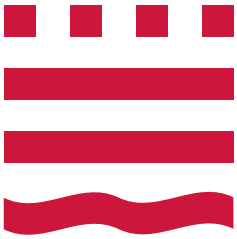


Technische Hochschule Brandenburg
University of Applied Sciences

BACHELORARBEIT

Serverless Architekturen für konventionelle Webanwendungen

Vorgelegt von: Dragoljub Milasinovic
Matrikelnummer: 20140076

am: 25. September 2017

zum
Erlangen des akademischen Grades

BACHELOR OF SCIENCE
(B.Sc.)

Erstbetreuer: Prof. Dr.-Ing. Schafföner
Zweitbetreuer: Jonas Brüstel, M.Sc.

Inhaltsverzeichnis

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die von mir eingereichte Bachelorarbeit selbstständig verfasst, ausschließlich die angegebenen Hilfsmittel benutzt und sowohl wörtliche, als auch sinngemäße entlehnte Stellen als solche kenntlich gemacht habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Brandenburg an der Havel, 25. September 2017

Dragoljub Milasinovic

Zusammenfassung

Die vorliegende Bachelorarbeit untersucht die Flexibilität bei der Gestaltung von Webanwendungen ausschließlich mit Verwendung von Serverless Technologien. Dazu wurde eine Linked Data Driven Webanwendung anhand einer Auswahl von Serverless Entwurfsmuster implementiert. Die Ergebnisse zeigten sowohl Schwierigkeiten bei Sitzungsverwaltung der Nutzern als auch eine große Freiheit bei der architektonischen Komposition von Diensten. Die empirische Bearbeitung erfolgte mit ER-Modellen, Ontologien, Protégé, dem europäischen Kompetenzmodell, SparQL, AWS Lambda, API Gateway, S3 und die Theoretische mit Architekturentwurfsmustern, OpenLambda, DynamoDB, SQS, SES, Kinesis Streams und SNS. Aktuelle Problemlagen im Bereich der Softwarearchitektur, des Projektmanagements, der Wirtschaftsinformatik und der Softwareentwicklung werden aufgegriffen.

Abstract

This bachelor thesis was aiming at the analysis of structural flexibility, in the case of a web application that exclusively consists of serverless architectures. An exemplary linked data driven web application was implemented. On the one hand, the results suggested that there are difficulties regarding management of clients' sessions, on the other hand, it provides great freedom of choice within the architectural design of services. The empirical approach dealt with ER-Models, Ontologies, Protégé, the Europe's competency model, SparQL, AWS Lambda, API Gateway, S3 and on a theoretic level with the architectural design patterns, OpenLambda, DynamoDB, SQS, SES, Kinesis Streams and SNS. Current questions of software architecture, project management, economic computer sciences and software development are addressed.

1 Einleitung

Ideen entstehen, verändern sich, werden im Laufe der Zeit vergessen, manchmal begeistern sie. Ihnen Form und Inhalt zu geben, also sie umzusetzen, ist die Voraussetzung, um nachzuvollziehen ob die ursprüngliche Idee wirklich ausgebaut und verstanden worden ist.

Die Technik kann als Medium für den Ausdruck solcher Ideen eingesetzt werden. Diese kann so komplex werden, dass sie eine Barriere in Form eines Wissensmonopols darstellt, die hinderlich für die Umsetzung neuer Ideen ist.

Die Faktoren am Anfang einer technologischen Umsetzung einer Idee sind: der konzeptionelle Beweis, die Vorlaufzeit oder Produkteinführungszeit (Time-To-Market), die Personalkosten und Mangel von Fertigkeiten, die technischen und technologischen Details, sowie die Rentabilität.

Ein Zeichen für die Existenz dieser Komplexität im Rahmen des Cloudcomputings ist die Entstehung neuer Technologien für die Vereinfachung der Entwicklungsprozesse eines Projekts.

Je mehr Anforderungen auf ein System z.B. Webanwendung zukommen, desto komplexer wird es. Je mehr Softwarekomponenten, desto mehr Verwaltungsaufwand mittels Load Balancing, Messaging usw. entsteht. Je mehr Dynamik, desto schwieriger ihre Integration und Skalierung. [HW04]

Mittels des Serveless Architekturstils versuchen die Cloudanbieter diesen Verwaltungsaufwand zu umgehen, die Skalierung zu vereinfachen und Integrationsschwierigkeiten zu lösen.

Um die Umsetzungsvorgänge einer Webanwendung möglichst simpel zu halten, werden in der vorliegenden Arbeit die Serverless Architekturen für konventionelle Webanwendungen untersucht.

1.1 Vorgehen

Auf dem Weg zur technologischen Umsetzung einer neuen Idee treten unbekannte Komplikationen bei den Entscheidungen über ihre Umsetzung auf. Problematisch können sich der Architekturentwurf, die IT Infrastruktur, die Drittanbieter von Software, die Auswahl der Infrastruktur usw. gestalten. Hinzu kommen Schwierigkeiten, die spezialisierte Kompetenzen, Fertigkeiten und „Know-How“ erfordern. Diese gehören jedoch nicht immer zum Problem der Domain der Anwendung.

Der Begriff Serverless weist darauf hin, dass die Verwaltung der zugrunde liegenden Serverinfrastruktur der Anwendung von Cloudanbietern übernommen wird.

Für die oben genannten Schwierigkeiten wird Function as a Service (FaaS) Kapitel 3, als Lösung unter der Rubrik „ Serverless“ Abschnitt 4.1 von den Hauptanbietern von „Cloud“ Technologien vorgestellt.

FaaS definiert das Programmiermodel, eine Funktion oder auch „Nano-Microservice“genannt, um den serverless Architekturstil zu adaptieren.

Im Rahmen des Cloud Computing handelt es sich in dieser Arbeit um eine Untersuchung der Serverless Architekturen am Beispiel einer konventionellen Webanwendung. Dabei wird besonders beachtet, ob und wie solche Technologien die Umsetzung erleichtern. Die Entwurfsmuster und die Kernfunktionalität werden ausschließlich mit Serverless Technologien am Beispiel einer Webanwendung (Kompetenz Matrix (KOMA), siehe Kapitel 6), mit AWS umgesetzt.
Als Instrument zur Unterstützung von pädagogischer Diagnostik und Intervention werden in KOMA Kompetenzen und ihre Ausprägungen in Form einer Matrix dargestellt (d.h. KOMA).

1.2 Ziel

Das Ziel ist ein Minimal Viable Product, in Form von einer Single Page Application (SPA) Abschnitt 6.3, ausschließlich mit Serverless Technologien zur Verfügung zu stellen. Die Webanwendung soll möglichst flexibel für zukünftige Änderungen sein.

Anschließend werden die Erfahrungen und Ergebnisse ausgewertet, um zukünftige Entscheidungsprozesse bei der Umsetzung einer Webanwendung zu unterstützen. Es

wird hinterfragt, in wie fern der Serverless Ansatz am Beispiel von Amazon Web Services (AWS) die Umsetzung tatsächlich erleichtert. Eine weitere Analyse prüft, wie sich eine möglichst hohe Flexibilität bei AWS erzielen lässt.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit beschäftigt sich mit dem Serverless Ausschnitt der Cloud Dienste.

Zuerst wird der Leser in die klassischen Servicemodelle (Kapitel 2) eingeführt. Zunächst werden die technischen Anforderungen und die dazugehörigen Beispiele des Serverless Ansatzes erläutert. Das nächste Kapitel überblickt die aktuellen Serverless Angebote der größten Cloud Anbieter. Den Kern der Arbeit bildet die Analyse und Darstellung von Serverless Architekturen (Kapitel 5) und sie fokussiert sich auf AWS. In diesem Abschnitt werden die Serverless Architekturen (Kapitel 5) und das Programmiermodell vorgestellt, sowie die Entscheidungsprinzipien (Abschnitt 4.1) erläutert. Der praktische Teil setzt sich aus der Umsetzung und Bewertung von der oben genannten Serverless Webanwendung KOMA (Kapitel 6) zusammen.

Am Ende erfolgt eine Diskussion darüber, welche Trade-offs entstehen und welche Zukunftsperspektiven Serveless Technologien bieten.

2 Klassische Service-Modelle

Um wiederkehrende IT-Probleme zu beheben, werden Service-Modelle entworfen. Als solches beschreibt Cloud Computing die Bereitstellung von IT-Infrastruktur und IT-Leistungen wie beispielsweise Speicherplatz, Rechenleistung oder Anwendungssoftware als Service über das Internet. [Cha14]

Aus einem Meer von Cloud Diensten ist die richtige Auswahl je nach Anforderungen und Art der technologischen Umsetzung schwer zu treffen. Als Softwarearchitekt, Entwickler oder Projektmanager ist es daher wichtig, die spezifischen Eigenschaften von Cloud Angeboten zu verstehen.

Im Allgemeinen teilen Cloud Angebote laut Chandrasekaran ([Cha14]) folgende Merkmale:

- On-Demand Self-Service - Nutzer können die IT-Kapazitäten, die sie benötigen, selbständig ordern und einrichten. Der Anbieter muss in den Prozess nicht eingebunden werden.
- Broad Network Access bezeichnet den standardbasierten Netzzugriff von verschiedenen Endgeräten (z.B. Smartphones, Tablets, Laptops, PCs) aus.
- Measured Service bietet eine automatische Kontrolle und Optimierung der genutzten Ressourcen durch ihre Dosierung (Metering), wodurch Transparenz für Anbieter und Nutzer sichergestellt wird. Somit bezahlen Kunden nur die Dienstleistungen, die sie auch tatsächlich in Anspruch nehmen.
- Resource Pooling (3) - Ressourcen des Anbieters (z.B. Speicher oder Bandbreite) werden gebündelt, multimandantenfähig bereitgestellt und nach Bedarf zugewiesen.
- Rapid Elasticity (Kapitel 2) - Kapazitäten sind schnell und dynamisch verfügbar und können je nach Bedarf skaliert werden.

Daher ergeben sich für die, in der Cloud betriebenen Anwendungen, folgende Eigenschaften [Cha14]:

- Isolated state - Der Zustand wird in kleinen Einheiten der Anwendung isoliert, so dass sie besser skalieren. Eine zustandslose IT Ressource kann ohne Synchronisierungen aggregiert oder gelöscht werden. Dieser Zustand bezieht sich nicht nur auf die Verwaltung der Interaktionen eines Clients, sondern auch auf dessen Datenverarbeitung.
- Distribution - Anwendungen müssen so in Komponenten zerlegt werden, dass sich ihre Ressourcen weltweit auf-/verteilen.
- Elasticity - Die Anpassung sowohl auf die Anzahl als auch auf die Leistungsfähigkeit der zu benutzenden IT Ressourcen kann im Sinne einer Addition oder Subtraktion erfolgen. Im ersten Fall nimmt auf der Ebene der horizontalen Skalierung (scale out) die Anzahl der Server zu. Während bei der vertikalen Skalierung (scale up) die Leistungsfähigkeit der Ressourcen der Server steigt.
- Automated management - Die konstanten Aufgaben zur Verwaltung von Elasticity sollen automatisiert werden, um eine Cloudanwendung fehlerresitent auf Ressourcenebene zu implementieren.
- Loose coupling - Die Minimierung von Abhängigkeiten einer Anwendung von IT Ressourcen vereinfacht die Bereitstellung, die Fehlerkontrolle und Wiederverwendung von Komponenten. Kapitel 4

Die nächsten Unterkapitel stellen exemplarisch klassische Service-Modelle vor.

