Containervirtualisierung und Orchestrierung

PRAXISBERICHT T1000

für die Prüfung zum

Bachelor of Science

des Studiengangs Informatik

Studienrichtung Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Jakob Jonathan Heitzmann

25. September 2020

Matrikelnummer	9119328
Kurs	TINF19B4
Ausbildungsfirma	YellowMap AG, Karlsruhe
Betreuer	Markus Lind
Prof. Jörn Eisenbiegler	

Erklärung

(gemäß §5(3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. 9. 2015)
Ich versichere hiermit, dass ich meine Projektarbeit mit dem Thema: ”Containervirtualisierung und Orchestrierung” selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Glossar

API Application Programming Interface. 6

JSON JavaScript Object Notation. 15

Legacy-System Ein älteres System, das nicht mehr aktiv weiterentwickelt wird. 8

Microservices Architekturkonzept bei dem die Funktionalität durch viele unabhängige Dienste, die miteinander kommunizieren, gegeben ist. 10

Middleware Anwendungsneutrales Programm, das zur Kommunikation zwischen verschiedenen Programmen eingesetzt wird. 10

YAML YAML Ain't Markup Language. 20

1 Einführung

Ziel der Arbeit ist es, vorzuzeigen, was ich in den ersten beiden Semestern meines Studiums bei der YellowMap AG gelernt und getan habe. Dabei wird sich auf die Gesamtheit der Bemühungen begrenzt, die dazu galten, die bestehende Serverstruktur zu brechen und einen Prototyp zu konzeptionieren, mit wessen Vorbild man das bestehende System in ein verteiltes System wandeln könnte. Zu diesem Zweck musste ich mich mit verschiedenen Arten der Serverkommunikation und Containervirtualisierung beschäftigen.

1.1 Kontextuelle Betrachtung der Rahmenbedingungen

YellowMap[35] entwickelt und hostet in erster Linie Kartenanwendungen, bietet Entwicklern aber auch mit einer Java-Script-API die Möglichkeit, mit wenig Selbstaufwand selbst eine solche Anwendung zu entwickeln[30]. Dabei laufen die Dienste mit allen gängigen großen Kartenanbietern so wie Google, OSM, Bing und sogar Baidu. Exemplarische Lösungen, die YellowMap anbietet, sind zum Beispiel die FilialFinder[12], mit denen alle Filialen eines Klienten auf dessen Webseite angezeigt werden können. Die Anzeige erfolgt natürlich auf einer Karte. Gesucht wird für gewöhnlich mit einer Umkreissuche auf den Standort des Gerätes. Die gefundenen Filialen können auch neben der Karte auch auf einer Liste sortiert ausgegeben werden.

Im umrissenen Zeitraum habe ich viele Tätigkeiten besonders in den Bereichen Backend und Testing übernommen. Doch auch ein paar kleinere Aufgaben im Bereich Frontend fielen mir zu. Gearbeitet habe ich mit .NET Core, Gatling, HTML, CSS, JavaScript, Docker, Kubernetes und RabbitMQ. Darunter war die Entwicklung eines Dienstes, der dazu fähig ist, Termindaten von einer Datenbank zu ziehen und diese dann auf kleinen E-Paper-Bildschirmen anzuzeigen, was vollständig in C# realisiert wurde. Auch entwickelte ich zu Demozwecken Testwebseiten, die Funktionen der JavaScript API präsentieren sollten. Ein anderes Projekt war die Realisierung von Stresstests auf ein schon bestehendes Projekt mittels Gatling. Dafür musste ich mich in die Programmiersprache Scala und Maven einarbeiten.

Das Projekt, auf das ich in diesem Bericht näher eingehen möchte, war nun die Entwicklung eines Prototyps zur Containervirtualisierung und automatischen horizontalen Skalierung mittels Docker und Kubernetes, sowie die Entwicklung eines Prototyps für die Serverkommunikation mit Message Broker mittels RabbitMQ. Für diese Aufgabe wurden diese Technologien vorgegeben.

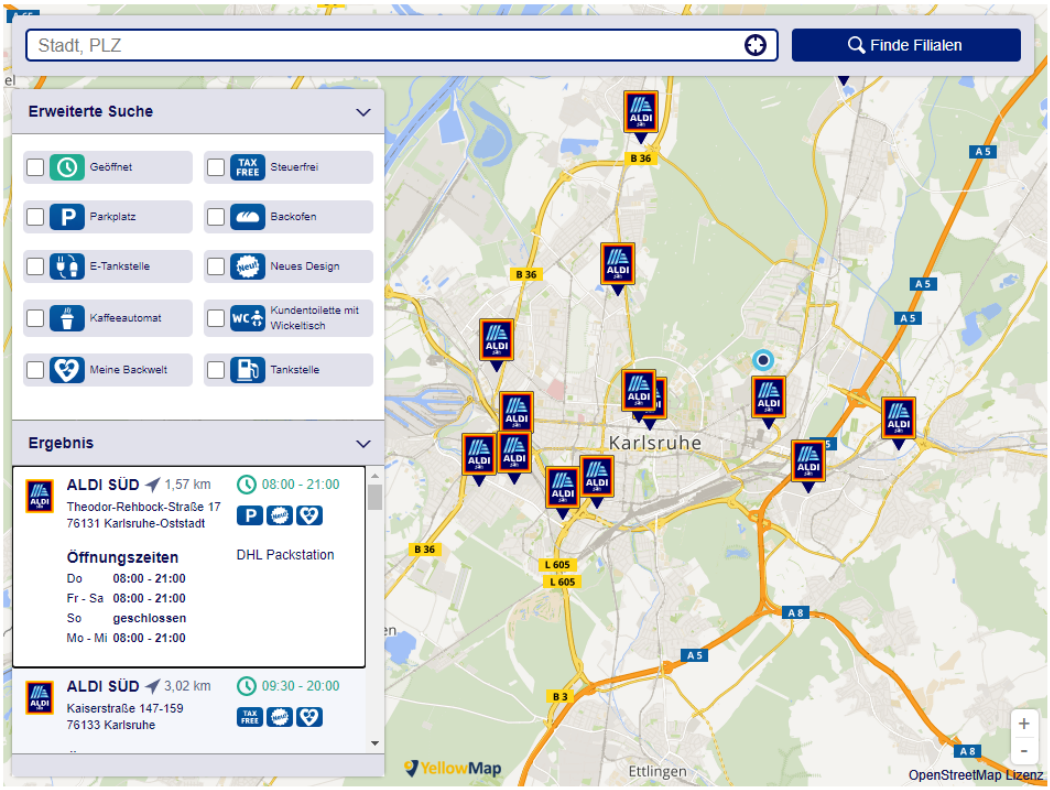

Abbildung 1.1: FilialFinder

1.2 Das bisherige Serverdesign

Das bestehende System wurde entworfen, damit viele Anwendungen unabhängig voneinander entwickelt werden können, ohne dass jede Funktionalität in jeder neuen Anwendung auch neu geschrieben werden müsste. Dazu greifen alle Projekte auf dieselbe Schnittstelle zu, diese bietet dann die einzelnen Funktionalitäten in einem einheitlichen Format. Diese Art der Kommunikation ermöglicht es, unabhängig von Art der Anwendung, die Anbindung an die internen Dienste sicherzustellen. Eine Anfrage eines Projekts auf diese Schnittstelle landet als Erstes am Proxy der *Core*, also zu Deutsch dem Kern, hier wird die Anfrage angenommen und auf erster Ebene authentifiziert. Diese Anfrage wird dann an den *System Call* weitergeleitet. Hier befindet sich der Partnerschalter. Er reichert die Daten der Anfrage mit den Daten zum gelieferten Partner aus der Datenbank an. Heißt er schaut zum Beispiel, von welchem Provider der Partner seine Daten beziehen möchte und trifft dem entsprechend Vorkehrungen, dass die Daten in den folgenden Aufrufen und Aktionen richtig weitergegeben und verarbeitet werden. Auch wird hier die auszuführende Aktion ermittelt und der dazugehörige Befehl in Form eines *Commands* oder einer *Command-Group*. Ein *Command* reichert die Daten nun weiter an und reicht dann an einen Provider spezifische *Command-Action* weiter, die die angefragte Aktion dann ausführt. In Abbildung 1.2 sehen wir einen Aufruf, der dann aber beispielshalber auf ein *Lagacy-System* zugreift, das wiederum selbst eine eigene

Command-Action besitzt, die die Anfrage dann letztendlich ausführt. Diese *Legacy-Systeme* werden aber in dieser Arbeit nicht weiter betrachtet, da diese nicht mehr aktiv weiterentwickelt werden.

