Duale Hochschule Baden-Württemberg

Stuttgart Campus Horb

Konzeption und prototypische Implementierung einer Search Engine in einer Microservice-Architektur

T3300 - Bachelorarbeit

---

eingereicht von:	Moris Kotsch
Matrikelnummer:	1317681
Kurs:	TINF2018
Studiengang:	Informatik
Hochschule:	DHBW Stuttgart Campus Horb
Ausbildungsfirma:	ENISCO by FORCAM GmbH
Ausbildungsleiterin:	Dipl.-Betriebsw. Angela Rasch
Unternehmen der Bachelorarbeit:	ENISCO by FORCAM GmbH
Betrieblicher Betreuer:	Dipl.-Ing. (FH) Franziska Simmank
Gutachter der DHBW:	Prof. Dr. phil. Antonius van Hoof
Bearbeitungszeitraum:	07.06.2021 - 31.08.2021

Freudenstadt, 6. Juli 2021

Sperrvermerk

Die vorliegende Praxisarbeit zum Thema „Konzeption und prototypische Implementierung einer Search Engine in einer Microservice-Architektur“ beinhaltet interne vertrauliche Informationen der Firma ENISCO by FORCAM GmbH. Die Weitergabe des Inhaltes der Arbeit und eventuell beiliegender Zeichnungen und Daten im Gesamten oder in Teilen ist untersagt. Es dürfen keinerlei Kopien oder Abschriften - auch in digitaler Form - gefertigt werden. Ausnahmen bedürfen der schriftlichen Genehmigung der Firma ENISCO by FORCAM GmbH.

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich:

1. dass ich meine Praxisarbeit mit dem Thema **„Konzeption und prototypische Implementierung einer Search Engine in einer Microservice-Architektur“** ohne fremde Hilfe angefertigt habe;
2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe;
3. dass ich meine Praxisarbeit bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Freudenstadt, 6. Juli 2021

Moris Kotsch

Zusammenfassung

ToDo

Abstract

ToDo

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

E-MES	Enisco Manufacturing Execution System
MCC	Manufacturing Control Cloud
MES	Manufacturing Execution System
QoS	Quality of Service
REST	Representational State Transfer

Kapitel 1

1 Einleitung

Im folgenden Kapitel wird die betreuende Firma mit dem dazugehörigem Kernprodukt, die Aufgabenstellung und das geplante Vorgehen erläutert. Um das Studieren der Arbeit zu erleichtern, wird ein Überblick über den Aufbau der Arbeit gegeben.

1.1 ENisco und E-MES

Die vorliegende Arbeit wurde im Rahmen einer Bachelorarbeit bei der Firma Enisco verfasst. Enisco GmbH & Co. KG wurde 2015 als Tochtergesellschaft der Eisenmann SE gegründet und ist mittlerweile ein eigenständiges Unternehmen, welches unter dem Namen „Enisco by Forcam GmbH[1]“ agiert.

Als Kernprodukt vertreibt die Firma Enisco das Produktionsleitsystem „Enisco Manufacturing Execution System (E-MES)“. Dieses wird für die Überwachung und Steuerung von Produktionsanlagen eingesetzt. E-MES vernetzt dabei die Anlage horizontal, über den gesamten Fertigungsprozess, als auch vertikal, über alle Prozessebenen hinweg und bildet so ein System, welches zwischen Unternehmensebene (ERP, engl. für Enterprise Ressource Planning) und Steuerungsebene (PLC, engl. für Programmable Logic Controller) agiert. [Eni21]

E-MES ist dafür modular aufgebaut und besitzt als Basis ein Platform-Modul. Dieses beinhaltet alle Grundlagen und Schnittstellen für die Installation weiterer Module. Die zusätzlichen Module (engl. Add-Ons) ergänzen E-MES um bestimmte Funktionen und können kundenspezifisch installiert und konfiguriert werden.

1.2 Aufgabenstellung

Derzeit erfährt das aktuelle Produktionsleitsystem E-MES eine Neugestaltung. Dabei wird von einer monolithischen 3-Schichten-Architektur auf eine verteilte Microservice-Architektur gewechselt. Ein Wechsel der Architektur beruht auf dem Eintritt der Muttergesellschafft Forcam GmbH in die „Open Industry 4.0 Alliance“. Durch den

[1] Im Folgenden wird aus Gründen der Lesbarkeit auf die Rechtsform der Enisco by Forcam GmbH verzichtet

Zusammenschluss von mehreren Unternehmen aus dem Bereich „Industrie 4.0“, können einheitliche Schnittstellen definiert werden, um so die Interoperabilität zwischen den Softwarelösungen der beteiligten Firmen zu stärken [Ope21]. Um die benötigte Interoperabilität zu ermöglichen, setzen die beteiligten Unternehmen vermehrt auf Technologien, wie Docker und Kubernetes. Um nun auch die Neugestaltung von E-MES in diesem Umfeld anzubieten, wurde sich für eine verteilte Microservice-Architektur entschieden. Neben der Architektur wird auch der Produktname von „E-MES“ in den Produktnamen „Manufacturing Control Cloud (MCC)“ abgeändert.

Im Zuge der Neugestaltung von E-MES werden neue Funktionalitäten, wie die Suchfunktion integriert. Eine Suchfunktion in modernen Informationssystemen wird von den Benutzern als gewohnter Komfort wahrgenommen. Auftretende Problemstellungen können dabei mithilfe von Suchabfragen gelöst werden. Solch eine Funktion liefert dem Benutzer eine Liste mit Suchergebnissen, aus der der Benutzer den geeignetsten Treffer auswählen kann.