2.1 IaaS

Infrastructure as a Service (IaaS) kann als ein Service beschrieben werden, der Abstraktionen für Hardware, Server und Netzwerkkomponenten bereitstellt. Der Serviceanbieter stellt die Ausrüstung zur Verfügung und ist für die Unterbringung, die Inbetriebnahme und die Wartung der Server verantwortlich [You15]. Der Benutzer bezahlt nicht für die Hardware, deren Lagerung und den Zugriff, sondern für die Nutzung des gesamten Servicemodells z.B.: Zahlung nach benutzten Stunden, Ressourcen usw.

Die Aufgaben für Systeme mit Softwareelementen wie Load Balancing, Transaktionen, Gruppierung (Clustering), Caching, Benachrichtigung (Messaging) und Datenredundanz werden komplexer. Diese Elemente fordern an, dass Server zu verwalten, warten, flicken (patched) und zu sichern sind. In einer nicht-trivialen Systemumgebung sind solche Aufgaben zeitintensiv, aufwändig fertigzustellen und daher schwer effizient zu betreiben. Infrastruktur und Hardware sind zwar nötige Komponenten für jegliche IT-Systeme, aber gleichzeitig stellen sie nur das Medium für deren Anwendung dar - sei es Geschäftslogik, oder ein darauf aufbauender Dienst. [Cha14]

2.2 PaaS

Platform as a Service (PaaS), kann als ein Service beschrieben werden, der eine Rechenplattform liefert, z.B. ein Betriebssystem, eine Ausführungsumgebung für Programmiersprachen (siehe ElasticBeanstalk [AWS17a]), eine Datenbank oder einen Webserver. Dieser Dienst übernimmt je nach benutzerdefinierter Konfiguration sowohl die Wartung der Datenbank, des Webservers und der Versionen des Laufzeitquellcodes, als auch deren Skalierbarkeit. [You15].

Inkonsistenzen in der Infrastruktur oder den Umgebungen können durch den hohen Aufwand der Serververwaltung entstehen. Sie werden durch standardisierte und automatisierte Angebote von PaaS umgangen. Deren effiziente Benutzung ist abhängig davon, wie gezielt der Quellcode auf die Features der Plattform abgestimmt ist. Dies ergibt auf einer Seite weniger Wartung, aber andererseits erfordern die importierten Anwendungen (z.B. für ein „Standalone“ Server) eine Anpassung an die Plattform.

Die *Containerisierung* ist eine Isolierung der Anwendung von ihrer Umgebung. Die Konfiguration des Containers, sowie dessen Einsatz (Deployment) ist nicht trivial und erfordert daher spezialisiertes Wissen über Containerisierung. Für das Monitoring werden bestimmte Tools wie Boot2Docker [Goo17a] oder cAdvisor [Goo17b] benötigt. Jedoch bietet die Containerisierung eine ausgezeichnete Lösung für Anwendungen mit starker Kopplung zu anderen Softwarekomponenten. [You15]

2.3 SaaS

Software as a Service (SaaS) kann als ein Service beschrieben werden, der OS-Images mit konfigurierbaren Diensten wie Datenbanken, Webanwendungen usw. bereitstellt. SaaS gestaltet sich benutzerfreundlich, da die Konfiguration und das Deployment dieser Softwaredienste nicht erlernt werden müssen, um sie in eine größere Anwendung einzubinden. Anfallende Gebühren berechnen sich nach der Nutzungsdauer.
Ein Großteil traditioneller Software bietet seine Version als SaaS nicht an. Dies impliziert laut Chandrasekaran, dass dieses Serviceliefermodell sich für Anwendungen nicht gut eignet wegen der folgenden Punkte:

- Geringe Latenz kann durch die Entfernung der gespeicherten Daten für Echtzeitanwendungen nicht gewährleistet werden.
- Die Datensicherheit kann nicht sichergestellt werden, da die mitbeteiligten Drittanbieter bei SaaS die Service Level Agreement (SLA)s von Kunden nicht immer erfüllen.
- Anforderungen bestimmter Software verlangen eine Zentralisierung und eine Lokalisierung vor Ort, anders als bei SaaS.

[Cha14]

3 FaaS als Grundlage der Serverless Architekturen

Function as a Service (FaaS) kann als ein Rechenservice beschrieben werden, der nach Anfrage isoliert, unabhängig und granular ausgeführt wird. Komplexe Probleme wie horizontale und vertikale Skalierbarkeit, Fehlertoleranz und Elastizität werden von Kunden nur noch nach Bedarf konfiguriert und von dem Anbieter verwaltet. Die Besonderheit von FaaS ist die „Unit of Deployment“ und die Skalierung in Form einer Funktion. [You15]

Somit lässt sich in der folgenden Tabelle 3.1 die Skalierungseinheit (Unit of Deployment) und die Abstraktionsebene von den relevanten Dienstmodellen vergleichen.

Tabelle 3.1: Vergleich IaaS PaaS FaaS Skalierung und Abstraktion

	IaaS	PaaS	FaaS
Skalierungseinheit	Virtuelle Maschine	Anwendung	Funktion
Abstraktion	Hardware	Betriebssystem	Laufzeitumgebung der Sprache

Im Rahmen des „Open Lambda“ Projekts wurde die Latenz zwischen einer PaaS und einer FaaS Anwendung verglichen. Deren Autoren (siehe [HSH+16]) stellten fest, dass die Funktion kürzere Antwortzeiten lieferte, weil sie für jede Anfrage eine neue Instanz kreierte, wogegen die PaaS nicht skalierte u.d. die Anfragen in eine Queue (Schlange) einreihte.

Es werden nun die konzeptionellen Bestandteile von FaaS näher ausgeführt.

Events innerhalb eines verteilten Systems müssen verwaltet werden. Technologien wie virtualisierte oder containerized Server erzeugen neue Serverinstanzen zur Verarbeitung von einer Kette von variablen Events, die danach gelöscht werden. [Kin16]

Die entstehende Problematik ergab sich durch eine starke Zunahme an Elementen, die einen hohen Verwaltungsaufwand forderten. Was zurück auf die oben genannten Aufgaben Abschnitt 2.1 führte.

Polling ist der Ausdruck für eine zyklische Abfrage über einen Status den Ressource z.B. von Ports oder Locks. Die Verwendung von Systemressourcen ist ineffizient im Vergleich zu Alternativansätzen wie z.B in dem Push- oder Pull- Kommunikationsmodell. [Kin16]

Funktionale Programmierung ist ein Programmierparadigma, in dem Funktionen nicht nur definiert und angewendet werden können, sondern auch wie Daten miteinander verknüpft, als Parameter verwendet und als Funktionsergebnisse auftreten können. Zustand und mutable Daten werden vermieden, damit Nebeneffekte nicht entstehen und die Komposition flexibler wird. Das stellt einen Vorteil für die Skalierung eines Softwarekomponenten dar. [Ray13]

Die Implementierung einer solchen Funktion geschieht durch die Auswahl der Programmiersprache und der von dem Cloudanbieter vorgegebenen Funktionsfassade. Diese wird vom Cloudanbieter aufgerufen, stellt aber keine zusätzlichen Bibliotheken zur Verfügung, daher ist es nötig, dass die auszuführende Datei alle Abhängigkeiten enthält.

Der Fokus bei FaaS liegt auf der Quellcodeentwicklung und nicht auf der Bereitstellung (Provisioning) von Servern, der Installation von Software, dem Einsatz (Deployment) von Containern oder auf konkreten Details der Infrastruktur.

Für die Betrachtung, ob FaaS eine Lösung für eine konkrete Problemstellung ist, folgt eine Auflistung von Kriterien nach King [Kin16]:
Es ist nicht empfehlenswert FaaS zu benutzen, wenn:

- der Entwickler Rootzugriffsrechte auf alle Ressourcen eines Servers benötigt
- die Priorisierung von Betriebssystemattributen wie CPU, GPU, Networking oder Speichergeschwindigkeit angefordert ist
- Sicherheit relevant ist. Unautorisierte Zugriffe können mit FaaS nur auf Systemebene erkannt werden
- dauerhaft laufende Prozesse angefordert sind

Es ist empfehlenswert FaaS zu benutzen, wenn:

- Aufgaben als Reaktion auf Events erledigt werden

- ein Scriptbehälter für z.B Cron-Aufgaben benötigt wird. Hier sind die Zugriffsrechte beschränkter, Fehler einfacher zu erkennen und an einer Stelle aggregiert ([AWS17b]), des Weiteren können Deployments einfacher angestoßen werden
- die Skalierung des Servers bei einer ressourcenintensiven Verarbeitung vermieden werden soll Abschnitt 4.2
- Services vorgegeben sind, die selten benutzt werden
- die Verwaltung von API-Server umgangen werden soll

Die Frage, wie sich Architekturen mit einer FaaS Technologie gestalten, wird im nächsten Kapitel erläutert.

4 Serverless-Angebote und Architekturen

Durch Softwarearchitekturen wird kommuniziert, für welchen Zweck die Software konzipiert ist. Ihre Entwurfsmuster bieten generische Lösungen für wiederkehrende Probleme bei der Softwareentwicklung.

Eine konventionelle Webanwendung in diesem Sinne, ist ein System, das über eine Präsentation-, Daten- und Logikschicht verfügt. Jede Schicht kann mehrere Logik-Layers enthalten, die für unterschiedliche Funktionalitäten der Domains verantwortlich sind. Logging ist ein Beispiel für das Cross-Cutting Concern, das Layers überspannt. Die Komplexität der Anwendung wächst zusammen mit der Anzahl der Schichten. Eine Überprüfung der Architektur ist sinnvoll, wenn eine erfolgreiche Codeänderung von einer Anderen abhängt.

Service Oriented Architecture (SOA) unterliegt der Annahme, dass ein System aus mehreren kleinen, austauschbaren, wiederverwendbaren und entkoppelten Diensten besteht. Für die Entwicklung und Integration solcher Systeme liefert SOA eine Reihe von Entwurfsprinzipien und Standards. Entwickler erstellen autonome Services, die durch Nachrichtenübergabe kommunizieren und oft ein Schema haben oder eine Schnittstelle, die definiert, wie die Nachrichten erzeugt werden. [Cha14]

SOA ermöglicht gegenseitigen Datenaustausch zwischen Programmen von unterschiedlichen Anbietern, ohne dass zusätzliche Änderungen an den Services vorzunehmen sind. Die Bestandteile eines solchen Architekturstils sind sowohl Standardschnittstellen als auch voneinander unabhängige Services [Sta07].

