 2020-06-14

[10] Cloudflare, Inc.: *Function as a Service.* – URL https://www.cloudflare.com/learning/serverless/glossary/function-as-a-service-faas/. – Zugriffsdatum: 2020-06-14

[11] Dunkel, Jürgen ; Holitschke, Andreas: *Softwarearchitektur für die Praxis.* Springer-Verlag Berlin Heidelberg, 2003. – ISBN 9783642555527

[12] Ernst, Hartmut ; Schmidt, Jochen ; Beneken, Gerd: *Grundkurs Informatik: Grundlagen und Konzepte für die erfolgreiche IT-Praxis – Eine umfassende, praxisorientierte Einführung.* Springer Vieweg, Wiesbaden, 2016. – ISBN 9783658146344

[13] Esposito, Dino ; Saltarello, Andrea: *Microsoft .NET - Architecting Applications for the Enterprise.* Microsoft Press, 2014. – ISBN 9780735685352

[14] Evans, Eric: *Domain-Driven Design/Tackling complexity in the heart of software.* Pearson Education, Inc., 2004. – ISBN 9780321125217

[15] Eyk, Erwin van: *Function Composition in a Serverless World [Video].* – URL https://blog.fission.io/kubecon-eu-18/. – Zugriffsdatum: 2020-05-28

[16] Fission Project: *Docker Desktop.* – URL https://docs.fission.io/docs/installation/docker-desktop/. – Zugriffsdatum: 2020-06-18

[17] Fission Project: *Documentation - Concepts.* – URL https://docs.fission.io/docs/concepts/. – Zugriffsdatum: 2020-03-01

[18] Fission Project: *Installing Fission.* – URL https://docs.fission.io/docs/installation/. – Zugriffsdatum: 2020-06-19

[19] Fission Project: *Java.* – URL https://docs.fission.io/docs/languages/java/. – Zugriffsdatum: 2020-06-04

[20] Fission Project: *Kafka.* – URL https://docs.fission.io/docs/triggers/message-queue-trigger/kafka/. – Zugriffsdatum: 2020-06-19

[21] Fission Project: *Message Queue Trigger - How Message Queue Trigger Works.* – URL https://docs.fission.io/docs/triggers/message-queue-trigger/. – Zugriffsdatum: 2020-06-04

[22] Fission Project: *NATS Streaming.* – URL https://docs.fission.io/docs/triggers/message-queue-trigger/nats-streaming/. – Zugriffsdatum: 2020-06-06

[23] Fonseka, Nate: *Kubeless vs Fission: The Kubernetes Serverless match up.* – URL https://medium.com/@natefonseka/kubeless-vs-fission-the-kubernetes-serverless-match-up-41f66611f54d. – Zugriffsdatum: 2020-06-02

[24] Goll, Joachim: *Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java.* Springer Vieweg, Wiesbaden, 2014. – ISBN 9783658055325

[25] Gurturk, Cagatay: *Building Serverless Architectures.* Packt Publishing Ltd., 2017. – ISBN 9781787129191

[26] Hoffmann, Dirk W.: *Software-Qualität.* Springer-Verlag Berlin Heidelberg, 2013. – ISBN 9783642356995

[27] Holmström, Petter: *Strategic Domain-Driven Design.* – URL https://vaadin.com/learn/tutorials/ddd/strategic_domain_driven_design. – Zugriffsdatum: 2019-10-05

[28] Holmström, Petter: *Tactical Domain-Driven Design.* – URL https://vaadin.com/learn/tutorials/ddd/tactical_domain_driven_design. – Zugriffsdatum: 2019-10-05

[29] IBM Cloud Education: *FaaS (Function-as-a-Service).* – URL https://www.ibm.com/cloud/learn/faas. – Zugriffsdatum: 2020-05-09

[30] Kubeless 2020: *Kubeless.* – URL https://kubeless.io/. – Zugriffsdatum: 2020-06-02

[31] Kubernetes Documentation: *DNS for Services and Pods.* – URL https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/#aaaaa-records. – Zugriffsdatum: 2020-06-09

[32] Lilienthal, Carola: *Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen.* dpunkt.verlag GmbH, 2020. – ISBN 9783960888963

[33] Luber, Stefan ; Karlstetter, Florian: *Was ist Function as a Service (FaaS)?.* – URL https://www.cloudcomputing-insider.de/was-ist-function-as-a-service-faas-a-758571/. – Zugriffsdatum: 2020-02-25