Für die Umsetzung einer Suchfunktion in einem Manufacturing Execution System (MES) muss festgelegt werden, welche Funktionen und Inhalte des MES von der Suchfunktion abgedeckt werden sollen. Neben der Suche nach Funktionalitäten des Systems, kann es auch hilfreich sein, nach bestimmten „Objekten“ innerhalb des Systems zu suchen. Solche Objekte können zum Beispiel in Form von eindeutigen Aufträgen oder Maschinen in einem MES vorkommen. Gibt der Benutzer die Kennung eines Objektes in das Suchfeld ein, sollen ihm alle Funktionen und Informationen bezüglich dieses Objektes angezeigt werden.

Neben der Anforderungsklärung bezüglich der Suchoptionen und der Granularität der Suchanfragen, gilt es auch eine Konzeption für die Integration einer Search Engine zu entwerfen. Hierbei sind die besonderen Anforderungen zu beachten, welche durch die Einführung der verteilten Microservice-Architektur entstanden sind. So ist zu klären, welche Strategie für die Datenaktualisierung zwischen einer Search Engine und den Datenhaltungsschichten der einzelnen Services den Anforderungen am besten entspricht. Anhand von selbstgewählten technischen und lizenzbezogenen Kriterien, sollen diesbezüglich Strategien und auch potentiell geeignete Search Engines verglichen werden. Bei der Konzeption für der Integration einer Search Engine sind monolithische Seiteneffekte, welche durch die Missachtung von Prinzipien der Microservice-Architektur entstehen könnten, zu vermeiden.

Das erstellte Konzept gilt es anschließend mithilfe einer Proof of Concept - Anwendung umzusetzen.

1.3 Vorgehensweise und Aufbau der Arbeit

Die Vorgehensweise und die schriftliche Ausarbeitung der vorliegenden Arbeit gliedert sich in drei Hauptteile. Als Vorarbeit für die eigentliche Bearbeitung werden in **Kapitel 2** die theoretischen Grundlagen über die verteilte Microservice-Architektur, den Search Engines und dem Entwicklungsstand von MCC beschrieben.

Im ersten Schritt wird definiert, mit welchem Suchumfang die Search Engine innerhalb von MCC nach Funktionen und Objekten agieren soll. Da zum Zeitpunkt der Erstellung dieser Arbeit noch keine produktreife Version von MCC existiert, wird sich an dem Produktumfang und den Funktionalitäten des aktuellen Produktes E-MES orientiert. Im **Kapitel 3** wird der Suchumfang für die Suchfunktionalität definiert. Hierbei wird bei E-MES analysiert, welche Objekte innerhalb des Systems „suchbar“ gemacht werden sollen.

Ein weiterer Schritt ist die Konzeption für die Integration einer Search Engine in MCC. Es werden hierbei in **Kapitel 4** verschiedene Search Engines anhand von technischen und lizenzbezogenen Kriterien miteinander verglichen. Ebenso werden verschiedene Möglichkeiten der Datenaktualisierung zwischen einer Search Engine und den Datenhaltungsschichten der einzelnen Services erläutert und anhand der erarbeiteten Entscheidungskriterien miteinander verglichen. Vorbereitend für die prototypische Umsetzung wird zusätzlich ein Gesamtkonzept erstellt.

Anschließend an die Konzeption folgt in **Kapitel 5** die prototypische Umsetzung anhand einer Proof of Concept - Anwendung. Hierfür wird zunächst festgelegt, welchen Umfang jene prototypische Umsetzung besitzen soll und ob bereits Softwareteile aus MCC verwendet werden können.

Abgeschlossen wird die Arbeit mit einem Fazit und einem Ausblick **(Kapitel 6)**.

Kapitel 2

2 Grundlagen

Um eine Konzeption und prototypische Umsetzung einer Search Engine in MCC zu ermöglichen, wird in folgendem Kapitel auf die architekturellen Hintergründe von MCC eingegangen. Hierfür werden zu Beginn allgemein gültige Architektur-Prinzipien erläutert, welche auch bei der Integration einer Search Engine berücksichtigt werden müssen und somit bei der Auswahl eines geeigneten Konzeptes von Bedeutung sind.

Aufbauend auf den Architektur-Prinzipien wird die verwendete Microservice-Architektur erläutert, wobei neben einer allgemeinen Einführung in die Architektur auch auf die Kommunikation von Microservices untereinander eingegangen wird. Um mögliche Fehlkonzeptionen zu vermeiden, werden häufig auftretende Anti-Pattern aufgezeigt, welche unter Umständen zu monolithischen Seiteneffekten führen können.

2.1 Software-Architekturen

> Die Software-Architektur eines Systems ist die Menge von Strukturen, die benötigt werden, um Entscheidungen über das System zu treffen, welche die Software-Elemente, die Relationen zwischen ihnen und die Eigenschaften von beiden betreffen. ~ Len Bass [BCK13, S. 4]

Wie aus der Definition von Len Bass zu entnehmen ist, beschreibt eine Software-Architektur die Eigenschaften und Beziehungen von Software-Bausteinen zueinander [BCK13, S. 4]. Ein Software-Baustein wird hierbei als eine Teil-Komponente der gesamten Software betrachtet und wird bei der Erstellung einer Architektur als elementarer Bestandteil angesehen. Dabei wird ein Software-Baustein nicht näher spezifiziert, sondern als Komponente betrachtet, dessen konkrete Implementierung für die Architektur nicht von Bedeutung ist. Der Fokus einer Software-Architektur liegt auf den Schnittstellen der Software-Bausteine, über welche die Bausteine miteinander kommunizieren können.

2.1.1 Architektur-Prinzipien

Für das Erstellen einer guten Software-Architektur wurden von Vogel [Vog09, S. 128-147] einige Grundprinzipien definiert. Diese Prinzipien sollten bei der Erstellung einer Software-Architektur beachtet werden: [Vog09, S. 128-147]

Lose Kopplung:

Der Kern einer Software-Architektur besteht aus der Beschreibung der Software-Bausteine eines Software-Systems und deren Interaktionen zueinander. Unter dem Begriff Kopplung versteht man hierbei die Beziehung unter den Software-Bausteinen einer Software-Architektur. Eine Kopplung charakterisiert demnach die Interaktionen der Software-Bausteine.