Der Fokus in der Cloud liegt daher auf Service und Servicekomposition [Cha14].

Microservices und Serverless versuchen die Komplexität der SOA (Abbildung 4.1)

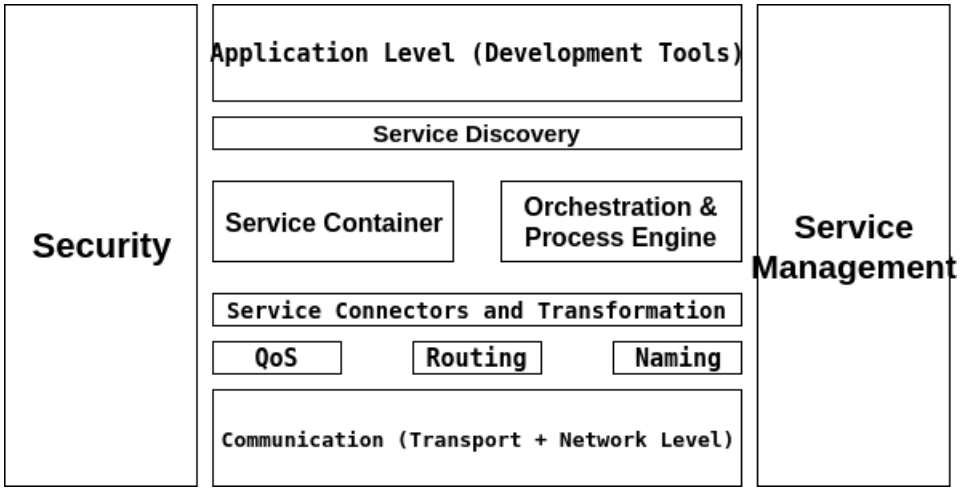


Abbildung 4.1: SOA Architekturreferenz [Sta07]

anzusprechen. Beide Ansätze führen Separation of Concerns, häufige Deployments und heterogene Domain Specific Language (DSL)s mit sich. [H+17]

Auf einer Seite können Microservices ihren Zustand sowie ihre Daten speichern und mit Hilfe von Frameworks implementiert werden. Auf der anderen Seite sind Serverless Architekturen zustandlos, ihre Datenspeicherung ist zeitlich begrenzt und sie unterstützen Frameworks nicht direkt. ConnectWise [Con17a], Netflix [Net17] und UNLESS [UNL17] sind Beispiele für Unternehmen, die auch von Serverless Architekturen profitieren.

Event Driven Architecture (EDA) ist eine Softwarearchitektur, in der das Zusammenspiel der Komponenten durch Ereignisse gesteuert wird. Die Ereignisorientierung besitzt das Potenzial, dass die Architekturen von Anwendungen agiler, reaktionsschneller und echtzeitfähig werden. Laut Ralf Bruns und Jürgen Dunkel zeichnet sich EDA durch komplexe Fachlogik, große Datenvolumina, geringe Latenzzeit, Skalierbarkeit und Agilität aus. [RB]
Die Serverless Technologien können durch Benachrichtigungen gestartet werden. Dieser EDA-Stil verstärkt die Entkopplung auf einer temporären Ebene zwischen Producer und Consumer. Weiterhin ermöglicht ein Kommunikationskanal zur Benachrichtigung eine asynchrone Verarbeitung, ohne dass das System auf Grund von Fehlern abstürzt. [HW04]

Zusammenfassend sind folgende Vorteile ersichtlich [Cha14]:

- Die Wiederverwendung von Services in unterschiedlichen Anwendungen senkt

die Entwicklungskosten und den Time-To-Market.

- Durch die Standarisierung der Services kann ein System mit einer Rekonfiguration und ohne Weiterentwicklung schnell auf die geschäftlichen oder externen Bedürfnisse angepasst werden. Somit wird ein agiles Arbeiten möglich.
- Das Monitoring hilft Fehler zu erkennen und die Leistung zu messen.
- Aggregate von bereitgestellten Services können komplexere und domainübergreifende Aufgaben ausführen.

Im späteren Kapitel wird REST als Teil des EDA-Architekturstils vorgestellt.

4.1 Serverless

Serverless kann als ein Ansatz beschrieben werden, der die Verwendung von einem Rechenservice, Dienste von Drittanbietern, von Application Interface (API)s und die Anwendung von Architekturmustern fördert. Ein solcher Anwendungsfall ist die Kommunikation mithilfe eines „Delegation-Tokens“ zwischen den Front- und Back-End Diensten. FaaS ist nur ein Aspekt dessen.

Serverless übernimmt die Entwurfsprinzipien von SOA und EDA (siehe Kapitel 4) und daher ergeben sich laut Sbarski folgende Richtlinien:

- Ein Rechenservice wird genutzt, um Quellcode auf Anfrage auszuführen, kein Server.
- Zustandslose Funktionen unterliegen dem Single Responsability Principle (SRP).
- Für den Architekturentwurf werden Push basierte ereignisorientierte Pipelines genutzt.
- Front-Ends werden durch die Einbettung von mehr Zuständigkeiten verstärkt.
- Dienste von Drittanbieter werden dem Schreiben von eigenem Quellcode bevorzugt.

[Sba17]

Die Vernetzung von zustandslosen Funktionen erlaubt, komplexe Systeme zu entwerfen, die einfach zu skalieren sind. Die Komplexität und längerfristige Wartbarkeit des

Systems lässt sich dadurch reduzieren, dass der Controller und/oder Router aus dem Model View Controller (MVC) [Fow17a] vom Back- zum Front-End verschoben wird und Dienste von Drittanbieter hinzugefügt werden. [You15]

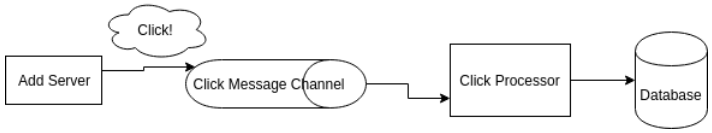


Abbildung 4.2: Classic Add Server

In einem beispielhaften konventionellen AdServer wird nach einem Klick auf eine Werbung eine Nachricht über einen Kanal an einen Klickprozessor geschickt, der innerhalb einer Anwendung ausgeführt wird.

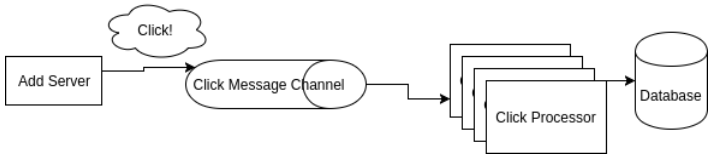


Abbildung 4.3: Serverless Add Server

Mit dem Serverless Ansatz wird dieser Klickprozessor pro Nachricht als eine neue Instanz der Funktion ausgeführt. Ihre Laufzeitumgebung und ihr Messagebroker wird von dem Cloudanbieter verwaltet. [Fow17b]

Dazu wird für den Entwurf von Serverless Architekturen eine Reihe von Mustern von unterschiedlichen Autoren vorgeschlagen.

4.2 Pipes and Filters, Compute as a Glue

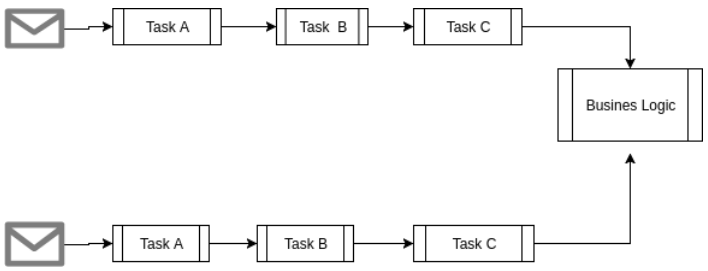


Abbildung 4.4: Pipes and Filters Entwurfsmuster

Eine Anwendung kann Aufgaben von unterschiedlicher Komplexität bewältigen. In einem monolithischen Modul sind das Refactoring, die Optimierung und die Wiederverwendung erschwert.
Zunächst werden die Aufgaben in diskrete Elemente (oder Filter) nach dem SRP zerteilt und in einer Pipeline kombiniert. Dies hilft redundanten Quellcode zu vermeiden, ihn zu löschen, zu ersetzen oder in zusätzliche Komponenten zu integrieren, sobald sich die Aufgabenanforderungen ändern [HSB+14]. Ein weiterer Vorteil besteht darin, dass ein Flaschenhalseffekt vermieden wird, in dem mehrere Instanzen erzeugt werden, falls ein Element nicht genug Ressourcen für die Verarbeitung hat.

4.3 Legacy Api Proxy

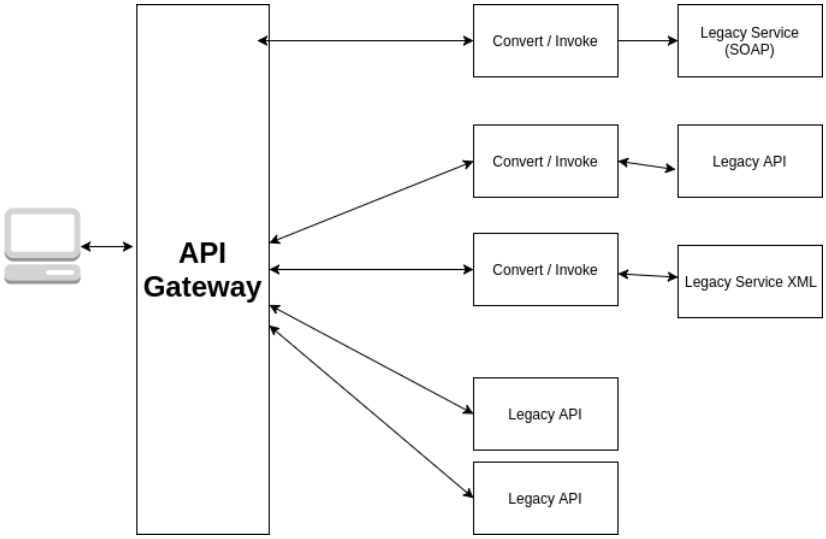


Abbildung 4.5: Legacy API Muster

Wenn eine API veraltet oder schwer zu benutzen ist, kann eine extra (RESTful) API in den Vordergrund gestellt werden, die in gesonderten Prozessen Daten transponieren und für die angeforderten Formate aufstellen (marshall). Dies ist besonders nützlich, wenn die Legacy-Services selten benutzt werden. Zusätzlich erleichtert der Api Proxy die Integration mit anderen Architekturansätzen.