[34] Millet, Scott ; Tune, Nick: *Patterns, Principles, and Practices of Domain-Driven Design.* John Wiley & Sons, Inc., 2015. – ISBN 9781118714706

[35] Nair, Vijay: *Practical Domain-Driven Design in Enterprise Java*. Apress, Berkeley, CA, 2019. – ISBN 9781484245439

[36] NATS Documentation: *Introduction*. – URL https://docs.nats.io/nats-streaming-concepts/intro. – Zugriffsdatum: 2020-06-06

[37] OpenFaaS Ltd.: *Create new functions*. – URL https://docs.openfaas.com/cli/templates/. – Zugriffsdatum: 2020-06-02

[38] OpenFaaS Ltd.: *OpenFaaS*. – URL https://www.openfaas.com/. – Zugriffsdatum: 2020-06-02

[39] Pientka, Frank: *A (Very!) Quick Comparison of Kubernetes Serverless Frameworks*. – URL https://vshn.ch/blog/a-very-quick-comparison-of-kubernetes-serverless-frameworks/. – Zugriffsdatum: 2020-01-21

[40] Red Hat Limited: *Cloudnative Anwendungen - Was ist Function-as-a-Service (FaaS)?*. – URL https://www.redhat.com/de/topics/cloud-native-apps/what-is-faas. – Zugriffsdatum: 2020-05-09

[41] Roberts, Mike: *Serverless Architectures*. – URL https://martinfowler.com/articles/serverless.html. – Zugriffsdatum: 2020-05-24

[42] Schlosser, Hartmut: *Domain-driven Design im Fokus: „Ein Bounded Context ist kein Microservice!“*. – URL https://jaxenter.de/architektur/domain-driven-design-bounded-context-microservice-88986. – Zugriffsdatum: 2020-01-20

[43] Schlosser, Hartmut: *„DDD, Event-Sourcing und CQRS ergänzen sich ausgezeichnet“ – Interview mit Golo Roden*. – URL https://entwickler.de/online/web/ddd-event-sourcing-cqrs-579829343.html. – Zugriffsdatum: 2020-06-20

[44] Starke, Gernot: *Effektive Softwarearchitekturen: Ein praktischer Leitfaden*. Carl Hanser Verlag GmbH & Co. KG, 2015. – ISBN 9783446444065

[45] Statista 2020: *Die beliebtesten Programmiersprachen weltweit laut PYPL-Index im Juni 2020*. – URL https://de.statista.com/statistik/daten/studie/678732/umfrage/beliebteste-programmiersprachen-weltweit-laut-pypl-index/. – Zugriffsdatum: 2020-06-18

[46] THE KUBERNETES AUTHORS: *Namespaces.* – URL https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/. – Zugriffsdatum: 2020-06-18

[47] VERNON, Vaughn: *Implementing Domain-Driven Design.* Pearson Education, Inc., 2013. – ISBN 9780321834577

[48] VERNON, Vaughn: *Domain-Driven Design Distilled.* Pearson Education, Inc., 2016. – ISBN 9780134434421

[49] VIEWEG, Iris ; WERNER, Christian ; WAGNER, Klaus-P ; HÜTTL, Thomas ; BACKIN, Dieter: *Einführung Wirtschaftsinformatik: IT-Grundwissen für Studium und Praxis.* Gabler Verlag | Springer Fachmedien Wiesbaden GmbH 2012, 2012. – ISBN 9783834968562

[50] WASSON, Mike ; BOEGLIN, Adam: *Using tactical DDD to design microservices.* – URL https://docs.microsoft.com/bs-latn-ba/azure/architecture/microservices/model/tactical-ddd. – Zugriffsdatum: 2020-05-09

[51] WINDER, Phil: *A Comparison of Serverless Frameworks for Kubernetes: OpenFaas, OpenWhisk, Fission, Kubeless and more.* – URL https://winderresearch.com/a-comparison-of-serverless-frameworks-for-kubernetes-openfaas-openwhisk-fission-kubeless-and-more/. – Zugriffsdatum: 2020-06-18

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: ______________________________

Vorname: ______________________________

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Die Kombination von Domain-Driven Design und Function as a Service

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

____________	____________	______________________
Ort	Datum	Unterschrift im Original