Eine starke Kopplung von Software-Bausteinen hat zur Folge, dass beim Verstehen und Ändern eines Software-Bausteines auch zwingend weitere Software-Bausteine verstanden und geändert werden müssen. Um jenes Problem zu umgehen, besagt das Prinzip der losen Kopplung, dass die Kopplung zwischen Software-Bausteinen möglichst niedrig gehalten werden sollen.

Um eine lose Kopplung in einer Architektur zu erreichen, ist die Einführung von Schnittstellenabstraktionen ein wichtiger Aspekt. Dabei werden die Implementierungsinformationen hinter den Schnittstellen verborgen. Durch die Begrenzung von Schnittstellenelementen und der Häufigkeit des Austauschs der Schnittstellenelemente, kann eine Kopplung von Software-Bausteinen kontrollierbar gemacht werden.

Hohe Kohäsion:

Im Gegensatz zur Kopplung, in welcher die Beziehungen zwischen Software-Bausteinen gemeint ist, versteht man unter dem Begriff Kohäsion die Abhängigkeiten innerhalb eines Software-Bausteins.

Beim Prinzip der hohen Kohäsion ist das Ziel die Abhängigkeiten innerhalb eines Software-Bausteins möglichst hoch zu gestalten. Wie bei der losen Kopplung geht es auch hier um die lokale Änderbarkeit und Verstehbarkeit von Software-Bausteinen.

Wie in Abbildung 2.1 zu erkennen, stehen Kopplung und Kohäsion normalerweise miteinander in einer Wechselbeziehung. Hierbei gilt, dass je höher die Kohäsion individueller Software-Bausteine einer Architektur ist, desto geringer ist die Kopplung zwischen den Software-Bausteinen. Schematisch ist dieser Zusammenhang in Abbildung 2.1 abgebildet, worin zu erkennen ist, das eine Gesamtstruktur mit einer hohen Kohäsion und einer losen Kopplung (rechte Seite) eine höhere Übersichtlichkeit besitzt.

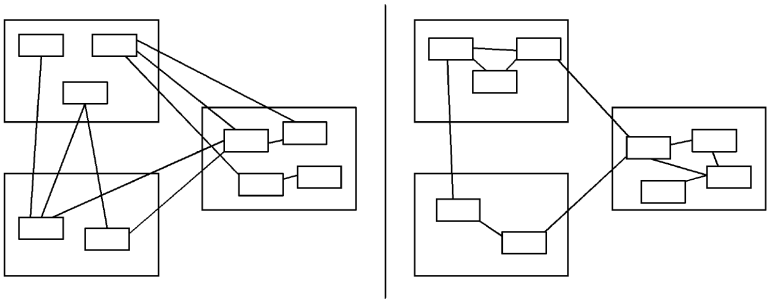

Abb. 2.1: Zusammenspiel von loser Kopplung und hoher Kohäsion - [Vog09]

Entwurf für Veränderung:

Durch den stetigen Wandel von Software-Systemen in Form von Anforderungen und Technologien, ist es von Vorteil solche Änderungen bereits in der Phase der architekturellen Konzeption zu berücksichtigen. Das Prinzip des Entwurfs für Veränderung (englisch: Design for Change) sieht nun vor, dass man vorhersehbare Änderungen architektonisch vorausplant. Dabei sollte man versuchen, die Architektur so zu entwerfen, dass man leicht mit den wahrscheinlichen Änderungen eines Software-Systems umgehen kann.

Separation of Concerns:

Abgeleitet von dem römischen Prinzip „Teile und herrsche“ wird beim Prinzip Separation of Concerns ausgesagt, dass ein Software-System in individuelle Software-Bausteine zerlegt werden soll.

Separation of Concerns unterstützt hierbei die Modularisierung eines Software-Systems. Es geht darum Teile eines Software-Systems zu identifizieren, welche für bestimmte Angelegenheiten, Aspekte und Aufgaben verantwortlich sind. Jene Teile werden dann als eigene Software-Bausteine gekapselt. Eine Zerteilung des Gesamtsystems in relativ unabhängige Einzelteile ermöglicht zudem noch die Verteilung von Verantwortlichkeiten für verschiedene Software-Bausteine und auch das parallele Arbeiten an dem Software-System durch mehrere Entwickler wird dadurch ermöglicht.

Durch das Aufteilen des Software-Systems in relativ unabhängige Software-Bausteine, werden auch die Prinzipien lose Kopplung und hohe Kohäsion begünstigt.

Information Hiding:

Das Prinzip Information Hiding sagt aus, dass man einem Klienten nur die für die Bearbeitung eines Problems notwendigen Informationen zeigen soll. Dies erleichtert die Gliederung und das Verständnis von komplexen Software-Systemen. Die restlichen Informationen sollen nach außen hin verborgen bleiben. Ermöglicht wird solch ein „geheim halten“ von Informationen durch die Bereitstellung von

definierten Schnittstellen, über welche nur bestimmte Informationen zu erreichen sind.

Abstraktion:

Als übergeordnetes Prinzip dient eine Abstraktion dazu, ein komplexes System verständlicher zu machen. Dazu werden wichtige Aspekte identifiziert und unwichtige Details vernachlässigt. Im Bereich der Software-Architektur gilt die Schnittstellenabstraktion als Teilprinzip der Abstraktion. Hierbei liegen die Schnittstellen im Fokus, welche für das Zustandekommen und die Qualität von Beziehungen verantwortlich sind.

Solch eine Schnittstellenabstraktion in einem Software-System ist eng verbunden mit dem Prinzip der losen Kopplung und dem Information Hiding. Ein Aspekt für den starken Zusammenhang zwischen der Abstraktion und dem Information Hiding ist die Portabilität von Software-Systemen. So sollte eine Architektur oder ihre Software-Bausteine auch in anderen Umgebungen verwendbar sein. Um solch eine Plattformunabhängigkeit sicherzustellen, werden Abstraktionen verwendet, die ein Information Hiding der Platform-Details leisten.