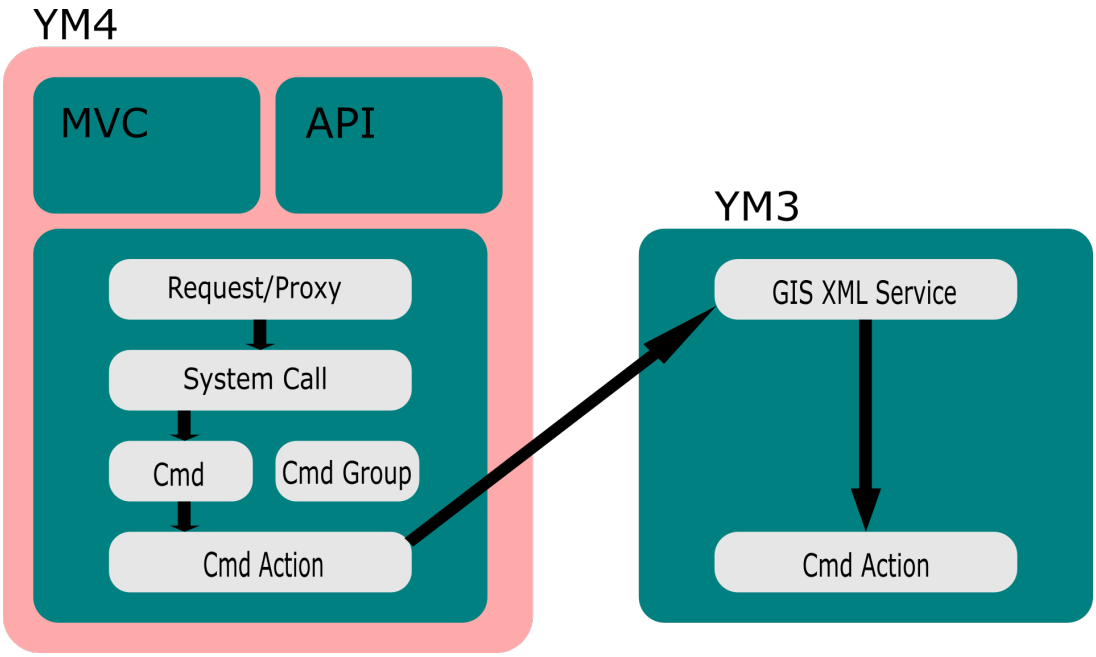

Abbildung 1.2: Beispielanfrage

Das System ist also kein Monolithisches, da es aus vielen Einzelprojekten besteht, die größtenteils unabhängig von einander agieren können, aber es ist ebenso kein verteiltes System, da sich trotz des modularen Designs alles auf demselben Server befindet.

1.3 Ziel der Arbeit

Beabsichtigt ist es, das eben beschriebene System nun in ein verteiltes System umzuwandeln. Ein verteiltes System ist ein System, dass aus einer Ansammlung von unabhängigen Komponenten besteht, die über ein Netzwerk miteinander kommunizieren, aber trotzdem nach außen als einzelnes System auftritt. Im Fall des Zielsystems soll dies mit einer zentralen API-Schnittstelle realisiert werden, die mögliche Anfragen dann an die verschiedenen Komponenten weiterleitet. So tritt es zwar als einzelnes System auf, aber kann komponentenweise sehr flexibel gehandhabt werden. So können einzelne Komponenten unabhängig vom Gesamtsystem einfach ausgetauscht werden. So wird uns auch ermöglicht, das System auf sehr spezifische Weise zu Skalieren.

Das Ziel meiner Arbeit liegt also darin, anhand der Entwicklung eines Prototyps die nötigen Grundlagen zu erarbeiten und mir das nötige Wissen anzueignen, das im Weiteren benötigt wird, um ein verteiltes System aufzusetzen und zu verwalten. Meine Arbeit legt den Grundstein für alle kommenden Projekte, die auf dieser

Plattform entwickelt werden sollen und auf weitere Untersuchungen, die sich notwendigerweise aus dieser Untersuchung ergeben.

Wichtig für diesen Prototypen sind Flexibilität, Unabhängigkeit und Robustheit.

Das System soll aus austauschbaren Einzelkomponenten bestehen, die dynamisch miteinander kommunizieren. Dies bezeichnet die Flexibilität des Systems und stellt sicher, dass Komponenten in laufenden Betrieb ausgetauscht werden können und auch, dass der Ausfall einer Einzelkomponente innerhalb des Systems keinen Totalausfall für das gesamte System bedeuten würde.

Ebenso soll sollen die Komponenten in ihren eigenen Umgebungen unabhängig von den Anderen lauffähig sein, sodass bei der Entwicklung dieser Abhängigkeiten minimiert werden. So sollen weniger Konflikte zwischen verschiedenen Projekten und ihren Abhängigkeiten gewährleistet werden, was wiederum die Flexibilität erhöht, da dies das Aufrüsten einzelner Komponenten innerhalb des Systems ermöglicht.

Weiterhin soll das System auch Robustheit beweisen, indem es bei dem Ausfall von Komponenten innerhalb des Systems auf diesen reagieren und diese wieder neu starten sollte. Also muss das System sich selbst nicht nur überwachen, sondern auch regenerieren können.

2 Serverkommunikation mit Message Broker

Zuerst begann ich den Prototypen für die Serverkommunikation mittels Message Broker zu konzeptionieren. Der Message Broker sollte eine neue interne Schnittstelle zwischen dem Serversystem und dem *Legacy-System* etablieren, da diese zuvor via normale *HTTP*-Aufrufe, die über das externe Netz gingen, realisiert wurde. Dies benötigte das Herauslösen eigenständiger Komponenten des *Legacy-Systems* und die Umwandlung dieser in sogenannte *Microservices*. Im Hinblick auf die späteren Änderungen, die noch an dem System vorgenommen werden sollten, sollte auch die Möglichkeit betrachtet werden, die Protokollierung von Systemdaten über den Message Broker auf eine Vielzahl von Empfängersystemen zu ermöglichen.

2.1 Funktion eines Message Brokers

Ein Message Broker ist ein System zur Nachrichtenübertragung und Nachrichtenverteilung. Dafür fungiert der Message Broker selbst als der zentrale Umschlagplatz, von dem aus Nachrichten empfangen und verteilt werden. Ebenso übersetzt er aktiv zwischen verschiedenen Protokollen[11]. Er stellt eine Art *Middleware* da, was bedeutet, dass er als neutrale Komponente im System nur für die Vermittlung von Information zwischen anderen Komponenten verantwortlich ist.

2.2 Anwendungsspezifische Betrachtung

In Hinblick auf die Containervirtualisierung ist das Prinzip eines Message Brokers mehr als nützlich. Ein Message Broker kann einen Nachrichten-Pool schaffen, den er kontinuierlich und gezielt mit Nachrichten versorgt, die dann von den letztendlichen Verbrauchern konsumiert werden können. Dadurch schafft man ein Netz von Geräten, die nur noch eine zentrale Adresse kennen müssen und dadurch dynamischer und unabhängiger voneinander agieren können.

In Abbildung 2.1 sehen wir, wie der Message Broker seine Funktion als *Middleware* zwischen einem *API-Gateway* und mehreren Diensten erfüllt.

2.3 RabbitMQ

RabbitMQ[26] ist ein Open-Source-Message-Broker, der das *Advanced-Message-Queuing-Protokoll (AMQP)* unterstützt[1] und sogar um einige nützliche Funktionalitäten erweitert[25]. Darunter ist eine grafische Managementoberfläche und Unterstützung für *Remote-Procedure-Calls (RPC)*. Dies sind beides Funktionen, die enorm praktisch für die vorgesehene Verwendung sind.

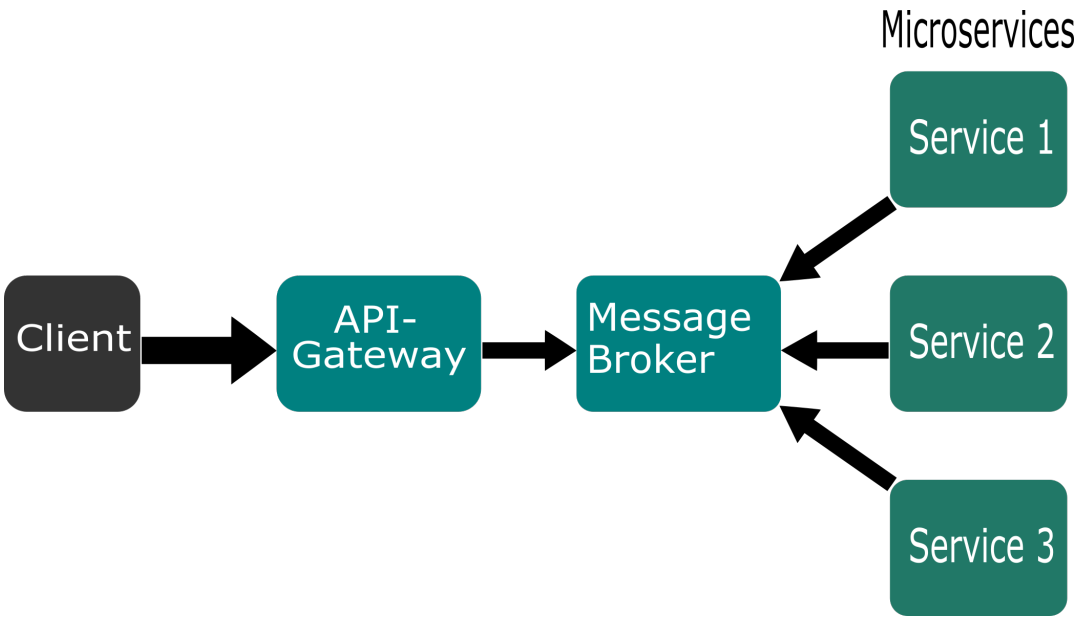

Abbildung 2.1: Serversystem mit Message Broker

Mittels grafischer Managementoberfläche wird die Fehlersuche immens einfacher gestaltet. Sie zeigt an, an welchen Endpunkten Nachrichten eingetroffen sind, wie viele es sind, ob ein Endpunkt gemeldete Konsumenten hat und vieles mehr. Mit ihr kann man ebenfalls sämtliche Konstrukte selbst erzeugen und die Nachrichten im allgemeinen Verwalten.