4.4 Compute as a Backend

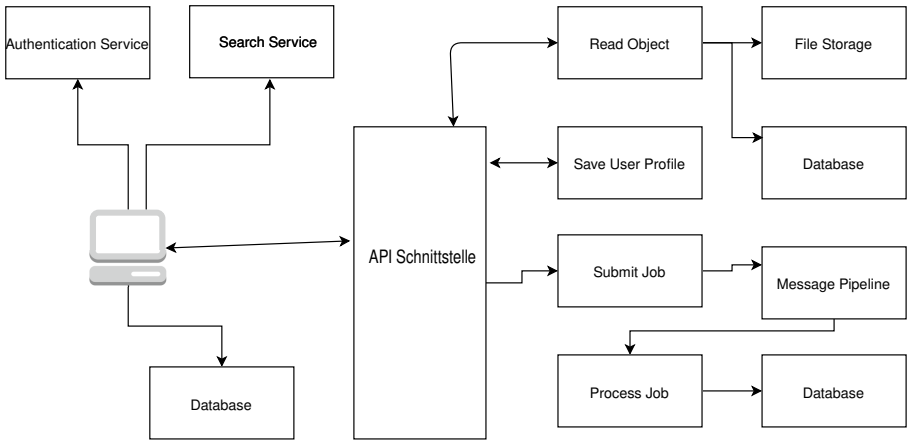


Abbildung 4.6: Compute as a Backend Muster

Obwohl der Client direkt mit Services kommunizieren kann, müssen vertrauliche Informationen geschützt werden, indem sie das Back-End verarbeitet [Sba17]. Diese Aufgaben können hinter einer REST Schnittstelle koordiniert werden. Wie in Abschnitt 4.1 erwähnt, minimieren die Einbettung der Dienste von Drittanbietern und verstärkte Front-Ends den Fußabdruck des eigenen Back-Ends.

4.5 Graph Query

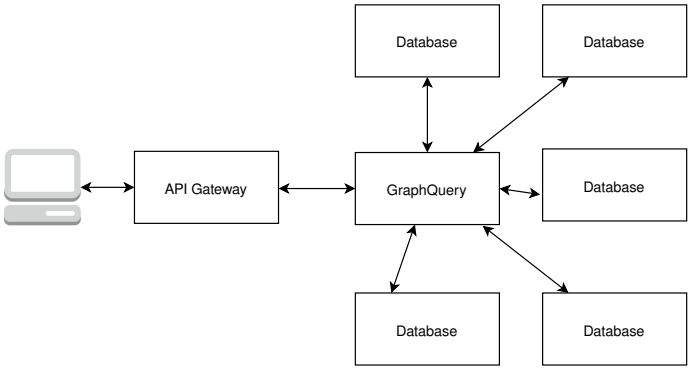


Abbildung 4.7: Graph Query Muster

Wenn mit einer Anfrage mehrere Datenbanken abgefragt werden, entstehen multiple Paketumlaufzeiten (Round-Trip). Stellt eine REST Schnittstelle in einer Anfrage zu wenig Queryparameter zur Verfügung, dann entsteht Overfetching, weil die Anfrage

nicht präzise genug ist. Dagegen kann der Client die Parameter für die Abfrage spezifizieren und das Back-End baut sie zusammen und führt sie aus.

4.6 Real time processing

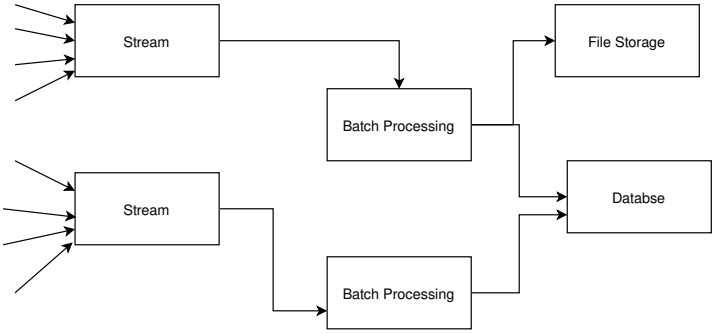


Abbildung 4.8: Real Time Processing Muster

Die Verarbeitung von Streams in Echtzeit kann durch einen Puffer erfolgen, der je nach Konfiguration die Daten weiter an den Worker leitet. Einen wesentlichen Vorteil stellt die unabhängige Skalierung des Streams und des Workers je nach Anfrage dar. Bei fehlerhafter Verarbeitung werden die Prozesse neu angestoßen.

4.7 Priority Queue

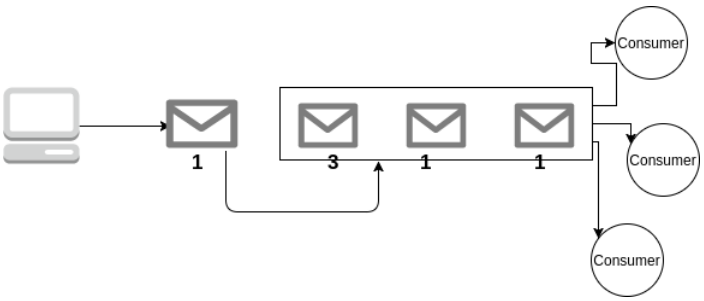


Abbildung 4.9: Priority Queue Muster

Anwendungen können spezifische Aufgaben delegieren, wie z.B die Integration mit anderen Anwendungen und Services. Anders als bei einer FIFO (First In First Out) Queue können die Nachrichten nach Priorität automatisch sortiert und asynchron verarbeitet werden. Für Systeme ohne integrierte Priorisierung können mehrere Queues

für unterschiedliche Prioritäten benutzt werden und die Anzahl von Consumerprozessen wird entsprechend angepasst. Im letzteren Fall wird dabei die Starvation von Nachrichten, mit geringer Priorität, vermieden.

Das Priority Queue wird im Rahmen des Entwurfsmusters Competing Consumers als ein Consumerservice aufgefasst. Steigt stark die Anzahl von Anfragen an die Kapazitäten des Systems, kommt es zu seiner Überbelastung. Ein Consumerservice als Moderator von Anfragen ist im Stande das zu verhindern.

4.8 Fan Out

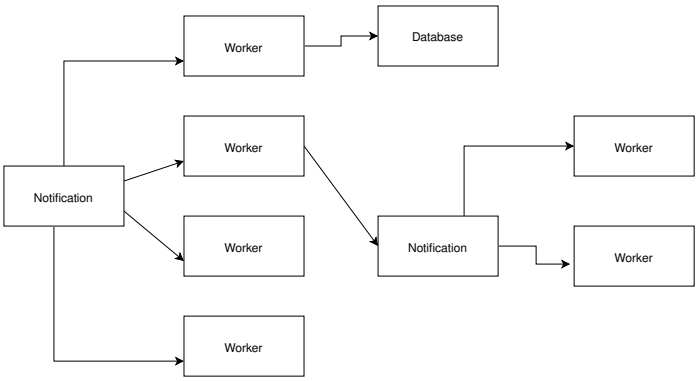


Abbildung 4.10: Fan Out

Bei dem Eintritt eines Events können ein oder mehrere Subscriber durch die gepushte Benachrichtigung angestoßen werden. Damit wird ein Kommunikationskanal Abschnitt 4.7 zur Verwaltung von Nachrichten wiederverwendet und extra Geschäftslogik umgangen, z.B. kein Command Pattern für die gleiche Funktionalität [Sba17].

4.9 Federated Identity

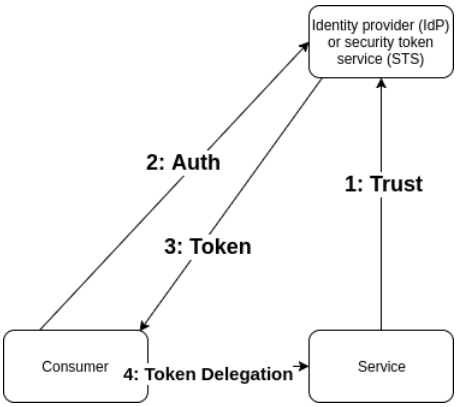


Abbildung 4.11: Federated Identity

Benutzer arbeiten mit multiplen Anwendungen von unterschiedlichen Organisationen. Um gleiche Zugangsdaten für Benutzter wiederverwenden zu können, wird dem Nutzer eine Identität bei einem Drittanbieter zugeteilt, diese wird mit einem Token an die Anwendung weitergeleitet. Der Authentisierungscode kann daher von dem Anwendungscode abgetrennt werden, damit kann zusätzliche Komplexität bei der Codierung für die Authentisierung vermieden werden.

5 AWS-Serverless-Angebote

AWS ist ein Cloudandbieter unter vielen. Hier ein Überblick über verschiedene FaaS Angebote:

	AWS Lambda	Iron.io	Google Funktions	MS Funktions	IBM OpenWhisk
JS	+		+	+	+
Python	+	+		+	+
Java	+	+			+
Trigger					
- Event Trigger	+		+		+
- HTTP Trigger	+		+		

Tabelle 5.1: Vergleich FaaS Angebote der Cloudanbietern

Die oben angezeigten Eigenschaften entsprechen dem FaaS Angebot als solches, ohne zusätzliche Kombination mit Technologien wie Containerisierung der Lambdalaufzeitumgebung.

AWS zeichnet sich dadurch aus, dass FaaS gut in seinem Ökosystem integriert ist. Da KOMA mit AWS umgesetzt ist, folgt eine Beschreibung der unterschiedlichen FaaS Angebote.

Lambda ist ein Rechenservice, der aus Quellcode und dessen Abhängigkeiten besteht. Die horizontale Skalierung erfolgt automatisch, elastisch und vom Cloudanbieter verwaltet; währenddessen die vertikale nach Konfiguration erfolgt. Lambda wird als Einheit für die

Skalierung und das Deployment benutzt. AWS unterstützt Javascript, Python, C# und Java. Die letzte Programmiersprache betrifft das Konzept „Cold- und Warm run“ besonders, da es nötig ist, die Java Virtual Machine (JVM) in ihrer Laufzeitumgebung hochzufahren (cold run). Der Cloudanbieter stützt sich auf das Pull- und Push- Kommunikationsmodell. Bei Pull überprüft die Lambda Laufzeitumgebung in regelmäßigen zeitlichen Abständen, ob Events z.B. bei Kinesis eingetreten sind und ruft die entsprechende Lambda Funktion mit einer Event-Datennutzlast auf. Bei Push ruft die Eventquelle z.B S3 je nach Konfiguration (event source mapping) die entprechende Lambda Funktion auf.

Zustände können von anderen Dienste wie einer Datenbank aufgerufen werden.

In der folgenden Abbildung 5.1 findet sich eine Übersicht der Architektur von Lambda. Nachdem der Load Balancer ein Event (1) in Form eines Remote Procedure Call (RPC)s aufgenommen hat, wird der Server benachrichtigt (2) und lädt (3) entsprechenden Quellcode in die Arbeitsspeicher (4), um ihn auszuführen und die Ergebnisse zurück an den ursprünglichen Load Balancer zu senden(5).