Modularität:

Das Modularitätsprinzip, welches bereits auch in den Beschreibungen der anderen Prinzipien vorkam, definiert die Aufteilung eines Systems in klar definierte Software-Bausteine mit abgegrenzten funktionalen Verantwortlichkeiten. Die Modularität ist dabei eine Kombination aus den Prinzipien Abstraktion, Separation of Concerns und Information Hiding, welche bei der Umsetzung der Prinzipien der losen Kopplung und der hohen Kohäsion kombiniert werden.

Auch für die spätere Konzeption einer Search Engine in einer Microservice-Architektur werden die eingeführten Prinzipien als Grundlage dienen.

2.1.2 Monolithische und verteilte Architekturen

Bei der Neugestaltung von E-MES wird von einer monolithischen 3-Schichten-Architektur auf eine verteilte Microservice-Architektur gewechselt.

Dabei besitzt die ehemalige E-MES Architektur eine monolithische Struktur. In einer solchen monolithische Architektur wird die gesamte Architektur in nur einem Software-Baustein zusammengefasst. Dadurch erfolgt keine explizite Gliederung in Teilsysteme und Architektur-Prinzipien, wie lose Kopplung und Separation of Concerns sind nur schwer umsetzbar [Vog09, S. 216]. Zu finden sind monolithische Architekturen oftmals in Altsystemen, welche oft über Jahrzehnte gewachsen sind. Aufgrund der mangelnden Modularisierung steigt die Kompliziertheit des Systems und eine Wartung und Anpassung des Quellcodes wird erschwert [Pro12a]. Ein weiterer Nachteil der mangelnden

Modularisierung ist die kaum mögliche nebenläufige Ausführung von Teilen des Systems auf verschiedenen Rechnern [Pro12a]. Somit kann eine horizontale Skalierung nicht ermöglicht werden und eine effiziente, lastverteilende Programmausführung ist nicht gegeben.

Die Architektur von MCC wird eine verteilte Struktur aufweisen. Hierbei werden Teile des Gesamtsystems in unterschiedliche Software-Bausteine aufgeteilt. Eine Modularisierung der Software ist dadurch möglich und Architekturen-Prinzipien, wie lose Kopplung und Separation of Concerns sind umsetzbar [Vog09]. Durch die strikte Aufteilung der Geschäftslogik kann auch die Komplexität aufgeteilt werden. Somit können die einzelnen Software-Bausteine mit wenig Aufwand angepasst oder erweitert werden. Durch die Modularisierung von verteilten Architekturen kann die Ausführung bestimmter Aufgaben auf redundanter Hardware nebenläufig erfolgen [Pro12b]. Durch jene horizontale Skalierung kann eine effiziente und lastverteilende Programmausführung erfolgen, welche auch zur Ausfallsicherheit des Gesamtsystems beiträgt [Pro12b].

2.2 Microservice-Architektur

Als eine Architektur mit einer verteilten Struktur, wird die Software bei der Neugestaltung des MES der Firma Enisco, auf einer Microservice-Architektur aufgebaut.

Die Kernelemente dieser Architektur sind die Microservices, welche der Modularisierung der Software dienen. Somit ist eine Aufteilung des Gesamtsystems in verschiedene Software-Bausteine möglich. Ein Software-Baustein stellt dabei jeweils eine Funktionalität des Gesamtsystems dar. Im Gegensatz zu einer monolithischen Architektur läuft das Gesamtsystem nicht innerhalb eines Prozesses, sondern auf verschiedenen Prozessen. Dabei wird jedem Software-Baustein ein eigener Prozess zugeordnet. Jene Prozesse können nun nahezu beliebig auf verschiedene Rechner verteilt und durch Replizierung ausfallsicher gemacht werden. [Gar18]

Neben den Vorteilen der horizontalen Skalierung, ergeben sich aus der Aufteilung des Gesamtsystems in unterschiedliche Software-Bausteine auch Auswirkungen auf die Entwicklungsorganisation. So wird beim Umgang mit Microservices nach der Unix-Philosophie von Ken Thompson „Do one thing and do ot well“ [ION21] gearbeitet. Durch die Modularität von Microservices untereinander, können die Microservices von unterschiedlichen Entwicklerteams unabhängig voneinander entwickelt werden. Durch die Abstraktion der Microservices, können diese mit unterschiedlichen Technologien und Programmiersprachen implementiert werden. Auch der Datenhaushalt kann von jedem Microservice separat verwaltet werden. Zudem wird die Einarbeitung eines Entwicklers in die Codebasis reduziert, da durch die Aufteilung weniger Code verstanden werden muss.

Die Microservice-Architektur berücksichtigt die Architektur-Prinzipien Separation of Concerns, Information Hiding und Modularität und gewährleistet somit eine lose Kopplung zwischen den Microservices. Innerhalb der Microservices entsteht dadurch eine hohe Kohäsion.

Da die jeweiligen Microservices repliziert auf verschiedenen Rechnern laufen können, ist die Kommunikation zwischen den Microservices schwieriger als bei einem monolithischen System. Auf die Kommunikation zwischen Microservices wird in Unterabschnitt 2.2.1 näher eingegangen.

Eine Herausforderung bei der Konzeption einer Microservice-Architektur ist die Vermeidung von Abhängigkeiten, welche eine lose Kopplung der Microservices verhindern würden. Um solche monolithischen Seiteneffekte zu vermeiden, werden in Unterabschnitt 2.2.2 die häufigsten Microservice-Anti-Pattern aufgezeigt.