Ein *RPC* ermöglicht es uns, eine Funktion auf einem anderen Gerät auszuführen und ebenso dessen Wert zurückgeliefert zu bekommen. Was genau das ist, was wir in einem verteilten System wollen.

2.3.1 Grundlegende Funktionsweise

RabbitMQ stellt in erster Linie einen Serverdienst, eine Managementoberfläche und Nutzerschnittstellen zur Verfügung. Der Serverdienst nimmt Nachrichten entgegen und Verwaltet diese. Er besitzt *Exchanges* und *Queues*. Ein *Exchange* nimmt eine Nachricht entgegen und verteilt diese an eine oder mehrere *Queues*, die ein *Binding* auf den jeweiligen *Exchange* haben.

Abbildung 2.2: Produzent, Queue und Konsument

Es gibt verschiedene Arten von *Exchanges*, die Nachrichten auf verschiedene Weise verteilen.

Ein *Fanout-Exchange* sendet die Nachricht an jede *Queue*, mit der er ein *Binding* besitzt.

Ein *Direct-Exchange* sendet eine Nachricht nur an eine *Queue*, wenn der *Routing-Key* der Nachricht mit dem Namen einer *Queue* übereinstimmt, die natürlich auch ein *Binding* auf dem *Exchange* haben muss.

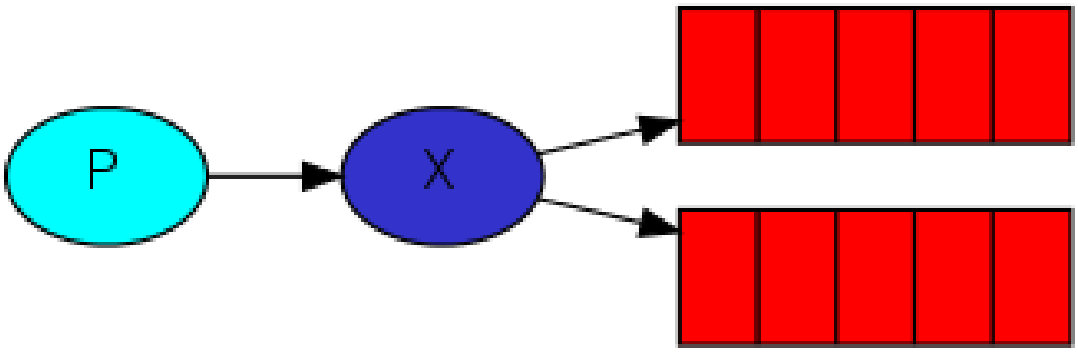

Abbildung 2.3: Produzent, Exchange und Queue

Ein *Topic-Exchange* wiederum verteilt an mehrere *Queues* in seinem *Binding*. Dabei dürfen die *Queues* keinen willkürlichen Namen tragen, sondern müssen einem gewissen Namensmuster Folgen, dass teilweise den *Routing-Key* der zu empfangenden Nachrichten reflektieren soll. Wenn zum Beispiel der *Routing-Key* einer Nachricht *core.failure.critical* wäre, so wäre auch **.*.critical* ein Name, unter dem eine *Queue* diese Nachricht empfangen würde. Dabei steht das Sternchen als Platzhalter für jedes mögliche Wort. Die Punkte reflektieren die Punkte im *Routing-Key*. Eine Raute ist dabei ein Platzhalter für mehrere Wörter, also über Punkte hinaus.

Der *Standard-Exchange* ist ein *Direct-Exchange* und wird innerhalb des Codes mithilfe einer leeren Zeichenkette ausgewählt. Er ist auch notwendigerweise mit allen *Queues* verbunden.

Ein *Exchange* wird von Produzenten befüllt. Ein Produzent erzeugt Nachrichten und schickt diese an *Exchanges* damit diese dann an einer oder mehreren *Queues* konsumiert werden.

Eine *Queue* ist ein Speicher, der Nachrichten zwischenspeichert und nach dem *First-in-First-Out (FIFO)*-Prinzip an Konsumenten verteilt. Sie sind Anlaufstelle für Konsumenten.

Ein Konsument meldet sich bei einer *Queue* und wird dort in eine Liste von Konsumenten aufgenommen. So wird er dann von dem Server mit Nachrichten bedient. Wird eine Nachricht konsumiert, kann ein Konsument diesen Vorgang bestätigen, falls der Produzent es wünscht.[5] Das heißt wiederum aber auch, dass der Produzent nun auch auf die Antwort des Konsumenten wartet und ihn somit potenziell blockiert.

Eine Nachricht wird in RabbitMQ mit vielen Attributen abgesendet. Sie folgt in erster Linie dem *AMQP*, doch RabbitMQ hat diese um einige Funktionen erweitert.

So kommt zum Beispiel das Attribut *reply_to* mit. Mit diesen Attribut kann eine Antwortadresse für zum Beispiel den Absender hinzugefügt werden. Dieses Attribut wird später im *Remote-Procedure-Call* benutzt. Sonst kommt offensichtlicherweise Informationen über die Zieladresse und den *Ziel-Exchange*, den *Routing-Key* und natürlich auch die eigentliche Nachricht, dem *Body*, mit.

Die Managementoberfläche und Nutzerschnittstellen bieten eine Plattform, von der man mit dem Serverdienst wie beschrieben interagieren kann.

2.3.2 Remote-Procedure-Call (RPC)

Der *RPC* ist ein Muster, das uns ermöglicht, auf eine Nachricht eine entsprechende Antwort zu bekommen, wie bei einem Funktionsaufruf.[27]

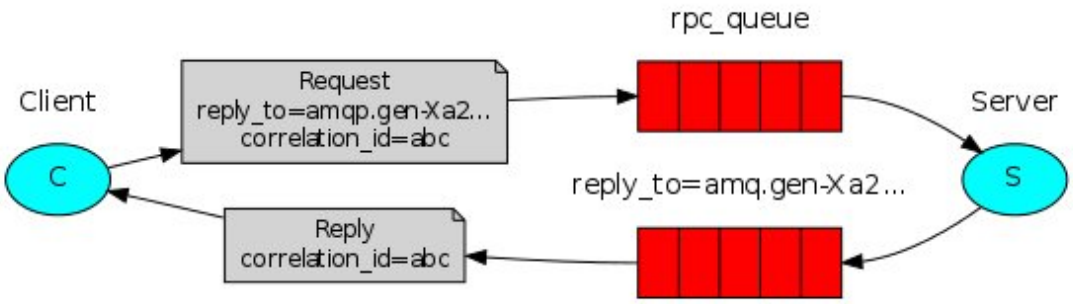

Abbildung 2.4: RPC

Für dieses Muster verwenden wir zuerst einen normalen Produzenten, den wir jedoch um die Attribute *reply_to* und *correlation_id* erweitern. Wir setzten die *Antwort-Queue* auf einen zufälligen Namen, den wir uns von RabbitMQ generieren lassen. Die *correlation_id* lassen wir zufällig generieren. Dann öffnen wir einen Konsumenten auf eben diese *Queue*.

Der Konsument nimmt nun ganz normal diese Nachricht an und ließt das *reply_to*-Attribut aus. Und sendet eine Antwort auf diese *Queue*.

Der Sender konsumiert diese Antwort und überprüft die *corelation_id*. Wenn diese nun übereinstimmt, gibt er den Antworttext zurück und trennt die Verbindung zu der *Queue*, die, wenn diese die Eigenschaft *auto_delete* gesetzt hat, sich dann von alleine löscht.

2.4 Implementierung

Bevor ich beginnen konnte, meinen Prototypen zu bauen, musste ich eigenständige Teile in der Serverarchitektur erkennen und isolieren. Meine Methodik lässt sich darauf beschränken, dass ich das Projekt kopierte, die zu isolierende Funktion im

Code ausmachte und nun alles entfernte, was offensichtlich keine Abhängigkeit zu der Funktion hatte. Nun konnte ich den Aufruf durch die alte Schnittstelle löschen und mit dem Aufruf durch RabbitMQ ersetzen. Ich benutzte dafür das *RPC*-Muster, das ich zuvor schon beschrieben habe.

```
public void startWorker()
{
    factory = new ConnectionFactory() { HostName = "localhost" };
    connection = factory.CreateConnection();
    channel = connection.CreateModel();
    channel.QueueDeclare(queue: requestQueue, durable: false, exclusive: false, autoDelete: false, arguments: null);
    channel.BasicQos(0, 1, false);
    var consumer = new EventingBasicConsumer(channel);
    channel.BasicConsume(queue: requestQueue, autoAck: false, consumer: consumer);

    consumer.Received += (model, ea) =>
    {
        string response = null;

        var body = ea.Body;
        var props = ea.BasicProperties;
        var replyProps = channel.CreateBasicProperties();
        replyProps.CorrelationId = props.CorrelationId;

        try {
            var message = Encoding.UTF8.GetString(body);
            ExecuteAction();
            response = "Received" + message;
        }
        catch (Exception e) {
            response = "";
        }
        finally {
            var responseBytes = Encoding.UTF8.GetBytes(response);
            channel.BasicPublish(exchange: "", routingKey: props.ReplyTo, basicProperties: replyProps, body: responseBytes);
            channel.BasicAck(deliveryTag: ea.DeliveryTag, multiple: false);
        }
    };
}
```

Abbildung 2.5: RPC-Empfänger

In Abbildung 2.5 sehen wir ein Beispiel für einen Empfänger. Als Erstes wird eine Verbindung zum Server aufgebaut und die *Anfrage-Queue* deklariert. Zusätzlich konfigurieren wir den Server nur eine unbestätigte Nachricht pro *Queue* zu erlauben. Wird dieser Wert erreicht, so werden keine Nachrichten mehr von der *Queue* verteilt. Dann wird ein Konsument erzeugt und dem Event *Received* via *Lambda*-Ausdruck eine Funktion übergeben, die auf empfangene Nachrichten reagiert. Innerhalb dieser wird die Nachricht ausgelesen und mit den ausgelesenen Informationen eine Methode aufgerufen. Daraufhin schickt sie die Rückgabe der Methode an die *Antwort-Queue*. Zu guter Letzt wird der Konsument dann auch an die *Anfrage-Queue* angehängt.