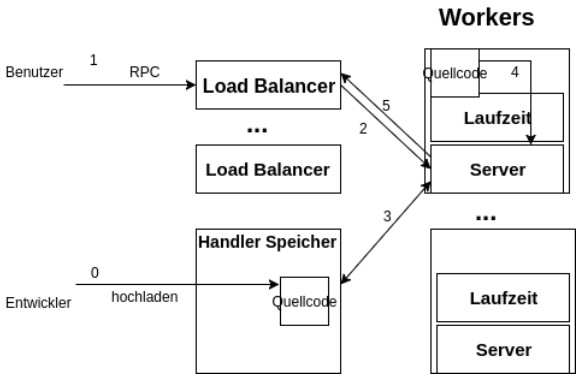


Abbildung 5.1: Vereinfachte Architektur von Lambda

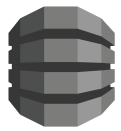
Der Ausgangspunkt dieses Entwurfs legt die gemeinschaftliche Benutzung von vorhandenen Ressourcen, so oft es sich anbietet, nahe.

Steigt die Anzahl der „Worker“ , entsteht die horizontale Skalierung. Diese kann bis Null sinken, u.d. kann der Benutzer sich die Kosten für nicht benutzte Ressourcen sparen.

API Gateway ist eine Fassade, um Operationen sicher auszuführen, wie die Benachrichtigung von Kunden per Email und deren Identitätsüberprüfung. Weil AWS die Skalabilität von API Gateway und von Lambda übernimmt, ist die Bereitstellung und Wartung von EC2 Instanzen und die Konfiguration deren Load Balancer nicht mehr nötig.

Simple Notification Service (SNS) erweitert das schon gut etablierte Beobachtermuster, in dem es einen Kanal für Events hinzufügt. Die konzeptionelle Technologie hinter SNS wird als „Publish-Subscribe Channel [HW04]“ Muster bezeichnet und entspricht den oben genannten „Fan Out“ Abschnitt 4.8 Architekturmustern. AWS kann durch Redundanz mindestens eine Lieferung der Nachricht gewährleisten.

Simple Storage Service (S3) ist ein Speicherdienst, der durch SNS Abschnitt 5 die Events (z.B das Löschen oder Erzeugen eines Objektes) an SNS Abschnitt 5, Simple Queue Service (SQS) Abschnitt 5 oder Lambda schicken kann. „Buckets“ sind Verzeichnisse auf der höchsten Ebene des Verzeichnissystems. Ein Objekt ist eine Kombination von Daten, Metadaten und eines Keys, der innerhalb des Buckets eindeutig ist.



DynamoDB ist eine NoSQL Datenbank. Ihre Tabellen bestehen aus Items (Zeilen) und deren Attributen (Spalten). Die Datenbank hat laut AWS unendliche Datenkapazitäten und der Datenverkehr ist unbegrenzt. Durch automatische Skalierung mindert sich die Leistung nicht. Bei der Änderung eines Zeilenwertes in DynamoDB ist die Konfiguration eines Anstoßes der Lambdafunktion möglich.

Simple Queue Service (SQS) ist eine message queue. Sie erlaubt die Interaktion von mehreren Publisher und Consumer in einer SQS und verwaltet automatisch den Lebenszyklus der Nachrichten und kontrolliert Auszeiten (Time out) oder individuelle Verzögerungen.

Kinesis Streams ist ein Service für Echtzeitverarbeitung von Datenstreams. Es wird für Logging, Datenimport, Metriken, Analytics und Reporting benutzt. Ein Kinesis Stream ist eine sortierte Folge von Datensätzen, die auf „Shards“ verteilt sind. Diese definieren die Kapazität des 'Durchsatzes' (Throughtput) von einem Stream und können nach Bedarf vergrößert werden.

Relational Database Service (RDS) hilft bei dem Setup und Wartung von mySQL, MariaDB, Oracle, MS-SQL, PostreSQL und Amazon Aurora mit automatischer Bereitstellung (provisioning), Sicherung, patching, recovery, repair und Fehlererkennung. RDS kann durch SNS Abschnitt 5 über eigene Events berichten.

Simple Email Service (SES) behandelt die Absendung und die Empfangsoperationen wie Spamfilterung, Virus Scan und Ablehnung nicht vertrauter Quellen. Emails können weiterhin in S3 gespeichert, an Lambda versendet werden oder eine SNS Benachrichtigung auslösen.

6 KOMA, eine Beispielanwendung

Im Folgenden wird eine Beispielanwendung, unter Verwendung der in Kapitel 5 genannten Entwurfsmuster, implementiert.

Die Beschleunigung der Veränderungen in der heutigen Gesellschaft und der technologischen Landschaft prägt sich in Bildung und Beruf in so fern aus, dass die heutigen Rahmenlehrpläne nicht mehr fachlich, sondern an der Kompetenzentwicklung orientiert sind, um u.a Kompetenzprofile für Lerner zu erstellen. Es existiert bereits ein anerkannter Europäischer Rahmen [EQF17] für Kompetenzbildung: European Qualifications Framework (EQF). Am Beispiel von Sachsen-Anhalt [SA17] werden diese Kompetenzorientierung auf die spezifischen Bedürfnisse der Schule beschrieben und die Unterrichsstunden entsprechend gestaltet.

Die Anwendung soll für die pädagogische Diagnostik und Intervention genutzt werden. Ziel der Umsetzung ist es daher, auf einer Seite die von einem Schüler erworbenen und zu erwerbenden Kompetenzen und deren Niveau nachzuvollziehen. Andererseits bietet sie das Potenzial die Bildungs-, Unterrichts- und Stundenplanung zu unterstützen.

Kommt diese Anwendung in Bildungsinstitutionen zum Einsatz, dienen die Rahmenlehrpläne als Leitpfad für die Bezeichnungen und Anforderungen der Kompetenzen, wohingegen KOMA für die Organisation der einzelnen Fachrichtungen oder Lehrveranstaltungen zuständig ist. Da der EQF als Basis mit internationaler Anerkennung genutzt wird, den Kompetenzstand eines Individuums abzubilden, wird die Bildungsqualität international vergleichbar.

Den Kern von KOMA bildet die Zuweisung von Aktivitäten auf vordefinierte Kompetenzen. Erstere lassen sich einzeln oder in einer Sequenz anordnen. Sequenzen werden in Lehrveranstaltungen zusammengestellt. So können Aktivitäten, Sequenzen und Kompetenzen als Gestaltungsmittel für Lehrveranstaltungen genutzt werden.

Das folgende Beispiel beschreibt einen fachorientierten Ansatz zur Gestaltung von Lehrveranstaltungen:
Das Fach „Web Computing “ lässt sich mit einer Sequenz von Lernaktivitäten (Unterrichtseinheiten) gestalten. Die zu behandelnden Themen erfordern grundlegendes Wissen und Fertigkeiten wie das Beschreiben von Kommunikationsprotokollen und Bash Scripting. Dabei ist zu beachten, dass Wissen und Fertigkeiten kumulativ wachsen und damit aufeinander aufbauen.

Im Gegensatz zu dem fachorientierten Ansatz wird nun der kompetenzorientierten Ansatz erläutert: Das Kompetenzmodell differenziert unterschiedliche Kompetenzen in vielfältigen Zusammensetzungen. Ihre Fertigkeiten und/oder Wissen werden während der Lernaktivitäten, bei der die Auswahl von Themen freigestellt ist, erworben.Durch die Sequenzierung von Lernaktivitäten baut sich das Kompetenzmodell aus.

6.0.1 Anforderungsanalyse

Die Auflistung Unterabschnitt 6.0.1 stellt einen für diese Arbeit angepassten Ausschnitt der Anforderungen für KOMA dar:

- Mit dem EQF vergleichbare Kompetenzdefinitionen
- Berücksichtigen zukünftiger Erweiterungen
- Abrufbarkeit durch den Browser
- Private Datenspeicherung u.d Login
- Ertragen von großen Nutzlastschwankungen

6.0.2 ER-Modell

Aus der Beschreibung von KOMA Kapitel 6 ergibt sich folgendes Modell. Unterabschnitt 6.0.1

Dieses ER-Diagramm Unterabschnitt 6.0.1 definiert die Beziehungen zwischen den in Kapitel 6 beschriebenen Konzepten. Eine Kompetenz (Competence) besteht aus Fertigkeiten (Skill), die in einer Lernaktivität (LearningActivity) erworben werden. Die Letztere gehört zu einer Sequenz von Lernaktivitäten (LearningSequence). Erworbene Kompetenzen können nach Klassenstufen und Fächern aufgerufen werden, damit

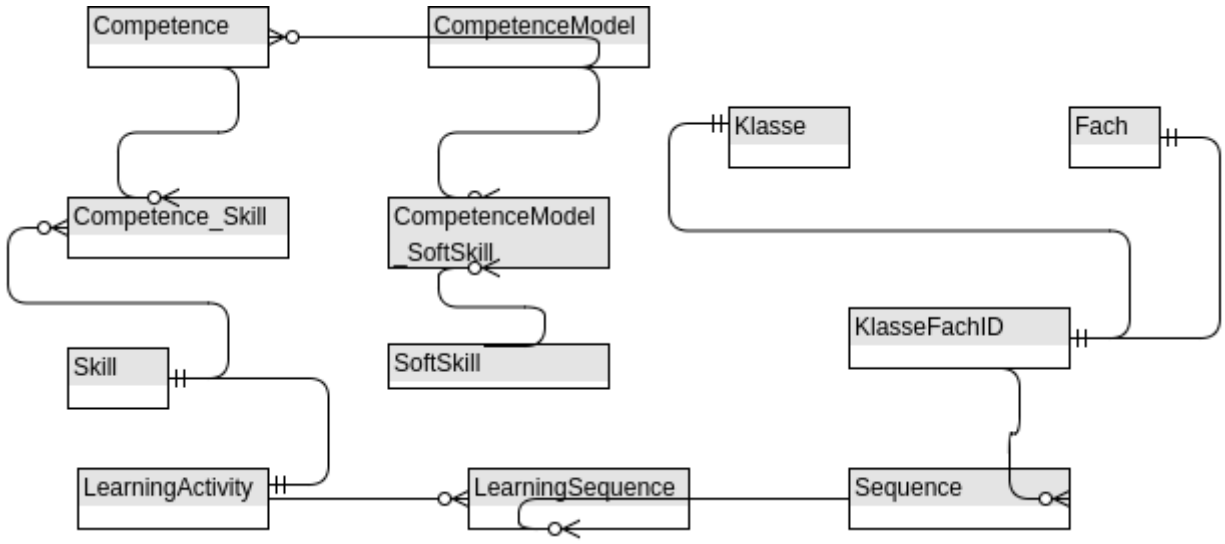


Abbildung 6.1: Entity-Relationship Modell für KOMA

ein Kompetenzstand abgeleitet werden kann. Das Modell ermöglicht die Gestaltung von EQF-konformen Kompetenzprofilen (CompetenceModel).