2.2.1 Kommunikation zwischen Microservices

Auch wenn das Ziel einer Microservice-Architektur ist, dass einzelne Funktionalitäten des Gesamtsystems in getrennte Microservices gekapselt werden, müssen jene miteinander kommunizieren. Aufgrund der Modularität können die Microservices horizontal skaliert werden und auf verschiedenen Rechnern betrieben werden. Dies erhöht die Komplexität bei der Kommunikation der Microservices untereinander. [Mic19]

Bei der Wahl der Kommunikation zwischen Microservices kann zwischen einer synchronen und asynchronen Kommunikation entschieden werden.

Synchrone Kommunikation

Bei der synchronen Kommunikation handelt es sich um eine eins-zu-eins Kommunikation, bei der eine Anfrage geschickt und auf eine Antwort gewartet wird. Klassischerweise erfolgt die Kommunikation über HTTP mit einer REST-Schnittstelle. Representational State Transfer (REST) ist hierbei eine Spezifikation, wie eine über HTTP kommunizierende API konzipiert werden soll [Mic19]. Eine solche API sollte demnach vordefinierte HTTP-Methoden implementiert haben. Unter anderem sind das Methoden wie GET, POST, PUT und DELETE. Bei einer GET-Anfrage werden hierbei Ressourcen angefragt. Möchte man einen neuen Datensatz übermitteln wird die POST-Methode verwendet und wenn man einen bestehenden Datensatz ändern möchte, verwendet man die PUT-Methode. Mit der DELETE-Methode kann man einen expliziten Datensatz löschen. Jeder Datensatz bekommt hierbei einen eigenen Endpunkt und die jeweiligen Anfragen können mit URL-Parametern und Query-Parametern spezifiziert werden. Das Standard-Datenformat bei REST ist JSON.

Bei der Kommunikation von zwei Microservices über die jeweiligen REST-Schnittstellen, muss die Adresse des anderen Microservice bekannt sein. Durch die Verteilung der Microservices auf unterschiedliche Rechner in Folge einer horizontalen Skalierung, kann es während dem Betrieb vorkommen, dass einzelne Microservices auf zum Beispiel Rechner A gestoppt und auf Rechner B wieder gestartet werden. Dadurch ändern sich auch die Adressen der Microservices. Da die Verwaltung der Adressen ab einer Vielzahl an Microservices nicht mehr trivial ist, wird in einer Microservice-Architektur eine sogenannte Service-Discovery eingesetzt. Die Service-Discovery ist eine Software, bei der sich alle neuen Microservice registrieren. Bei einem REST-Aufruf wird dann zuerst eine Liste mit allen verfügbaren Adressen abgerufen.

Wie in Abbildung 2.2 dargestellt ist, wird bei einer Anfrage eines Services, jene Anfrage zunächst an einen Router-Service geleitet. Nach dem Abfragen der Adresse mithilfe der Service-Discovery wird gezielt der entsprechende Service beziehungsweise dessen Instanz mit einer REST-Anfrage angesprochen.

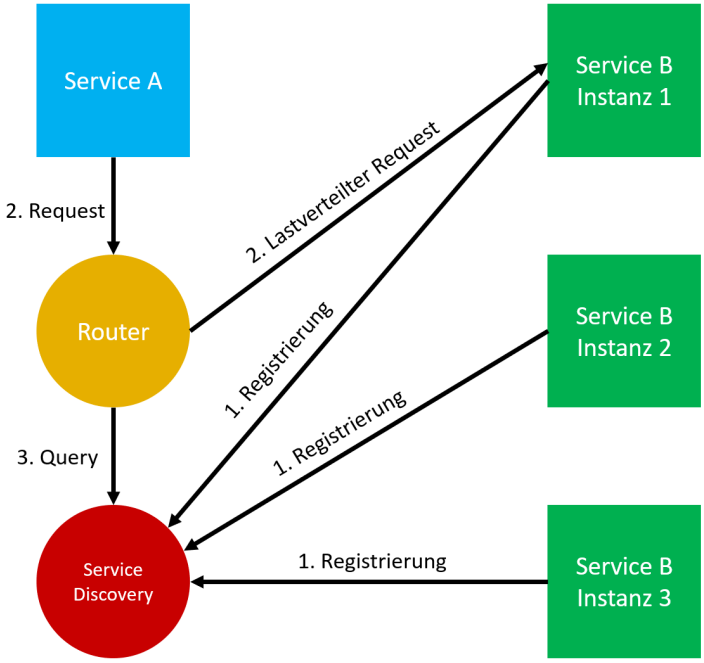

Abb. 2.2: Ablauf einer Anfrage mit der Service-Discovery - in Anlehnung an [Mic19]

Asynchrone Kommunikation

Anders als bei der synchronen Kommunikation wird bei der asynchronen Kommunikation nicht auf jede Anfrage eine Antwort erwartet. Kern einer asynchronen Kommunikation ist ein zentraler Nachrichtenkanal, auf dem Nachrichten ausgetauscht werden. Ein Service schickt Nachrichten an solch einen Nachrichtenkanal, auch Message Broker genannt, und wenn ein anderer Service an der Nachricht interessiert ist, kann er diese konsumieren. Eine Nachricht kann dabei auch von mehreren Services konsumiert werden.

Zu den bekanntesten Message Brokern zählen RabbitMQ und Apache Kafka. Während RabbitMQ leichtgewichtiger ist als Apache Kafka, hat letzteres mehr Features und weist eine größere Stabilität auf. Die Verantwortlichkeiten innerhalb des Systems werden bei der Verwendung eines Message Brokers umgedreht. Ein Service wird nun nicht mehr explizit von einem anderen Service aufgerufen, sondern schickt seine Nachrichten an den Message Broker, ohne zu wissen, wer diese Nachrichten konsumiert. Sender und Empfänger sind somit lose gekoppelt.

Zur Einteilung der Nachrichten werden in Message Brokern sogenannte Message Queues verwendet. Über eine Message Queue, welche einen bestimmten Topic besitzt, können Service dann entweder Nachrichten senden oder sich als Konsumenten an diesem Topic registrieren. In Abbildung 2.3 ist die Funktionsweise von Apache Kafka abgebildet. Durch den Publish/Subscribe-Mechanismus kann Service A in diesem Beispiel sowohl Publisher von zwei Topics sein, als auch Konsument von einem Topic.