Den Sender oder auch *RPC-Client* genannt, implementierte ich an der Stelle, an welcher auch normalerweise der *HTTP*-Aufruf für eben diese Funktion stattfand.

Abbildung 2.6 zeigt einen Beispiel-Sender. Er baut wie schon der Empfänger zuerst eine Verbindung zum Server auf. Danach erzeugt er Eigenschaften für die Nachricht und befüllt diese mit einem zuvor erzeugten Namen für die *Antwort-Queue* und einer zufälligen *correlation_id*. Nun wird nur noch dem Konsumenten eine Funktion übergeben, die den Inhalt der Nachricht als Zeichenkette zwischenspeichert. Wenn nun die *Call*-Methode aufgerufen wird, wird eine Nachricht an die *Anfrage-Queue* gesendet und direkt danach der Konsument an die *Antwort-Queue* ange-

```
public class RpcClient {
    private readonly IConnection connection;
    private readonly IModel channel;
    private readonly string replyQueueName;
    private readonly EventingBasicConsumer consumer;
    private readonly BlockingCollection<string> respQueue = new BlockingCollection<string>();
    private readonly IBasicProperties props;

    1 reference
    public RpcClient() {
        var factory = new ConnectionFactory() { HostName = "localhost" };

        connection = factory.CreateConnection();
        channel = connection.CreateModel();
        replyQueueName = channel.QueueDeclare().QueueName;
        consumer = new EventingBasicConsumer(channel);

        props = channel.CreateBasicProperties();
        var correlationId = Guid.NewGuid().ToString();
        props.CorrelationId = correlationId;
        props.ReplyTo = replyQueueName;

        consumer.Received += (model, ea) => {
            var body = ea.Body.Span;
            var response = Encoding.UTF8.GetString(body);
            if (ea.BasicProperties.CorrelationId == correlationId) {
                respQueue.Add(response);
            }
        };
    }

    1 reference
    public string Call(string message) {
        var messageBytes = Encoding.UTF8.GetBytes(message);
        channel.BasicPublish(exchange: "", routingKey: "rpc_queue", basicProperties: props, body: messageBytes);
        channel.BasicConsume(consumer: consumer, queue: replyQueueName, autoAck: true);
        return respQueue.Take();
    }
```

Abbildung 2.6: RPC-Sender

hängt. Wurde die Antwort empfangen, wird nun die zwischengespeicherte Zeichenkette zurückgegeben und aus dem Zwischenspeicher entfernt.

Nachdem ich also eine Methode hatte, Zeichenketten mit dem Message Broker austauschen zu können, musste ich nun entscheiden, wie ich die Übergabeobjekte in solch eine Zeichenkette umwandeln sollte, sodass diese auch am Empfänger rückübersetzt werden konnten. Ich entschied mich dafür, die Objekte in eine *JSON*-Zeichenkette umzuwandeln, da dies das Rückübersetzen in ein Objekt sehr einfach gestaltete und ich bis zu diesem Zeitpunkt aus den vorherigen Aufgaben schon mehrere Erfahrungen zum Umgang mit *JSON*-Objekten hatte. Da es sich aber um komplexe Objekte handelte, die nicht nur Basisdatentypen als Werte hatten, musste ich den Übersetzer der *JSON*-Bibliothek, die ich verwendete, anpassen, sodass innere Objekte der Anfrage mit übersetzt werden konnten. Dann machte ich mich langsam daran, gezielt überflüssigen Code zu entfernen, solange bis nur noch der Kern der Funktion über war.

Den Prototypen zum *Fanout-Exchange* implementierte ich als *Dead-Letter-Exchange*[7], eine Art *Exchange*, den man bei der Initialisierung einer *Queue* angeben kann. An ihn werden Nachrichten gesendet, wenn diese Nachrichten eine in der *Queue* gesetzten *Time-To-Live (TTL)*[32] überschreiten.

3 Containervirtualisierung mit Docker

Docker[9] ist ein Werkzeug zur Erstellung und Verwaltung von Containern. Container sind kleine, voneinander unabhängige Umgebungen, in denen Programme lauffähig und von außerhalb gezielt erreichbar sind. Sie werden dafür eingesetzt, eine Anwendung von äußeren Einflüssen zu befreien. Ihre Prämisse lautet, wenn eine Anwendung auf einem System funktioniert, funktioniert sie auf allen Systemen. Ein Container hat für gewöhnlich nur das Nötigste installiert, um lauffähig zu sein, alles andere wird vom Entwickler selbst beigesteuert. Was ihnen wiederum einen entscheidenden Vorteil im Vergleich zu einer *virtuellen Maschine (VM)* gibt. Aber nicht nur das. Einen Container aufzusetzen ist auch mit um einiges weniger Aufwand verbunden als das Aufsetzen einer *VM*, da bei Containern kein Gastbetriebsystem mit eingerichtet werden muss.

In Hinsicht auf die gezielte Serverstruktur soll Docker das Rückgrat dieser stellen. Die einzelnen Funktionen sollen durch die Containervirtualisierung voneinander abgekoppelt und skalierbar werden.

3.1 Grundlagen der Containervirtualisierung

Auch wenn Containervirtualisierung eine Art der Virtualisierung ist, laufen hierbei die Container nicht über ein Gastbetriebssystem, sondern befinden sich trotzdem auf ihrem Hostbetriebsystem. Auch wenn sie strikt getrennte Programminstanzen mit allesamt eigenen Abhängigkeiten und Ressourcen sind, teilen sie sich denselben Systemkernel.

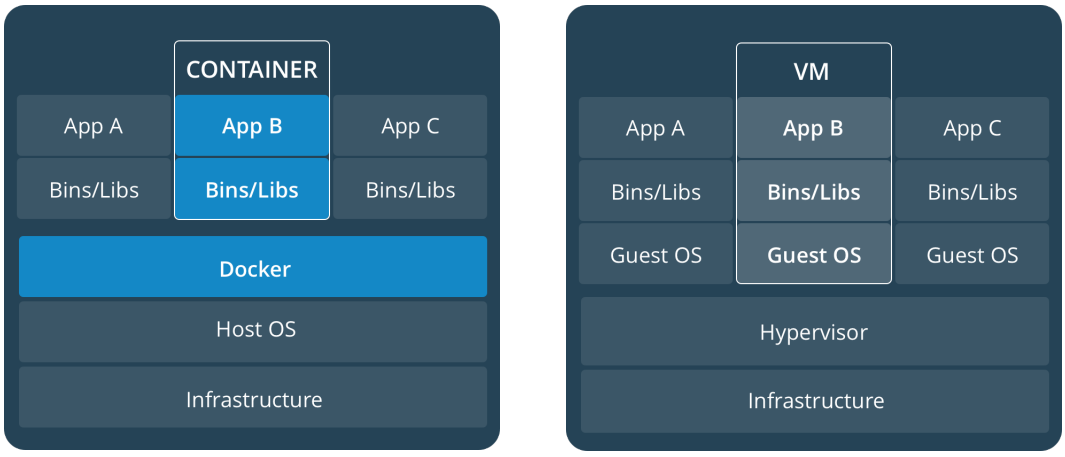

Abbildung 3.1: Containervirtualisierung im Vergleich zu klassischer Virtualisierung.

Auf Abbildung 3.1 sehen wir die Containervirtualisierung im Vergleich zur klassischen Virtualisierung. Es ist leicht zu erkennen, dass durch die Einsparung des Gastbetriebssystems auf jeder Instanz wertvoller Speicherplatz erhalten werden kann.[22]

Dieser Vorteil wird aber erst bei einer größeren Anzahl an Instanzen richtig ersichtlich, weswegen diese Art der Virtualisierung bei verteilten Systemen zum Einsatz kommt. Ein verteiltes System ist ein System, in dem die einzelnen verschiedenen Funktionen, die das System erfüllt, von unabhängigen, getrennten Komponenten erfüllt werden, die auf eine dynamische Weise miteinander verbunden sind. So wird es ein Einfaches, dieses System horizontal zu Skalieren. Das heißt, Instanzen des zu Skalierenden hinzuzufügen. Durch die dynamische Einbindung aller Komponenten wird sicher gestellt, dass die Last nun auf alle Instanzen desselben Typs gleichmäßig verteilt sind. Dies geschieht zum Beispiel mit einem *Load Balancer*. Mit einem Message Broker ist Ähnliches zu erreichen.