6.0.3 Komponentenübersicht

Um einen Anhaltspunkt zu geben, werden hier die Softwarekomponenten beschrieben.

Bei dem Aufruf der Website im Browser präsentiert sich ein „Sign in“ Knopf zum Einloggen und verschiedene Möglichkeiten zur Formulierung einer Abfrage. Diese ist in einem S3 Bucket als statische Webseite gelagert. Wird der Knopf zum Einloggen gedrückt, fordert eine Weiterleitung (mit dem dargestellten Schlüssel) die Authentisierung des Benutzers an. Mit der erfolgreichen Operation kehrt die Nutzeransicht mit einem „Token“ auf die Webseite zurück. Der erhaltene Token wird als Autorisierung-Parameter in dem „HTTP Request Header“ mit den nächsten Anfragen an das Back-End geschickt.

Nach dem Login kann der Browser durch Absenden der nächsten Anfragen seinen Token zur Überprüfung übergeben. Die API Gateway empfängt die Anfragen und transponiert deren Parameter, um zunächst die entsprechende Lambdafunktion synchron aufzurufen. Da die API nach REST entworfen ist, werden die Lambdafunktionen nach der HTTP-Methode und der URL abgebildet.

Dieser Loginprozess entspricht dem in Abschnitt 4.9 beschrieben Federated Identity

Muster.

Eine Anfrage an die `GET https://<host>/page/{individual}` URL führt die Lambdafunktion „OWL Parser“ aus. Die Funktion liest die Datenbasis von S3 „OWL Storage “ und extrahiert den Wert des `{individual}`-Parameters, um eine Antwort zu generieren. Diese abstrahierte Zuweisung von URLs auf Back-End Dienste entspricht einem REST Ansatz. Dagegen erwartet die `POST https://<host>/sparql` URL eine Abfrage im „Request Body“ der Anfrage mit dem Key „Query“ , um sie auf der oben genannten Datenbasis auszuführen. Dabei unterstützt die Java Bibliothek Jena ARQ [Fou17]

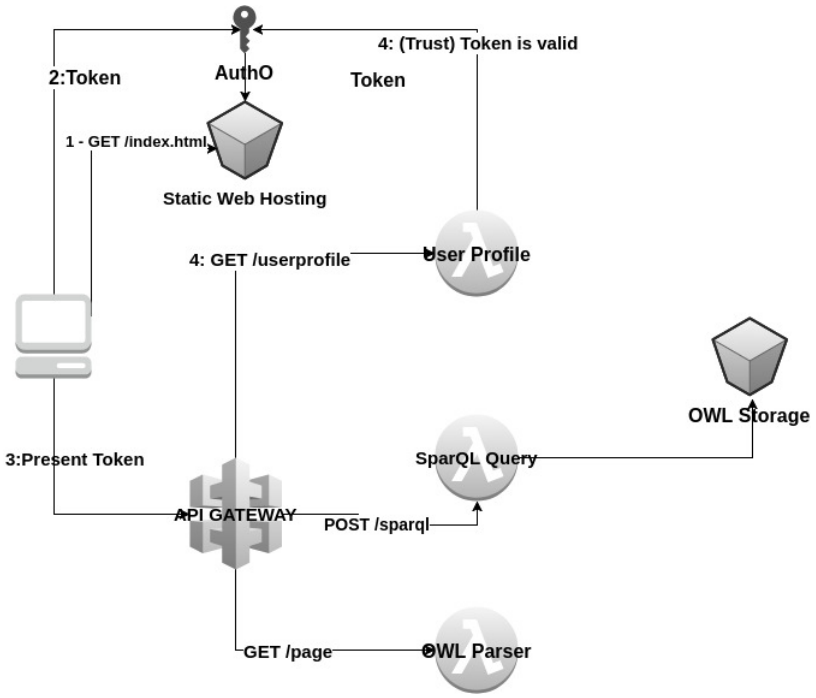


Abbildung 6.2: KOMA Components

6.1 Umsetzung

Die grundlegende Vorgehensweise bei der Umsetzung dieser Webanwendung gliedert sich zunächst in die Datenspeicherung und deren Analyse bzw. Auswahl. Als Zweites wird zwischen dem Entwurf mit den oben vorgestellten Serverless Architekturmustern und Technologien und deren Implementierung iteriert, um ein schnelles Feedback zu erhalten.

6.1.1 Datenspeicherung- Analyse und Auswahl

Die Gestaltung von Kompetenzmodellen und deren zukünftige Weiterentwicklung hängt stark von den spezifischen Bedürfnissen der jeweiligen Schulen ab. Die möglichen Erweiterungen oder Anpassungen des Modells stellt die Benutzung des einer relationalen Datenbank für KOMA in Frage. In der folgender Tabelle werden die Eigenschaften von relationalen mit ontologischen Schemas verglichen.

Tabelle 6.1: Vergleich relationalem mit ontologischem Tabelle 6.1.1 Schema

Eigenschaft	Relational	Ontologisch
Weltannahme	Existiert nur	Existiert mindestens
Individual	muss Unique	kann >= 1
Info	Ableitung = x	ja
Oritentation	Data	Bedeutung

Das ausgewählte Datendarstellungsformat ist das ontologische Schema, da es den Fokus auf Erweiterbarkeit und semantische Konzepte legt. Im „Semantic Web“ profitieren „Linked Data Driven Web Applications“ von den vernetzten Datenbanken. [Con17b]

Das „Semantic Web“ist eine Erweiterung des herkömmlichen Web, in der Informationen mit eindeutigen Bedeutungen versehen werden [GOS09]. Das World Wide Web Consortium (W3C) spezifiziert eine Zusammenstellung von Standards und best practices für die Mitteilung von Daten und deren semantischer Darstellung. [Bob13]. Diese Bedeutungen werden für Maschinen durch Ontologien dargestellt, welche in „.owl “ Dateien gespeichert werden. [Con17b]

Eine Ontologie ist eine formale Spezifikation über eine Konzeptualisierung [SBF98]. Die Denotation der dargestellten Signifikanten lässt sich durch ihre weltweit eindeutigen Präfix identifizieren z.B: `PREFIX owl: <http://www.w3.org/2002/07/owl#>` [Con17b]. Deren Beziehungen können zu externen Ontologie-signifikanten verweisen und dadurch ein Consensus über Begrifflichkeiten erreichen.

Der Entwurf der Ontologie wurde nach Ontology-Engineering-101 durchgeführt:

Während der Umsetzung wurde Protégé [**?**] als unterstützende Anwendung benutzt.

Um bereits vorhandene Technologien zu nutzen, wurde ein aktueller öffentlicher graphischer ontologischer Entwurf [RMG14] (siehe Abbildung 6.3), der in Moodle mit einer relationalen Datenbasis und PHP umgesetzt wurde, untersucht.

Eine Implementation dieser Ontologie ist jedoch nicht von Watson [Wat17] und LOD [LOD17] auffindbar.

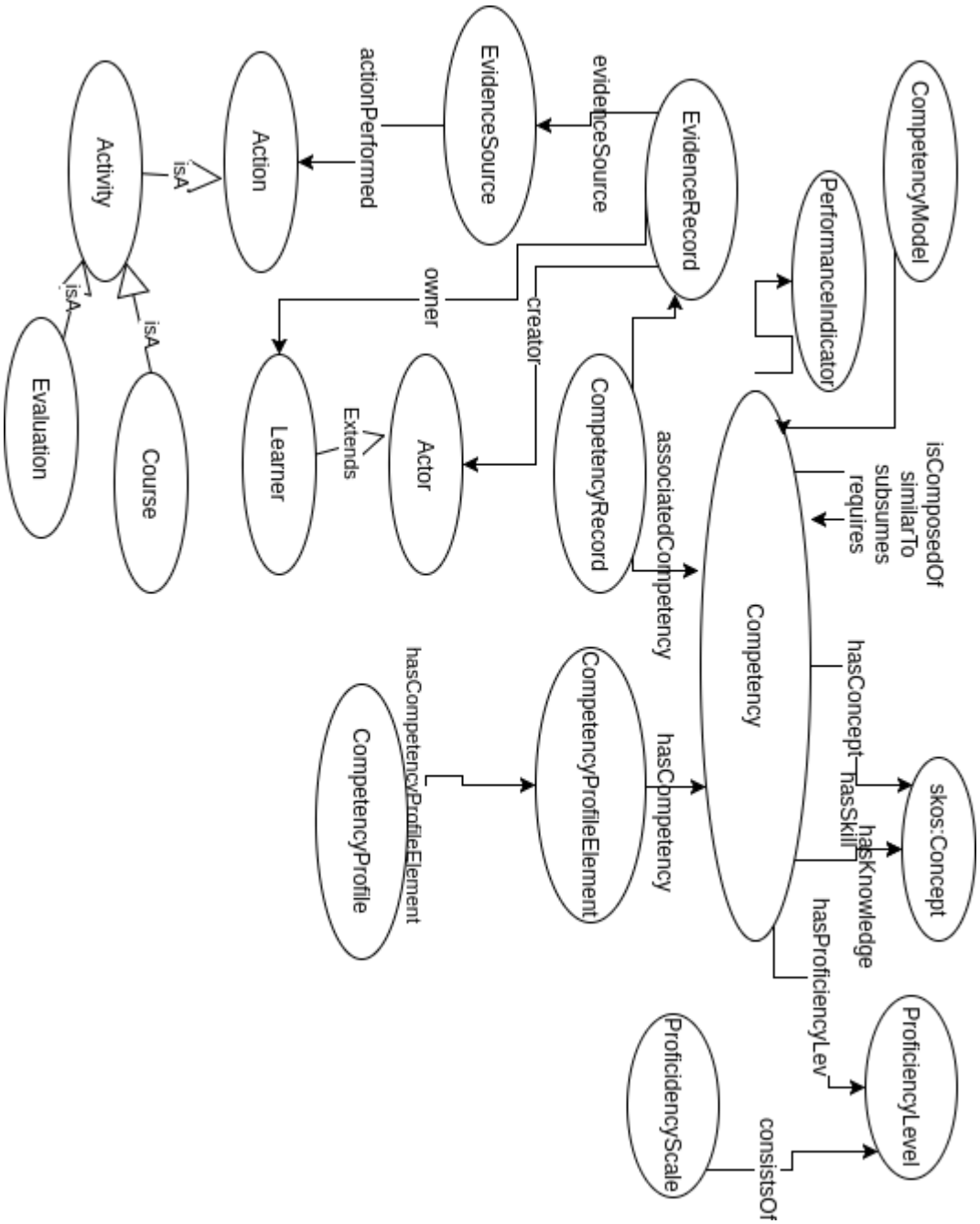


Abbildung 6.3: Kompetenzontologie

Der Entwurf und seine Dokumentation lassen freie Interpretation über Begriffe und deren Zweck, z.B „isComposedOf“, „subsumes“. Ein Standard zur graphischen Darstellung ist zur Zeit noch nicht anerkannt. Graphische Benutzeroberflächen zur Darstellung und zum Entwurf von Ontologien sind derzeit entwickelt, z.B. Graphol [CLSS14].