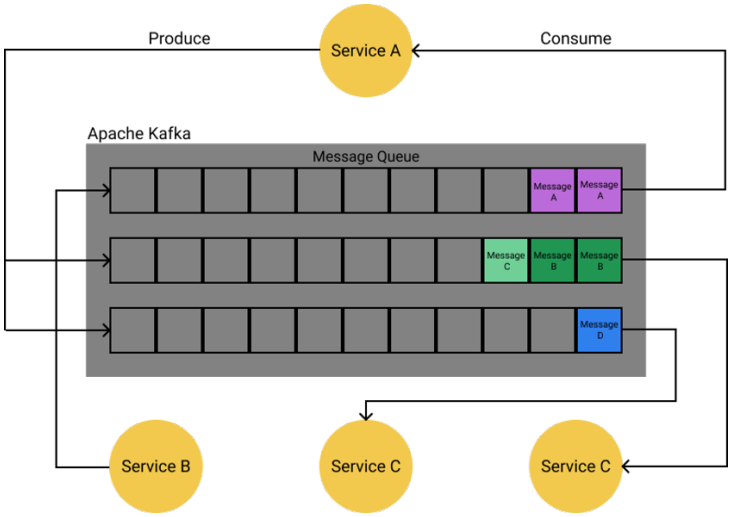

Abb. 2.3: Funktion eines Message Brokers - [Mic19]

Die Nachrichten werden in der Message Queue zwischengespeichert, bis die registrierten Konsumenten diese Nachrichten gelesen haben. Wenn ein Service sich an einer Message Queue registriert, können verschiedene Servicequalitäten für die Übertragung der Nachrichten angegeben werden. Folgende drei „Quality of Service (QoS)“ können angegeben werden: [Apa21]

QoS 0 „At most once“:

Eine Nachricht wird genau einmal versendet. Nach dem Prinzip „fire and forget“ wird keine Rückbestätigung erwartet. Verwendet wird diese Servicequalität meist für Nachrichten, welche mit einer hohen Frequenz versendet werden. So ist es für viele Sensoren in einer IoT-Umgebung nicht erforderlich, dass jeder gemessene Wert erfolgreich übermittelt wird.

QoS 1 „At least once“:

Es wird sichergestellt, dass eine Nachricht genau einmal beim Empfänger ankommt. Hierzu werden die Nachrichten beim Sender solange zwischengespeichert, bis vom Empfänger eine Bestätigung (PUBACK) empfangen wird. Wird über eine gewisse Zeitspanne keine Bestätigung empfangen, wird die Nachricht erneut versendet. Hierbei kann es vorkommen, dass ein Empfänger eine Nachricht mehrfach erhält. [Flo17]

QoS 2 „Exactly once“:

Eine Nachricht wird genau einmal zugesendet. Hierbei wird ein 4-Wege-Handshake zwischen Sender und Empfänger versendet. Durch diesen Ablauf ist eine sichere Übertragung der Nachricht gewährleistet. Durch den großen Overhead ist jedoch diese Servicequalität die langsamste Übertragungsmöglichkeit.

2.2.2 Microservice - Anti-Pattern

Um die Vorteile einer Microservice-Architektur im Bezug zur losen Kopplung und der damit eingehenden Unabhängigkeit der Microservices bezüglich Wartbarkeit und Erweiterbarkeit zu gewährleisten, ist es von Nöten die möglichen Anti-Pattern einer Microservice-Architektur zu kennen. Mit hilfe von Anti-Pattern können nach Roth und Hafner [SM19] wiederkehrende Fehler bei der Softwareentwicklung identifiziert und in generalisierter Form dokumentiert werden. So können Anti-Pattern dabei helfen häufig auftretende Fehler von vornherein zu vermeiden.

In einer Studie von Tighilt et al. [TAM+20] wurden 27 Paper und 67 Open-Source Anwendungen, welche alle Bezug zur Microservice-Architektur haben, untersucht. Die Ergebnisse der Studie [TAM+20] ergeben eine Sammlung von häufig aufgetretenen Anti-Pattern, welche durch Tighilt et al. in vier Kategorien unterteilt wurden. Folgend werden aus jeder Kategorie zwei Anti-Pattern näher erläutert.

Design Anti-Pattern

In diese Kategorie gehören Anti-Pattern, welche bereits in der architektonischen Entwurfsphase einer Microservice-Architektur zu Fehlern führen. Beispiele sind hierfür das „Falscher Schnitt“ - Anti-Pattern und das „Zyklische Abhängigkeiten“ - Anti-Pattern.

Falscher Schnitt:

Beim „Falscher Schnitt“ - Anti-Pattern, wird die Aufteilung des Gesamtsystems in unterschiedliche Microservices nicht nach Geschäftsfunktionen, sondern nach technischen Aspekten durchgeführt. Ähnlich wie bei einer monolithischen 3-Schichten-Architektur übernehmen dann ein Teil der Microservices die Präsentationsschicht, andere Microservices wiederum bedienen die Geschäftsschicht und die anderen bedienen die Datenzugriffsschicht. Durch dieses Vorgehen verliert die

Architektur die lose Kopplung zwischen den Microservices und die Modularität geht verloren. Auch müssen bei einer Änderung einer Geschäftslogik mehrere Microservice abgeändert werden, was zu einem Mehraufwand führt.

Verhindert werden kann dieses Anti-Pattern indem man die Trennung des Gesamtsystems anhand der Geschäftsfunktionen durchführt. Jedes Microservice sollte von einem Team implementiert und gewartet werden können. Die Abhängigkeiten zu anderen Microservices sollten so gering, wie möglich ausfallen, um eine lose Kopplung und die Modularität der Architektur zu gewährleisten.