3.2 Docker Grundlagen

Docker ist nun eine Plattform, auf der wir mit Containern interagieren können. Das heißt unter anderem Container erzeugen und löschen, Ports teilen und Speicher zuteilen. Ein Container wird mit einem *Image* erzeugt. Ein *Image* ist ein Abbild der auszuführenden Programminstanz und wird aus einer *Registry*, einer zentralen Sammelstelle für *Images*, gezogen. Docker *Images* werden mit *Dockerfiles* erzeugt und können dann lokal verwaltet und auf externe *Registries* geschoben werden. Eine *Registry*, die nicht über *HTTPS*, sondern nur über *HTTP* erreichbar ist, nennt man eine *Unsecure Registry* und muss der *daemon.json* hinzugefügt werden. Das geht unter Windows einfach über die Einstellungen des Programms *Docker Desktop*. Da mehrere Instanzen desselben Containers auch mehrere Anwendungen auf demselben Port und derselben Adresse heißen würde, muss man auf die Ports, die von der Anwendung verwendet werden, von anderen Ports verweisen. Dies nennt sich *Port-Mapping*. Docker bietet auch Container mit Netzwerken zu verbinden, sodass ein *Domain Name System (DNS)* Containernamen auf ihre IP-Adressen übersetzt. Es gibt zwar auch Standardlösungen für diese Netzwerke, auf denen aber die Auflösung von Containernamen nicht möglich ist. Deshalb ist es ratsam, diese Netzwerke selbst zu definieren. Dabei ist aber zu bedenken, ob die Anfrage auch wirklich vom Container aus gesendet wird und nicht wie im Falle von JavaScript clientseitig gestellt wird. Ein anderes Konzept, das Docker einführt, ist *Volume*. Ein *Volume* ist eine Art Speicher, mit dem sich Container verbinden und es dann benutzen können. Container können über die Kommandozeile mit dem Befehl *docker run example/image:test* gestartet werden. Dabei können zahlreiche Parameter mitgegeben werden. Mit *-p* können so *Port-Mappings* erstellt werden. Die lokalen *Images* können mit *docker images* eingesehen werden. Mit *docker ps* werden die laufenden Container angezeigt. Möchten wir auch die beendeten Container aufgelistet haben, so fügen wir das Kürzel *-a* an. Wenn ein Container gestoppt werden soll, geht dies mit *docker stop container_id* und zum Entfernen mit *docker rm container_id*. *Images*

werden mit *docker rmi image_id* entfernt. Zur Fehlersuche können wir mit *docker logs container-name* auf die *Log*-Dateien eines Containers zugreifen.[13]

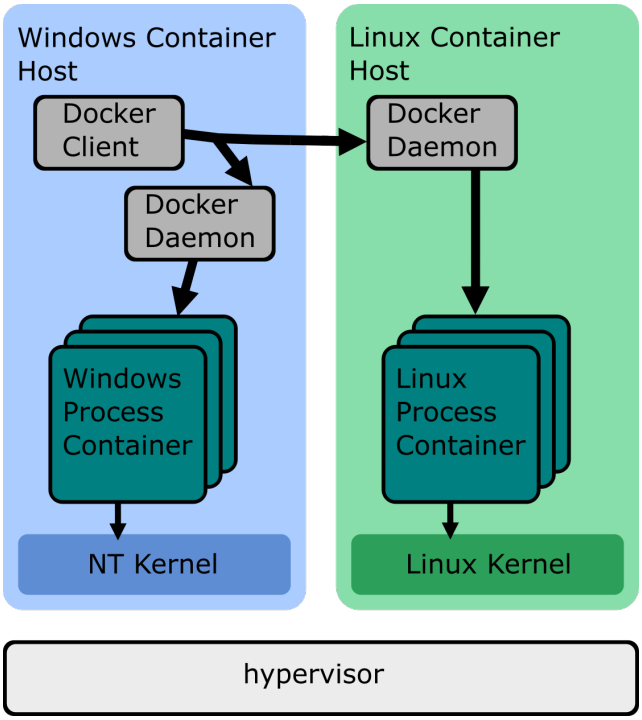

Abbildung 3.2: Linux Container auf Windows

Für gewöhnlich werden Linux-Container verwendet, auch unter Windows. Docker erstellt dafür dann eine *VM*, die das gewünschte Hostbetriebssystem so zur Verfügung stellt. Das wird auf Abbildung 3.2 verdeutlicht.

3.2.1 Dockerfile

Wie schon zuvor erwähnt wird mit einem *Dockerfile* ein Image unserer Anwendung erstellt.[10]

In Abbildung 3.3 sehen wir, wie ein *Dockerfile* exemplarisch aussehen könnte. Als Erstes wird eine Basis referenziert, die unser *Image* braucht, um ausgeführt werden zu können. In diesem Fall ist dies *.Net Core 3.1*. Danach geben wir die Anweisung, den Port 80 für diese Anwendung freizugeben. Nun wird alles Programmrelevante in den Kontext gezogen, gebaut und veröffentlicht. Zuletzt wird ein Einstiegspunkt gesetzt, das heißt, einen Befehl, der bei Start des Containers aufgerufen wird, um die Anwendung auszuführen.

Der *FROM* Befehl, den wir mehrfach in dem Dokument finden können, initialisiert ein *Basisimage* und wird im Weiteren dazu verwendet, die verschiedenen Phasen während des Bauvorgangs zu trennen. In der vorletzten Zeile finden wir eine *–from* Anweisung, die aussagt, dass hier auf den Kontext des *Images publish*, das zuvor initialisiert wurde, zugegriffen wird.

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1-buster-slim AS base
WORKDIR /app
EXPOSE 80

FROM mcr.microsoft.com/dotnet/core/sdk:3.1-buster AS build
WORKDIR /src
COPY ["SmartMapsSearch/SmartMapsSearch.csproj", "SmartMapsSearch/"]
RUN dotnet restore "SmartMapsSearch/SmartMapsSearch.csproj"
COPY . .
WORKDIR "/src/SmartMapsSearch"
RUN dotnet build "SmartMapsSearch.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "SmartMapsSearch.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "SmartMapsSearch.dll"]
```

Abbildung 3.3: Beispiel für einen Dockerfile

```
FROM nginx:alpine AS final
WORKDIR /usr/share/nginx/html
COPY --from=publish /app/publish/wwwroot .
COPY nginx.conf /etc/nginx/nginx.conf
```

Abbildung 3.4: Webserver als Basisimage

Im Falle einer statischen Website, die von einem *Blazor-WebAssembly*-Projekt erzeugt wird, muss diese zwar mit *.Net* gebaut werden, braucht aber keine *Runtime*-Umgebung, sonder benötigt einen Webserver. Dies wird wie schon beschrieben erreicht, indem wir die Basis wechseln und die gebauten Daten auf das neue *Basisimage* ziehen, wie auf Abbildung 3.4 gezeigt. Ein Einsprungpunkt wird in diesem Fall nicht benötigt.

Um einen *Dockerfile* nun ausführen zu können und so ein *Image* zu erstellen, müssen wir in der Kommandozeile den Befehl *docker build .* in dem jeweiligen Ordner ausführen. Damit wir das Image später besser referenzieren können, lohnt es sich, dem *Image* ein *Tag* zu geben. Dies machen wir mit dem Kürzel *-t*. Der *Dockerfile* kann mit dem Kürzel *-f* angegeben werden. So kann auch auf eine andere Position in der Ordnerstruktur hingewiesen werden.

Aus Sicherheitsgründen lohnt es sich abzuwägen, ob beim Bauen des *Images* ein neuer Benutzer angelegt und die Anwendung durch diesen gestartet werden sollte. Der Standardnutzer könnte eventuell *Root*-Rechte haben.

Images erzeugen zu können ist in Hinblick auf Kubernetes unerlässlich.

3.2.2 Docker-Compose

Mithilfe von Docker-Compose lassen sich Container vollständig konfiguriert und durch das Ausführen eines einzelnen Dokumentes starten. Das ermöglicht dem Nutzer komplexe Anwendungssysteme mit nur einem simplen Befehl ausführen zu können.[23]

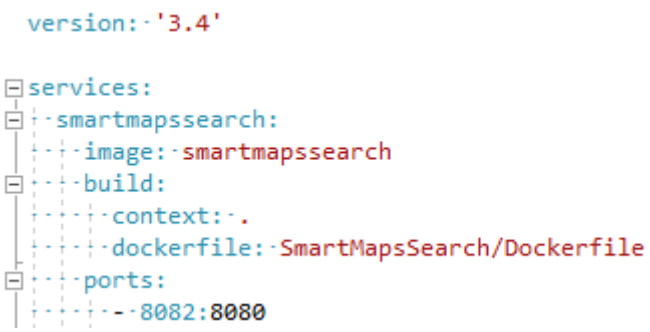

Abbildung 3.5: Beispiel für Docker-Compose

Dabei ist Docker-Compose sehr simpel aufgebaut und wird mit der Auszeichnungssprache *YAML* spezifiziert. Im Beispiel, das in Abbildung 3.5 zusehen ist, wird ein Container namens *smartmapssearch* definiert, der ein gleichnamiges *Image* benutzt. Es werden weiterhin ein *Port-Mapping* hinzugefügt und Information zum Bauen des *Images* angegeben. Gebaut wird das *Image* wie zuvor mit einem *Dockerfile*, dessen Position und Namen wir angeben.[3]

Mit Docker-Compose können aber auch *Volumes* und *Netzwerke* hinzugefügt werden, die natürlich auch gleich an Container angebunden werden können. Containern kann man ebenfalls auch einen Alias hinzufügen, unter dem sie einen anderen Container innerhalb des Netzwerkes finden können.

In Abbildung 3.6 sehen wir einen Codeauszug, der dies so implementiert. Unter dem Bezeichner *networks* werden alle Informationen des eigenen Netzwerks angegeben. Das deklarierte Netzwerk ist von Typ *bridge*, was bedeutet, dass Anfragen mit einem bestimmten Alias oder dem Containernamen auf die jeweilige IP-Adresse umgeleitet werden. Von diesem Typ ist auch das Standardnetzwerk. Der Netzwerktyp *host*, der nur unter Linux-Systemen auswählbar ist, wird hier vernachlässigt. Nach den Netzwerken werden wie gewohnt die Container definiert. Denen nun noch das Netzwerk und, wenn gewünscht, auch einen Alias hinzugefügt wird. Hier wurde der Dienst namens *smartsearch* mit einem gleichnamigen Alias versehen. Zu beachten ist, dass die erzeugten Containernamen nicht gleich den Servicenamen sind und es ratsam ist, falls benötigt, einen Alias zu setzen.