Daher folgt eine beispielhafte Erklärung, die auf den Anwendungsfall KOMA eine angepasste und ergänzende Interpretation der dargestellten Terminologie des Entwurfs und der RCD (s.u.) liefert.

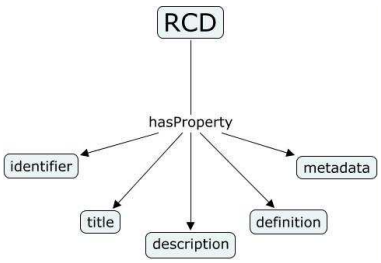


Abbildung 6.4: Reusable Competency Definition

Der EQF wurde für die ontologische Darstellung beschrieben und bietet eine europäisch anerkannte Definition von Kompetenz, die Reusable Competency Definition (RCD) [DCAB17]. Diese ist jedoch nicht in einer veröffentlichen Datenbasis umgesetzt worden.

Die zwei Leitmotive sind auf der einen Seite Kompetenzanforderungen, „die festlegen, über welche Kompetenzen ein Schüler, eine Schülerin verfügen muss, wenn wichtige Ziele der Schule als erreicht gelten sollen. Systematisch geordnet werden diese Anforderungen in Kompetenzmodellen, die Aspekte, Abstufungen und Entwicklungsverläufe von Kompetenzen darstellen“ [Kli03].

Auf der anderen Seite bildet die Definition von Kompetenz eine handlungsleitende Grundlage. Nach Weinert werden Kompetenzen als „die bei Individuen verfügbaren oder durch sie erlernbaren kognitiven Fähigkeiten und Fertigkeiten, um bestimmte Probleme zu lösen, sowie die damit verbundenen motivationalen, volitionalen und sozialen Bereitschaften und Fähigkeiten, um die Problemlösungen in variablen Situationen erfolgreich und verantwortungsvoll nutzen zu können“ [Wei02].

Die bisherige Analyse des Domainproblems wird nun anhand von Protégé in einen Resource Description Framework (RDF) Format bzw. Terse RDF Triple Language (TURTLE) beschrieben. TURTLE besteht aus einer für Menschen lesbaren Syntax und kann sowohl Ontologien in OWL als auch Sparql Abfragen darstellen.

Das folgende Listing 6.1 zeigt eine beispielhafte Darstellung von ontologischen Fakten, den sogenannten „Triples“ . Diese bestehen aus Subjekt, Merkmal und Objekt. Die erste Zeile lässt sich wie folgt interpretieren: dem Namen einer fiktiven Schülerin Alice (Subjekt) wird ein Merkmal in Form einer Kompetenzausprägung (Property) innerhalb eines Unterrichtsfachs, das in Unterrichtsreihen oder -einheiten sequenziert werden kann, wie z.B Mathematik II (Objekt). Diese Instanzen werden Individuals genannt.

Listing 6.1: Darstellung von Triples in TURTLE

```
:Alice :hasCompetency :Math_II .
:EvidenceRecord rdf:type owl:Class .

:actionPerformed rdf:type owl:ObjectProperty ;
  rdfs:domain :EvidenceSource ;
  rdfs:range :Action .
```

Um aus Ontologien Informationen zu entnehmen, wird die Abfragesprache Protocol And RDF Query Language (Sparql) verwendet. Diese ähnelt der traditioneller SQL.

Die einfachste Abfrage in Sparql wählt alle Triples von dem abgefragten Datenmodell (oder Graph) wie in Listing 6.2 gezeigt wird.

Listing 6.2: Sparql SELECT ALL

```
SELECT * WHERE { ?s ?p ?o .}
```

Hilfsvariablen können deklariert werden, um Ergebnisse aus einem Triple als Parameter für das nächste zu benutzen. Die folgende Abfrage ließe sich wie folgt formulieren: „Wähle alle Properties des Graphes und wähle alle dessen Subjekten mit Alice als Objekt“ .

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX koma: <https://s3-us-west-2.amazonaws.com/ontology.thb.de/koma-complex.owl#>
SELECT ?x WHERE {
  ?y rdf:type owl:ObjectProperty .
  ?x ?y koma:Alice .
}
```

Wie in Unterabschnitt 6.0.3 erwähnt wurde, werden Sparqlabfragen durch die API Gateway für Lambda transponiert.

6.2 RESTful API

Der Entwurf von einem benutzerfreundlichen Hypertext Transfer Protocol (HTTP) API beinhaltet die Abstraktion von komplexer Geschäftslogik und Datenverarbeitung in den vier Operationen Create, Read, Update, Delete (CRUD).

Die Komplexität des darunterliegenden Datenmodells erlaubt einer REST Schnittstelle nur einfache Abfragen zu formulieren. [H+17] Daher stellt KOMA zusätzlich einen Sparql-Endpunkt für komplexe Sparql Abfragen zur Verfügung. Die letzte Zeile der Tabelle 6.2 ist eine solche komplexe Abfrage.

Tabelle 6.2: RESTful API

Methode	URL	Rückgabe
GET	/ontology	Information über KOMA
GET	/ontology/{individual}	RDF von Individual
GET	/page	Auflistung von Entitäten
GET	/page/{individual}	Information über diesen Fakt
POST	/sparql	Abfragenergebnis

AWS API Gateway ermöglicht die Definition, Konfiguration und das Importieren von Schnittstellen. Beispielsweise kann der Anfrageparameter in `GET https://<host>/page/Alice}` mithilfe des Musters in Listing 6.3 an den Key „individual “ zugewiesen werden. Hier handelt sich um einen „body mapping template“ für den Inhaltstyp (content type) „application/json“ .

Listing 6.3: API Gateway Request Mapping Template

```
1 https://<host>/page/{individua}
2 ...
3 {
4 "individual" : "$input.params('individual')"
5 }
```

Nachdem eine Anfrage eingetreten ist und deren Parameter transponiert sind, erfolgt ein synchroner Aufruf der zuständigen Lambdafunktion. Die Verwaltung dieser Instanziierung entspricht dem Priority Queue Muster in Abschnitt 4.7, wobei die API Gateway der Queue und die Lambdafunktion dem Consumer entspricht.

Das folgende Listing 6.4 zeigt die Implementierung einer Lambdafunktionsfassade, die für die oben genannte URL zuständig ist.

Listing 6.4: Lambda Javascript Funktionsfassade

```
1 ...
2 exports.handler = function (event, context, callback) {
3   reqIndividual = event.individual; // check possible exception
4   async.waterfall([createBucketParams
5           , getS3ObjectBody
6           , parseOntology
7         ],
8         function (err, result) {
9           if (err) {
10            callback(res.createErrorResponse(500, err));
11          } else {
12          if (Object.keys(result).length === 0 && result.constructor === Object) {
13            callback(null, res.createErrorResponse(404, "there was no result on the
                  search"));
14          } else {
15            callback(null, res.createSuccessResponse(result));
16 ...
```

Die zweite Zeile entspricht der Fassadensignatur von Lambda für Javascript. Der erste Parameter, „event“, enthält in diesem Fall Informationen über die ursprüngliche Anfrage. Der Zweite, „context“ erlaubt den Zugang auf die Laufzeitumgebung von Lambda und kann definieren, ob die Ausführung erfolgreich (`context.success(Object result)`), fehlerhaft (`context.fail(Error error)`) oder beides (`context.done(Error error, Object result)`) war.

In der vierten Zeile befindet sich der Aufruf `async.waterfall(...)`. Dieser entspricht dem Waterfall Muster dessen Verarbeitungsschritte in der Abbildung 6.5 dargestellt werden. Dieses Muster erlaubt es, eine Reihe von Funktionen so zu verketten, dass das Ergebnis einer Funktion der Eingabeparameter der Nächsten ist. Wenn ein Fehler in einer Funktion entsteht, hält der Wasserfall an.

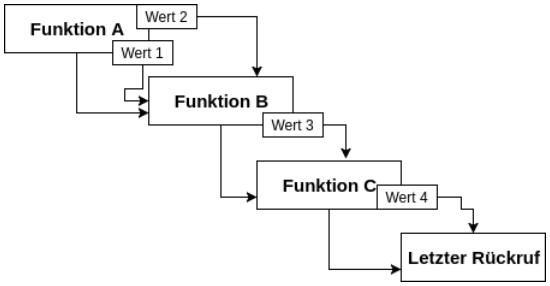


Abbildung 6.5: Waterfall Muster

Um die Wiederverwendbarkeit zu gewährleisten, ist die Abtrennung der Abhängigkeiten

in Bibliotheken der Lambdafunktion sinnvoll. Dieser Ansatz wird von dem SOA-Architekturstil Kapitel 4 gefordert.

Bei einer Abfrage an die `POST https://<host>/sparql` URL werden die Abfrageparameter ohne die Konfiguration von „body mapping“ weiter an die zuständige Lambdafunktion geleitet.

Die Funktion hinter dieser URL, wie in Listing 6.5 gezeigt, ist in Java 8 implementiert. Sie fordert die Implementierung der Schnittstelle RequestHandler. Die Generics-Parameter definieren den Klassentyp der Anfrage und des Ergebnisses, wie es im Fall der Abbildung „RequestClass“ als Anfrage und „String“ als Ergebnis ist.

Der Nutzer kann Umgebungsvariablen bei der Konfiguration von Lambdafunktionen anlegen und während Laufzeit auf sie zugreifen, z.B in der Zeile 7 mittels `System.getenv(StringConstante)`.

Listing 6.5: Lambda Sparql-Endpoint

```
1 public class Handler implements RequestHandler<RequestClass, String> {
2 ...
3 @Override
4 public String handleRequest(RequestClass input, Context context) {
5   context.getLogger();
6   request = input;
7   return new Controller(System.getenv(ENV_BUCKET), Regions.US_WEST_2.getName())
8     .executeQuery(request.getQuery(), request.getBucketKey());
9 }
```

Die Klasse „Controller“ (Zeile 7) benutzt die Jena ARQ Bibliothek, um Sparqlabfragen auszuführen. Ein Beispiel ihrer Benutzung wird in Listing 6.6 dargestellt.

Die Abfrage an die Datenbasis lässt sich mit `request.getQuery()` (Zeile 8) aus dem Bodyparameter „query“ extrahieren.

Listing 6.6: Lambda Sparql-Endpoint-Controller

```
1 ObjectMapper mapper = new ObjectMapper();
2 QueryResultWithMap resultWithMap =
3 new QueryResultWithMap();
4 Map<String, String> tmp;
5
6 Query query = QueryFactory.create(aQuery);
7 try (QueryExecution qexec = QueryExecutionFactory.create(query, model)) {
8
9 ResultSet results = qexec.execSelect();
```