Zyklische Abhängigkeiten:

Ein weiterer Fehler ist die Missachtung der losen Kopplung, indem man mit zyklischen Abhängigkeiten eine zu große Abhängigkeit mit anderen Microservices verursacht. Bei sowohl der Implementierung als auch der Wartung solcher Microservices sind die abhängigen Microservices auch zu berücksichtigen. Dies erhöht die Komplexität und die Wahrscheinlichkeit, dass Fehler auftreten.

Eine zyklische Abhängigkeit zwischen solchen Microservices kann verhindert werden, indem man stark voneinander abhängige Microservices in einen gemeinsamen Microservice zusammenfasst.

Implementation Anti-Pattern

Implementation Anti-Pattern entstehen aus der Art und Weise, wie Microservices implementiert werden. Beispiele sind hierfür das „Gemeinsame Bibliotheken“ - Anti-Pattern und das „Festcodierte Endpunkte“ - Anti-Pattern.

Gemeinsame Bibliotheken:

Bei der Verwendung von gemeinsam benutzen Bibliotheken entstehen starke Abhängigkeiten und die Bibliotheken stellen einen sogenannten „Single point of failure“ dar.

Um dieses Problem zu umgehen, kann man jedem Microservice seine eigene Bibliothek zur Verfügung stellen. Dieser Ansatz wiederum führt zu einer Komplexität bei der Wartbarkeit der Microservice. Ein weiterer Ansatz wäre es, die Bibliotheken durch einen Bibliothek-Service zu abstrahieren. Die Microservices bleiben lose gekoppelt und die Komplexität bei der Wartung bleibt überschaubar.

Festcodierte Endpunkte:

Bei der synchronen Kommunikation zwischen Microservices werden in der Regel Anfragen über REST APIs gestellt. Hierfür müssen die IP-Adresse und der Port des anderen Microservices bekannt sein. Ein Ansatz für schnelle Laufzeiten und eine schnelle Implementierung ist es, die Endpunkte fest im Quellcode zu hinterlegen. Jener Ansatz ist jedoch bei einer größeren Anzahl von Microservices

nicht mehr wartbar. Sobald sich ein Endpunkt eines Microservice ändert, müssen alle abhängigen Microservices geändert und neu deployed werden.

Abhilfe bringt die Einführung einer Service-Discovery (siehe Abschnitt 2.2.1).

Deployment Anti-Pattern

Auch in der Bereitstellungsphase einer Microservice-Architektur können Fehler auftreten. Jene Anti-Pattern werden Deployment Anti-Pattern genannt. Beispiele sind hierfür das „Manuelle Konfiguration“ - Anti-Pattern und das „Zeitüberschreitungen“ - Anti-Pattern.

Manuelle Konfiguration:

In einer Microservice-Architektur müssen aufgrund der Verteilung der Microservices auf unterschiedliche Rechner verschiedene Konfigurationen durchgeführt werden. Eine manuelle Konfiguration von jedem Microservice führt zu unzähligen Konfigurationsdateien und Abhängigkeiten zu Umgebungsvariablen. Mit wachsender Anzahl von Microservices führt die manuelle Konfiguration zu einer ansteigenden Komplexität bei der Wartung der Konfigurationen.

Vermieden werden kann das Anti-Pattern durch die Einführung eines Konfigurations-Servers. Durch geeignete Konfigurationsmanagement-Tools können sämtliche Konfiguration zentral verwaltet werden.

Zeitüberschreitungen:

In einer verteilten Microservice-Architektur kann es vorkommen, dass einzelne Services ausfallen und nicht mehr zur Verfügung stehen. Hat nun ein anderer Microservice eine Anfrage über eine REST API gestellt, kann es vorkommen, dass unter Umständen gar keine Antwort kommt. Hierfür kann der Entwickler Timeouts festlegen, nach welchen einem anfragenden Service mitgeteilt wird, dass ein anderer Service nicht erreichbar ist. Den richtigen Wert für solch einen Timeout zu wählen, hängt von der Art des Service ab. Ein zu kurzer Timeout kann dazu führen, dass langsame Service frühzeitig beendet werden und bei einer zu langen Wahl des Timeouts kann es vorkommen, dass der Endnutzer zu lange warten muss, bis ihm signalisiert wird, dass der gewünschte Service nicht verfügbar ist.

Um dem entgegen zu wirken verwendet man in einer Microservice-Architektur üblicherweise einen Circuit Breaker. Ein Circuit Breaker überwacht hierbei alle Instanzen einzelner Microservices und stellt fest, ob einzelne Timeouts bei der Kommunikation mit der jeweiligen Instanz überschritten wurden. Da solch ein Circuit Breaker als Proxy für alle Kommunikationskanäle verwendet wird, werden alle Anfragen an einen ausgefallenen Service mit einer Fehlermeldung beantwortet. Die Anfragen werden solange durch den Circuit Breaker blockiert, bis der

ausgefallene Service wieder zur Verfügung steht. Auch der Timeout des Circuit Breaker muss vom Entwickler festgelegt werden.

Monitoring Anti-Pattern

Zu dem Betrieb einer Microservice-Architektur gehört auch die Überwachung der Änderungen im System. Hierbei kann es auch zu häufig auftretenden Anti-Pattern kommen. Beispiele sind hierfür das „Kein Gesundheitscheck“ - Anti-Pattern und das „Lokale Protokollierung“ - Anti-Pattern.

Kein Gesundheitscheck:

In der Natur der Microservices kann es vorkommen, dass einige Services über gewisse Zeitspannen nicht erreichbar sind. Um ein kompletten Systemausfall zu vermeiden, dienen Mechanismen, wie der Circuit Breaker dazu die anfragenden Services, über das Überschreiten von Timeouts, zu benachrichtigen.

Mit der Integration von Gesundheitschecks können solche Ausfälle von einzelnen Services im Vorraus erkannt und abhängige Microservices darüber informiert werden, dass das Senden von Anfragen zu unterlassen ist. Für solche Gesundheitschecks werden API-Endpoints angeboten, welche periodisch abgefragt werden können.