```
networks:
  servicenetworkautocomplete:
    driver: bridge
services:
  autocomplete.server:
    image: yellowmap/autocompleteserver
    build:
      context: .
      dockerfile: Blazor.Autocomplete/Server/Dockerfile
    ports:
      - 12346:80
    links:
      - "smartmapssearch:smartmapssearch"
    networks:
      - servicenetworkautocomplete
  smartmapssearch:
    image: yellowmap/smartmapssearch:dev
    ports:
      - 12345:80
    networks:
      - servicenetworkautocomplete
```

Abbildung 3.6: Netzwerk mit Alias

3.3 Praktische Anwendung und Verdikt

Container sind zusammen mit Docker ein sehr praktisches Werkzeug für die Entwicklung von Diensten. Die isolierte Umgebung, die geboten wird, macht das Entwickeln von kleineren Anwendung sehr einfach, auch wenn eben diese Isolation anfangs zu so manchen Problemen führt, die zumeist aber durch ein Netzwerk gelöst werden können. In der Praxis war die Containervirtualisierung am besten bei kleineren Anwendungen mit dedizierten Aufgaben zu realisieren, so wie für den Dienst, den ich schon zuvor aus dem Serversystem herauslöste. Durch Docker-Compose ist sogar kleine Ansätze von Skalierbarkeit zu erkennen, doch auch wenn Docker-Compose wesentlich hilft, kleine Systeme aufzubauen und mit anderen Entwicklern zu teilen, ist der Aufwand, der mit der Erhaltung dieses Systems verbunden ist, sehr hoch.

Falls nun in einem Containersystem einer der Container blockieren sollte, muss dies rechtzeitig erkannt werden und der betroffene Container manuell neu gestartet werden. Auch ist *Loging* und *Debugging* sehr umständlich. Dafür muss eine eigene Lösung realisiert werden, die die Logingdaten des Gesamtsystems zum Beispiel in einer Datenbank speichert. In diesem Bereich wäre auch sicherlich eine Lösung mittels Message Broker denkbar.

Die schon angesprochene Skalierbarkeit ist mittels Docker-Compose zu erreichen, aber nur rudimentär implementiert. So lassen sich mit *docker-compose up –scale SERVICE=NUM* eine bestimmte Anzahl an Containern eines bestimmten Dienstes starten. Alternativ kann auch *docker-compose scale SERVICE=NUM* eingesetzt werden. Dieser Befehl gilt aber als veraltet und soll nicht mehr verwendet

werden. Hat man innerhalb der *docker-compose.yaml* ein *Port-Mapping* festgelegt, schlägt das Erstellen der Container auch fehl, da diese nicht alle unter demselben Port erreichbar seinen können.

4 Containerorchestrierung mit Kubernetes

Kubernetes[19] ist ein Werkzeug zur Container-Orchestrierung im Besonderen zum Erzeugen und verwalten von Containerclustern. Mit Hilfe von Kubernetes können große, komplexe Containersysteme aufgespannt werden, in denen die einzelnen Anwendungen auf keinen bestimmten Ausführungsort angewiesen sind und es dadurch egal wird, auf welchem Rechner die Anwendungen ausgeführt werden. Neben dem *Routing* übernimmt Kubernetes auch das *Monitoring* der Anwendungen und startet diese neu, wenn diese nicht mehr wie gewohnt zu erreichen sind. Letztlich lässt sich mit Kubernetes auch die Skalierung der Anwendungen automatisieren. Wozu verschiedenste Metriken hinzugezogen werden können.[34]

4.1 Kubernetes Komponenten

Kubernetes führt an dieser Stelle eine Reihe an Konzepten[18] ein, die das Interagieren mit der Hardware[20] vereinfachen und so abstrahieren. Ein Cluster besteht dabei aus mindestens einem Worker-Knoten und einem Master-Knoten. Auf dem Worker-Knoten laufen sogenannte *Pods*, die die Funktion des eigentlichen Containers darstellen.

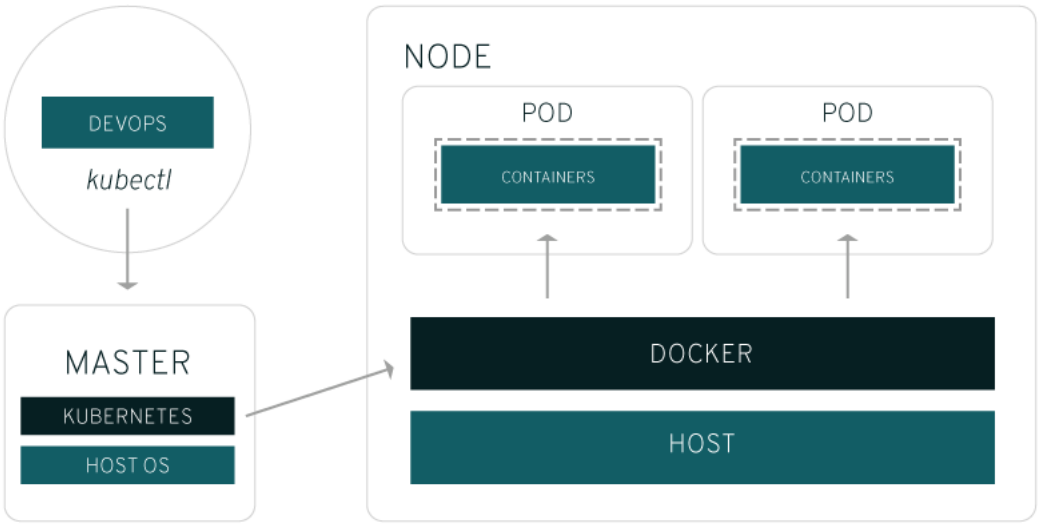

Abbildung 4.1: Simples Kubernetes Cluster

Auf Abbildung 4.1 sehen wir ein *Kubernetes-Cluster* mit einem *Worker-Knoten* auf dem sich zwei *Pods* befinden. Auf dem *Master-Knoten* laufen alle Controller, die Kubernetes für die Interaktion mit den *Cluster*-Komponenten benötigt. Darunter ist der *API-Server*. Er ist verantwortlich für den allgemeinen Datenaustausch innerhalb des Systems und bietet eine Schnittstelle, über die der Nutzer den Status des *Clusters* abfragen kann, aber die auch allen Komponenten innerhalb des *Clusters*

ermöglicht, miteinander zu kommunizieren. Auch werden durch diese Schnittstelle die einzelnen Komponenten konfiguriert.

Mit einem *Worker-Knoten* kommuniziert der *API-Server* über ein *kubelet*, das auf jedem *Worker-Knoten* läuft. Es ist dafür verantwortlich, die auf dem Knoten befindlichen Komponenten zu konfigurieren und zu überwachen.

Pods stellen, wie schon zuvor erwähnt, unsere eigentliche Anwendung dar. Sie werden mit einem Docker-Image konfiguriert und können entweder manuell in der Kommandozeile, mit *YAML* oder automatisch mit einem *Deployment* erzeugt werden. Sie können mehrere Container beinhalten.[24] Ein *Deployment* ist zur Verwaltung eines *Pods* und seines *ReplicaSets* gedacht. Erstellt man ein *Deployment*, so werden beide Komponenten erzeugt.[8] Das *ReplicaSet* ist für den Erhalt einer konstanten Anzahl gleichartiger *Pods* gedacht.[28] Neben dem *Deployment* gibt es auch *DaemonSets* und *StatefulSets*, die diesem ähneln. Ein *DaemonSet* geht sicher, dass auf allen oder auch nur manchen Knoten ein Exemplar des spezifizierten *Pods* läuft.[6] Das *StatefulSet* dem entgegen, verhält sich fast wie ein *Deployment*, aber die *Pods*, die es erzeugt, sind nicht absolut austauschbar und besitzen deshalb eine *ID*, mit der man diese auseinanderhalten kann. Dies kann wichtig sein, falls ein Container eigenen Speicher besitzt.[31] Egal wie man nun seine *Pods* erzeugt, um auf seine Anwendung zugreifen zu können, braucht man einen *Service*, der einen einfachen Weg bietet, die Anwendung zu erreichen. Auch benötigt man einen *Service*, weil *Pods* innerhalb eines *Deployments* nicht permanent gleich bleiben. Sie werden je nach Bedarf gestartet und beendet und jeder Neue bekommt auch eine neue IP-Adresse zugewiesen. Man kann also die *Pods* auf einem *Deployment* besser erreichen, indem man einen *Service* erzeugt, der alle *Pods* durch eine IP-Adresse erreichbar macht. Eine IP-Adresse lässt sich innerhalb des Clusters aber, ähnlich wie auch Docker, komplett vermeiden, indem man dem Schema *service-name.namespace.svc* folgt.[29]

Mit dem *Cluster* interagieren wir mittels des *kubectl* Befehls. So können wir durch *kubectl create deployment somename –image some/image:test* ein *Deployment* erschaffen und mit *kubectl delete deployment somename* wieder entfernen.

Doch deutlich genauer und auch angenehmer ist das Erzeugen der Komponenten über *YAML*.

Auf der Abbildung 4.2 sehen wir ein Beispiel dazu, wie dies für einen *Pod* aussieht. Zu Beginn wird die *apiVersion*, die Art der Komponente, der Namen und die Labels des *Pods* festgelegt. Die *apiVersion* kann sich von Komponente zu Komponente unterscheiden, wie wir später sehen werden. In *spec* werden nun die zum *Pod* gehörigen Container spezifiziert. Der Unterpunkt *containers* ist eine Liste und kann mehrere Container beinhalten. Zu einem Container gehören Name, Image, eine Liste an Ports und idealerweise auch eine *readinessPobe* und *livenessProbe*. Diese sind in Syntax und Argumenten identisch und unterscheiden sich nur in Funk-

```
apiVersion: v1
kind: Pod
metadata:
  name: autocomplete
  labels:
    app: autocomplete
spec:
  containers:
  - name: autocomplete
    image: yellowmap/autocompletewebservice:test
    ports:
    - containerPort: 80
    readinessProbe:
      httpGet:
        path: /api_autocomplete/v2/search
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 5
    livenessProbe:
      httpGet:
        path: /health
        port: 80
      initialDelaySeconds: 10
      periodSeconds: 5
```

Abbildung 4.2: Pod-Konfiguration

tion. Die *readinessProbe* hat die Aufgabe zu erkennen, ab wann die Anwendung betriebsbereit ist und Anfragen entgegennehmen kann. Die *livenessProbe* überprüft, ob die Anwendung noch betriebsbereit ist oder ob diese neu gestartet werden müsste. Im Beispiel benutzen beide eine HTTP-Anfrage, aber es diese Überprüfung kann auch auf andere Weise realisiert werden.[4]

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: autocomplete
  namespace: yellowmap-autocomplete
  labels:
    app: autocomplete
spec:
  replicas: 1
  selector:
    matchLabels:
      app: autocomplete
  template:
    metadata:
      name: autocomplete
      labels:
        app: autocomplete
    spec:
      containers:
```

Abbildung 4.3: Deployment-Konfiguration

Bei einem *Deployment* verändert sich nur wenig, wie man in Abbildung 4.3 sehen kann. In *metadata* wurde lediglich, um die Komponenten vom Rest des Clusters zu trennen, ein *Namespace* hinzugefügt. In *spec* kommt nun doch mehr hinzu. So können wir nun mit *replicas* die Anzahl der *Pods* in unserem *Deployment* auswählen und mit *selector* werden die vom *Deployment* betroffenen Container ausgewählt. Unter *template* finden wir die *metadata* und *spec* unseres *Pods* aus Abbildung 4.2 wieder.

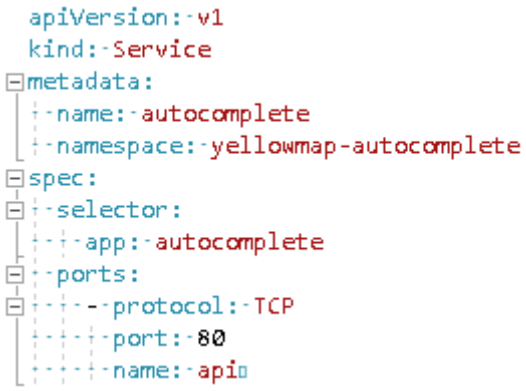

Abbildung 4.4: Service-Konfiguration

Auf Abbildung 4.4 sehen wir zu guter Letzt einen *Service*, der auf diese Weise definiert wird. Die allgemeine Struktur bleibt im Vergleich zu dem *Pod* und dem *Deployment* gleich. In *spec* haben wir einen *selector*, der das *Deployment* referenziert. Danach folgt eine Liste an Ports. Diese werden später dann nach außen zur Verfügung gestellt.

4.1.1 Helm

Helm[14] ist eine Erweiterung auf das uns schon bekannte *YAML*-Format, das bei der Erzeugung von Kubernetes-Komponenten zum Einsatz kam. Ein sogenanntes Helm-Chart definiert alle Komponenten im selben Schema wie schon zuvor, doch fügt einiges an Funktionalität hinzu. Ein Helm-*Chart* besitzt eine *Chart.yaml*, die Informationen zum *Chart* beinhaltet, einen *Template*-Ordner, der die *YAML*-Dateien der Komponenten beinhaltet, und eine *values.yaml*, in der Werte global für das komplette Chart gesetzt werden, die dann in den *YAML*-Dateien referenziert werden können. Es können sich auch in einem Charts-Ordner weitere *Subcharts* befinden. Helm-*Charts* können wie auch Images in Registries gespeichert werden.[2]

4.1.2 Skalierung

Die Idee hinter der Skalierung von Prozessen ist es, bei ansteigender Last auf ein System, dies auszugleichen, indem die relevanten Komponenten durch die Zuwei-

sung von Ressourcen oder die Vervielfältigung dieser gestärkt werden. Die Zuweisung zusätzlicher Ressourcen nennt man vertikale Skalierung[33], die Vervielfältigung der betroffenen Container horizontale Skalierung.

Beide Arten können durch Kubernetes dynamisch angepasst werden. Dies funktioniert auch automatisch. Bei horizontaler Skalierung erzeugt man in Kubernetes einen *Horizontal Pod Autoscaler (HPA)*[15] auf das gewünschte *Deployment*. Im *Deployment* müssen die Bedingungen für die Replikation der *Pods* angegeben sein. Dies geht am einfachsten über die Angabe der Prozessorauslastung. Zum Lesen der Metrikdaten von *Pods* benötigen wir einen *metrics-server*.[16] Diesen bekommen wir von *Kubernetes SIGs* auf *GitHub*[21].

4.2 Aufsetzen eines verteilten Systems

Nun galt es, das Gelernte in einem weiteren Prototyp zu verwirklichen. Das Ziel war es, das neue Projekt *Autocomplete* mit Testanwendung auf einem Kubernetes-Cluster als ein verteiltes System zu realisieren. Die Testanwendungen waren einerseits eine Benutzeroberfläche zum Testen der Rückgabe auf Nutzereingaben, andererseits ein Dienst zum schreiben von Fehlerberichten auf eine externe Datenbank. Den Projekten musste jeweils *readiness*- und *liveness*-Endpunkte hinzugefügt werden. Dies macht man in der *Setup.cs* indem man ihn in der Methode *Configure* über *UseEndpoints* konfiguriert.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks( pattern: "/api/healthcheck/live", new HealthCheckOptions()
    {
        Predicate = (_) => false
    });

});
```

Abbildung 4.5: Endpunkte hinzufügen in ASP.NET Core

In Abbildung 4.5 wird so ein Endpunkt hinzugefügt, der aber keine *Health Checks*[17] prüft und somit immer *healthy* zurückgeben wird. *Health Checks* werden hier aber nicht weiter betrachtet, da der Fokus der Betrachtung auf Kubernetes liegen soll.

Weiter habe ich für alle betroffenen Projekte mit einem *Dockerfile* Images gebaut und auf eine Registry geschoben, auf die ich von dem Cluster aus Zugriff hatte. Die Adressen der Dienste, die nun innerhalb des Clusters laufen und untereinander kommunizieren sollten, mussten davor natürlich so angepasst werden, dass diese auf einen *Service* und nicht einfach auf eine Adresse oder auf einen Container verwiesen. Kleinere Probleme gab es mit der *Registry*, da diese eine *insecure registry* war und somit dem Cluster hinzugefügt werden musste.

Um ein *Cluster* auf dem eigenen Rechner aufzusetzen, eignet sich Minikube. Wurde Minikube installiert und den Umgebungsvariablen hinzugefügt, können wir mit *minikube start* ein eigenes *Cluster* starten. Das ist sinnvoll aus Test- und Entwicklungszwecken. Mit Minikube lässt sich die Verwendung eines *Ingress*-Controllers komplett vermeiden. Ein *Service* kann auf *localhost* erreicht werden, wenn dieser mit *minikube service svc-name* ausgeführt wird.

Nachdem ich die Funktion vorerst lokal getestet hatte, konnte ich nun anfangen, die verschiedenen *YAML*-Dateien für die Kuberntes-Komponenten zu schreiben. Diese blieben zumeist unverändert von den Testdateien, die ich schon zuvor erstellt hatte. Für die drei Anwendungen gab es jeweils ein *Deployment* und einen *Service*. Des Weiteren gab es einen *Ingress*, der Anfragen auf die drei *Services* aufteilt. Ich verwendete *livenessProbes* und *readinessProbes* aber musste leider die automatische Skalierung außer acht lassen, da der *metrics-server* weder auf dem Testsystem vorhanden war, noch ich die nötigen Rechte hatte, ihn zu installieren. Für den Gebrauch der *Registry* musste ein *Secret* auf dieser hinzugefügt werden. Ebenso musste das *Secret* in der Datei unter *imagePullSecrets* referenziert werden. Da das Image, wenn sich dessen Namen oder *Tag* nicht verändert hat, nicht aktualisiert und die lokale Version verwendet wird, setzte ich die *imagePullPolicy* auf *Always*.