Lokale Protokollierung:

Um eine spätere Fehlerdiagnose zu ermöglichen, speichern Microservices einige Informationen über Änderungen in Log-Dateien. Ein häufiger Fehler ist, dass solch eine Protokollierung lokal in den Datenbanken der einzelnen Microservices gespeichert werden. Die Herausforderung liegt nun bei der Analyse und Abfrage von Logs aus verschiedenen Microservices.

Mit einem verteilten Logging-Mechanismus können die Logs der unterschiedlichen Microservices in einem zentralen Speicherort gespeichert werden. Durch eine einheitliche Formatierung der Log-Einträge kann auch deren Abfrage und Analyse vereinfacht werden.

Auch bei der Integration einer Search Engine, in solch eine Microservice-Architektur, gilt es die vorgestellten Anti-Pattern zu beachten, um so monolithische Seiteneffekte zu vermeiden.

Kapitel 3

3 Umfang der Suchfunktionalität

3.1 Volltextsuche in modernen Informationssystemen

3.2 Produktumfang von E-MES

3.3 SCADA

3.3.1 Elektrische/Technische Sicht

3.3.2 Asset Sicht

3.3.3 Suchumfang im SCADA-Umfeld

3.4 PCS

3.4.1 Suchumfang im PCS-Umfeld

Kapitel 4

4 Konzeption

Inhalt

4.1 Konzeptionskriterien

Inhalt

4.2 Auswahl einer Search Engine

Inhalt

4.2.1 Apache Solr

Inhalt

4.2.2 Elasticsearch

Inhalt

4.2.3 Vergleich

Inhalt

4.3 Datenaktualisierung zwischen Microservices und Search Engine

Inhalt

4.4 Gesamtkonzept

Inhalt

Kapitel 5

5 Prototypische Umsetzung

Inhalt

5.1 Client

Inhalt

5.2 Search Service

Inhalt

5.3 Anbindung an Apache Kafka

Inhalt

Kapitel 6

6 Fazit und Ausblick

Inhalt

Literaturverzeichnis

[Apa21] Apache Software Foundation ; Apache Software Foundation (Hrsg.): *Kafka 2.8 Documentation*. https://kafka.apache.org/documentation/. 2021

[BCK13] Bass, Len ; Clements, Paul ; Kazman, Rick: *Software architecture in practice*. Third edition, fifth printing. Upper Saddle River, NJ : Addison-Wesley, 2013 (SEI series in software engineering). – ISBN 9780132942775

[Eni21] Enisco by Forcam GmbH ; Enisco by Forcam GmbH (Hrsg.): *Homepage: Das Produktionsleitsystem E-MES*. https://enisco.com/. 2021

[Flo17] Florian Raschbichler ; Informatik Aktuell (Hrsg.): *MQTT - Leitfaden zum Protokoll für das Internet der Dinge*. https://www.informatik-aktuell.de/betrieb/netzwerke/mqtt-leitfaden-zum-protokoll-fuer-das-internet-der-dinge.html. 2017

[Gar18] Gary Calcott ; NetMediaEurope Deutschland GmbH (Hrsg.): *Microservices vs. Monolithische Architekturen: ein Leitfaden*. https://www.silicon.de/41666855/microservices-vs-monolithische-architekturen-ein-leitfaden. 2018

[ION21] IONOS SE ; IONOS SE (Hrsg.): *Microservice-Architectures: Mehr als die Summe ihrer Teile?* https://www.ionos.de/digitalguide/websites/web-entwicklung/microservice-architecture-so-funktionieren-microservices/. 2021

[Mic19] Michael Schwab ; Host Europe GmbH (Hrsg.): *Microservices — Grundlagen und Technologien von verteilter Architektur*. https://www.hosteurope.de/blog/microservices-grundlagen-und-technologien-von-verteilter-architektur/. 2019

[Ope21] Open Industry 4.0 Alliance ; Open Industry 4.0 Alliance

(Hrsg.): *Über Uns*. https://openindustry4.com/de/About-Us.html. 2021

[Pro12a] Prof. Dr. Andreas Fink ; Universität Potsdam (Hrsg.): *Monolithisches IT-System*. https://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/is-management/Systementwicklung/Softwarearchitektur/Architekturparadigmen/Monolithisches-IT-System. 2012

[Pro12b] Prof. Dr. Andreas Fink ; Universität Potsdam (Hrsg.): *Verteiltes IT-System*. https://www.enzyklopaedie-der-wirtschaftsinformatik.de/wi-enzyklopaedie/lexikon/is-management/Systementwicklung/Softwarearchitektur/Architekturparadigmen/Verteiltes-IT-System/index.html. 2012

[SM19] Stephan Roth ; Martina Hafner ; Vogel Communications Group GmbH & Co. KG (Hrsg.): *Anti-Patterns: Wiederkehrende Entwicklerfehler erkennen und vermeiden*. https://www.embedded-software-engineering.de/anti-patterns-wiederkehrende-entwicklerfehler-erkennen-und-vermeiden-a-67 2019

[TAM+20] Tighilt, Rafik ; Abdellatif, Manel ; Moha, Naouel ; Mili, Hafedh ; Boussaidi, Ghizlane E. ; Privat, Jean ; Guéhéneuc, Yann-Gaël: On the Study of Microservices Antipatterns. In: *Proceedings of the European Conference on Pattern Languages of Programs 2020*. New York, NY, USA : ACM, 072020. – ISBN 9781450377690, S. 1–13

[Vog09] Vogel, Oliver: *Software-Architektur: Grundlagen - Konzepte - Praxis*. 2. Aufl. Heidelberg : Spektrum Akad. Verl., 2009 http://deposit.d-nb.de/cgi-bin/dokserv?id=3123606&prov=M&dok_var=1&dok_ext=htm. – ISBN 978–3–8274–1933–0