```
paths:
- path: "/api/ReportApi"
  backend:
    serviceName: autocomplete-server
    servicePort: 8080
- path: "/api"
  backend:
    serviceName: autocomplete
    servicePort: 8080
- path: "/metrics"
  backend:
    serviceName: autocomplete
    servicePort: 8080
- path: "/"
  backend:
    serviceName: autocomplete-client
    servicePort: 8080
```

Abbildung 4.6: Ingress Paths

Auf Abbildung 4.6 sehen wir, unter welchen Paths die verschiedenen Services durch den Ingress erreichbar sind. Interessant ist, dass aufgepasst werden muss, dass Anwendungen sich mit ihren Endpunkten nicht gegenseitig in die Quere kommen, falls wir wollen, dass diese auf derselben URL erreichbar sind. Somit interagieren wir mit den Containern, als wären diese ein einzelnes System, aber halten diese austauschbar und dynamisch.

Ein Helm-Chart erstellte ich nur für die *Autocomplete*-Anwendung, für die Testanwendungen war dies meiner Meinung nicht notwendig, da diese nur zum Testen benötigt wurden.

5 Fazit

Nach Abschluss der Arbeit gilt nun zu bewerten, ob und wie die gesetzten Vorgaben denn jetzt schließlich erreicht oder auch nicht erreicht wurden.

Dazu vorerst einen kleinen Rückblick auf das Geschaffene. Das Produkt der Arbeit sind zweit funktionstüchtige Prototypen. Diese resultieren einmal aus der Untersuchung zur Nachrichtenübermittlung mittels Message Broker und der Untersuchung zur Containervirtualisierung und Containerorchestrierung.

Die Untersuchung der Nachrichtenübermittlung mittels Message Broker, die mit RabbitMQ durchgeführt wurde, sollte eine dynamische Kommunikation zwischen den sonst unabhängigen Anwendungen schaffen. Dies wurde auch ermöglicht und funktionierte zuverlässig, aber die spätere Untersuchung zur Containerorchestrierung bot ebenfalls gute Wege zur dynamischen Kommunikation zwischen den verschiedenen Anwendungen innerhalb des Clusters. Ein finales Verdikt hierzu sollte durch eine weitere Untersuchung in Betracht gezogen werden. Nichtsdestotrotz bot RabbitMQ, durch die Funktion eine Nachricht an viele Konsumenten auf einmal zu verteilen, einen Mehrwert, der im Sammeln von *Logging*-Daten weiter Verwendung finden kann.

Die Untersuchung zur Containervirtualisierung, die mit Docker durchgeführt wurde, sollte Anwendungen eine Umgebung zur Verfügung stellen, in der sie unabhängig von anderen Anwendungen und deren Abhängigkeiten ausgeführt werden können. Dies war ebenfalls erfolgreich, aber ebenfalls soll angemerkt sein, dass die Notwendigkeit einer Nachuntersuchung zur Fehlersuche und *Logging* bezüglich Containern in Betracht gezogen werden sollte, da dies hier nur in der Essenz betrachtet wurde.

Die Untersuchung zur Containerorchestrierung, die mit Kubernetes durchgeführt wurde, baute auf der vorherigen Untersuchung zur Containervirtualisierung auf und sollte eine Umgebung bieten, in der die zuvor untersuchten Container auf verschiedenen Systemen als ein System laufen können. Auch sollte untersucht werden, wie die einzelnen Container von Kubernetes überwacht und neu gestartet werden können. Dies wurde zufriedenstellend erreicht. Weitere Folgeuntersuchungen, die in Betracht gezogen werden könnten, wären hierzu ebenfalls zur Fehlersuche und *Logging*, aber auch wäre eine Untersuchung zur Skalierbarkeit denkbar.

Nun zur Betrachtung der Arbeit an Hand der drei gesetzten Begriffen der Flexibilität, Unabhängigkeit und Robustheit. Die Flexibilität wird auf vielerlei Weise erreicht. Durch die Möglichkeiten über einen Message Broker oder auch über das in Kubernetes enthaltene *Routing* gibt uns die Freiheit, Anwendungen auszutauschen, ohne mit direkten Adressen zu tun zu haben, das heißt, wenn eine neue Anwendung mit neuer Adresse startet, wird diese trotzdem wie zuvor erreicht.

Die Unabhängigkeit wird durch die Containervirtualisierung erreicht, die den An-

wendungen eigene Umgebung bietet, in der jede Anwendung durch den Container, in dem sie läuft, abgeschirmt von äußeren Einflüssen ist.

Die Robustheit wird durch Kubernetes erreicht. Kubernetes ist in der Lage, durch die *livenessProbe* und die *readinessProbe* zu erkennen, ob ein Container noch wie gewünscht antwortet und startet diesen neu, falls dies nicht der Fall sein sollte. Somit können Container nicht nur bei einem Absturz, sondern auch bei internen Fehlern automatisch neu gestartet werden.

Somit wurden alle drei Begriffe erfüllt und damit das Forschungsprojekt erfolgreich abgeschlossen.

Literatur

[1] „AMQP 0-9-1 Model Explained“. In: *https://www.rabbitmq.com/tutorials/amqp-concepts.html* (25.09.2020).

[2] „Charts“. In: *https://helm.sh/docs/topics/charts/* (25.09.2020).

[3] „Compose file version 3 reference“. In: *https://docs.docker.com/compose/compose-file/* (25.09.2020).

[4] „Configure Liveness, Readiness and Startup Probes“. In: *https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/* (25.09.2020).

[5] „Consumer Acknowledgements and Publisher Confirms“. In: *https://www.rabbitmq.com/confirms.html* (25.09.2020).

[6] „DaemonSet“. In: *https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/* (25.09.2020).

[7] „Dead Letter Exchanges“. In: *https://www.rabbitmq.com/dlx.html* (25.09.2020).

[8] „Deploments“. In: *https://kubernetes.io/docs/concepts/workloads/controllers/deployment/* (25.09.2020).

[9] „Docker“. In: *https://www.docker.com/* (25.09.2020).

[10] „Dockerfile reference“. In: *https://docs.docker.com/engine/reference/builder/* (25.09.2020).

[11] IBM Cloud Education. „Message Brokers“. In: *https://www.ibm.com/cloud/learn/message-brokers* (18.08.2020).

[12] „FilialFinder“. In: *https://www.yellowmap.com/loesungen/filialfinder/* (25.09.2020).

[13] „Get started“. In: *https://docs.docker.com/get-started/* (25.09.2020).

[14] „Helm“. In: *https://helm.sh/* (25.09.2020).

[15] „Horizontal Pod Autoscaler“. In: *https://kubernetes.io/de/docs/tasks/run-application/horizontal-pod-autoscale/* (25.09.2020).

[16] „Horizontal Pod Autoscaler Walkthrough“. In: *https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/* (25.09.2020).

[17] „Integritätsprüfungen in ASP.NET Core“. In: *https://docs.microsoft.com/de-de/aspnet/core/host-and-deploy/health-checks?view=aspnetcore-3.1* (25.09.2020).

[18] „Konzepte“. In: *https://kubernetes.io/de/docs/concepts/* (25.09.2020).

[19] „Kubernetes“. In: *https://kubernetes.io/de/* (25.09.2020).

[20] „Kubernetes (k8s) erklärt“. In: *https://www.redhat.com/de/topics/containers/what-is-kubernetes* (25.09.2020).

[21] „Kubernetes Metrics Server“. In: *https://github.com/kubernetes-sigs/metrics-server* (25.09.2020).

[22] „Overview“. In: *https://docs.docker.com/get-started/overview/* (25.09.2020).

[23] „Overview of Docker Compose“. In: *https://docs.docker.com/compose/* (25.09.2020).

[24] „Pods“. In: *https://kubernetes.io/docs/concepts/workloads/pods/* (25.09.2020).

[25] „Protocol Extensions“. In: *https://www.rabbitmq.com/extensions.html/* (18.08.2020).

[26] „RabbitMQ“. In: *https://www.rabbitmq.com/* (25.09.2020).

[27] „Remote procedure call (RPC)“. In: *https://www.rabbitmq.com/tutorials/tutorial-six-dotnet.html* (25.09.2020).

[28] „ReplicaSet“. In: *https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/* (25.09.2020).

[29] „Service“. In: *https://kubernetes.io/docs/concepts/services-networking/service/* (25.09.2020).

[30] „SmartMaps“. In: *https://www.smartmaps.net/* (25.09.2020).

[31] „StatefulSets“. In: *https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/* (25.09.2020).

[32] „Time-To-Live and Expiration“. In: *https://www.rabbitmq.com/ttl.html* (25.09.2020).

[33] „Vertical Pod Autoscaler“. In: *https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler* (25.09.2020).

[34] „Was ist Kubernetes?“ In: *https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/* (25.09.2020).

[35] „YellowMap“. In: *https://www.yellowmap.com/* (25.09.2020).