```
10  while (results.hasNext()) {
11  tmp = new LinkedHashMap<>();
12
13  QuerySolution soln = results.nextSolution();
14
15    for(String v : results.getResultVars()){
16      RDFNode node = soln.get(v);
17      tmp.put(v, node.isResource() ?
18           soln.getResource(v).getLocalName() :
19           soln.getLiteral(v).getString());
20      }
21    resultWithMap.getBody().add(tmp);
22
23  }
24  toClient = mapper.writeValueAsString(resultWithMap);
25 ...
```

Nach der Ausführung der Abfrage (Zeile 7) werden die Ergebnisse in Zeile 15 iteriert und anschließend wird eine Referenz zu einem JSON-Objekt zurück geliefert.

Mit der bisher beschriebenen Implementierung der Webanwendung können Daten abgefragt und verarbeitet werden. Diese Ergebnisse stellen ein Feedback dar, womit Konzepte (siehe Kapitel 1) bewiesen, ausgewertet und weiterentwickelt werden können.

Ein wesentliches Merkmal von den oben beschriebenen Lambdafunktionen ist, dass sie für jeden Aufruf die komplette Datenbasis neu herunterladen. Für die Gebrauchstauglichkeit der Anwendung werden im Folgenden Verbesserungsansätze erläutert.

Caching zielt auf die Verbesserung der Performanz und Skalabilität eines Systems ab, in dem es die oft gelesenen Daten temporär zwischenspeichert. Daher können wiederholte Zugriffe auf die Ontologie in einem Cache gepuffert werden. Hierzu bietet AWS „ElastiCache“ als Serverless Dienst. Es gibt zwei Typen von Software Cache: „Shared-Cache“ und „In-Memory“ [HSB+14]. Der Letztere teilt die Umgebung des Consumers (Lambda in diesem Fall) mit. Da die Laufzeitumgebung von Lambda ohne Vorwarnung wechseln kann und nur über nicht persistenten Speicher verfügt [AWS17c], lässt sich die In-Memory Cachevariante nicht konsistent implementieren.

Ein schnellerer Zugriff wird durch die Allokation eines Caches zwischen der Datenbasis und den Lambdafunktionen (siehe Abbildung 6.6) gewährleistet. Da die Consumer

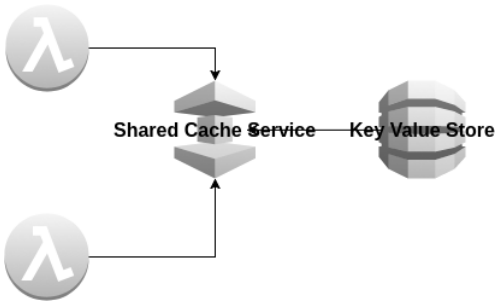


Abbildung 6.6: Shared Cache Muster

den gleichen „Snapshot“ der Daten abrufen, sinkt die Anzahl der inkonsistenten Ergebnisse. [HSB+14]

Mit der Benutzung der Anwendung werden voraussichtlich die OWL Dateien größer und somit die Latenz bei Abfragen verlängert.

Eine zusätzliche Maßnahme zur Optimierung des Datenzugriffs ist die Datenpartitionierung. Diese kann je nach Benutzungsmuster horizontal, vertikal oder funktional erfolgen.

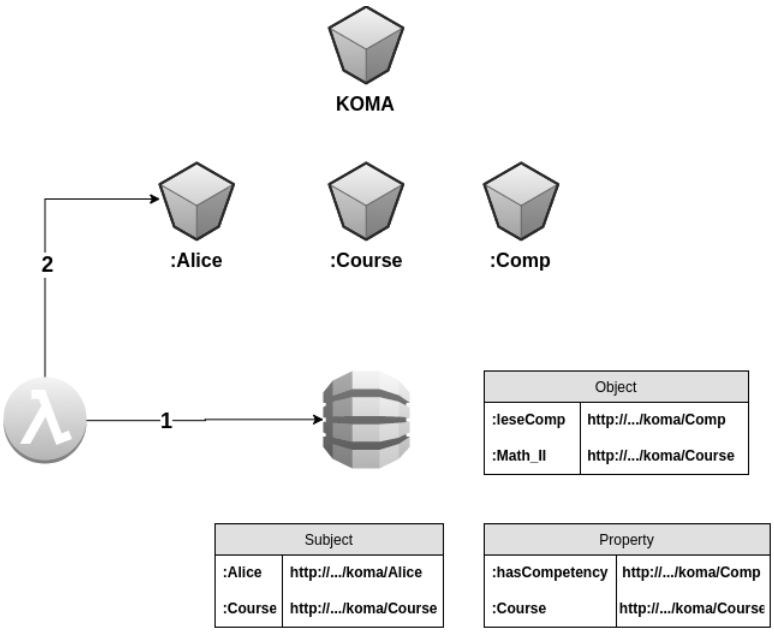


Abbildung 6.7: Partitionierung der Datenbasis

Im Folgenden wird die Anwendung als Beispiel zur Erklärung genutzt.
Bei der horitzontalen Datenpartitionierung wird die OWL Datei nach „Individuals“

aufgeteilt und in unterschiedliche Buckets einsortiert. Hingegen wird bei einer vertikalen Datenpartitionierung jeweils eine Tabelle für Subjekte, Merkmale (Properties) und für Objekte erstellt. DynamoDB eignet sich für diese Serverless Anwendung als „Key-Value Store“ , um als Keys die „Individuals“ und als Values dessen Universal Ressource Identifier (URI)s darzustellen. [AMMH07]

Die funktionale Partitionierung richtet sich nach Benutzungsfällen oder Geschäftsprozessen (bzw. Bounded Context). Die OWL Detei ließe sich beispielsweise nach der häufigsten Ergebnissen aufteilen.

Cloud Dienste werden oft in unterschiedlichen Datenzentren oder Regionen eingesetzt. Um die Verfügbarkeit, Performanz und Konsistenz zu maximieren und die Datenübertragungskosten zu minimieren, kann eine Master-Datenbasis mit erlaubten Lese- und Schreiboperationen und eine Subordinate-Datenbasis nur mit Leseoperationen definiert werden. [HSB+14]

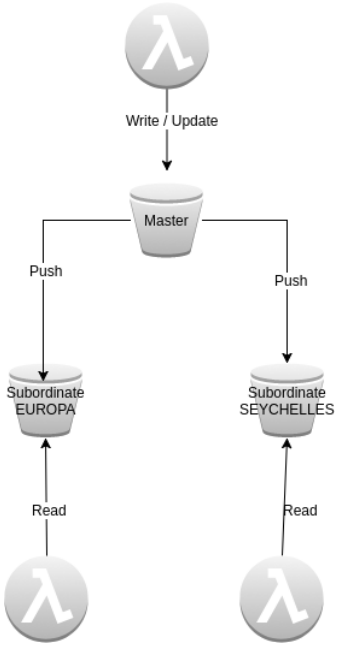


Abbildung 6.8: Replizieren der Datenbasis

Der Master pusht (one-way Synchronisierung) die Daten an seine Subordinate-Repliken. Diese Segregation favorisiert Lesezugriffe und vereinfacht die Datensynchronisierung, da sie nur in einer Richtung erfolgen. Bei angeforderter Auswahl der zu replizierenden Daten priorisiert man die statischen vor den dynamischen Daten.

Latenz verkürzt sich bei der Zuteilung einer Replik (siehe Abbildung 6.8) in der physischen Nähe des Consumers.

Viele

6.3 Single Page Application

Da KOMA ohne Vorkenntnisse zu benutzen sein soll, lässt sich die Entscheidung über die Art der Benutzeroberfläche leicht treffen. Die milliardenfache Nutzung von Web Browsern macht sie zum Favoriten.

Die Web Anwendung ist für alle Rechenaufgaben verantwortlich, die im Browser sicherheitstechnisch keine Gefahr darstellen, um das Backend oder den Server möglichst wenig auszulasten. Deswegen bietet sich eine Single Page Application an. Die SPA besteht aus einem einzigen HTML Dokument. Dadurch vereinfacht man die Konfiguration der Authentifizierung und unterbricht den Fluss der UI-Darstellung zwischen Seiten nicht.

Ein konfiguriertes Anfangsprojekt/Quickstart kann mithilfe von Initializr [Ini17] oder JHipster [Jhi17] schnell angelegt werden. Für die lokale Entwicklung der Webseite werden anhand von NodeJS und NPM folgende Bibliotheken als Abhängigkeiten verwaltet: Bootstrap als Stylescheet und jQuery als Javascript-Bibliothek. Die Webseite wird statisch mittels S3 geliefert. Dies geschieht mit folgendem Befehl:

Listing 6.7: Webseite veröffentlichen

```
1 $ aws --region us-west-2 s3 website --index-document index.html --error-document error.
      html 's3://koma.thb.de'
```

Da der Zugriff auf die Datenspeicherung gesichert werden soll, wird die Login-Funktionalität hinzugefügt. In Abbildung 6.2 kann eine Übersicht der nötigen Schritte zur Anmeldung bei AuthO, zur Authentisierung und zur Autorisierung erhalten.

Der Nutzer meldet sich im ersten Schritt mit dem importierten Script `<script src="https://cdn.auth0.com/js/lock-9.min.js">` und der von KOMA mit gelieferten Konfiguration (siehe Listing 6.8) an.

Listing 6.8: Auth0 Anmeldung

```
1
2 auth0Lock = new Auth0Lock(config.auth0.clientId, config.auth0.domain);
3
4 auth0Lock.getProfile(idToken, function (err, profile) {...}
```

Nach der erfolgreichen Anmeldung erhält der Nutzer einen JSON Web Token (JWT), um ihn bei weiteren Anfragen kodiert (in JSON Web Signature (JWS)) vorzuweisen. Dies geschieht mithilfe von der XMLHttpRequest (XHR) Schnittstelle. In diesem Fall wurde deren Implementierung von jQuery verwendet.

Listing 6.9: Remote Procedure Call mit XHR Schnittstelle

```
1 $.ajaxSetup({
2   'beforeSend': function (xhr) {
3   xhr.setRequestHeader('Authorization', 'Bearer ' + localStorage.getItem('userToken'));
4   ...
```

Die Lambdafunktion, gekennzeichnet mit „UserProfile“ , dekodiert den erhaltenen JWT durch einen Schlüssel, der in Umgebungsvariablen (siehe Abbildung 6.2) angelegt wurde. Die Überprüfung erfolgt mit einer Abfrage an AuthO (siehe Listing 6.10 in Zeile 2).

Listing 6.10: Authentisierung in Lambda

```
1 var secretBuffer = new Buffer(process.env.AUTH0_SECRET);
2 jwt.verify(token, secretBuffer, function(err, decoded){
3 ...
4 url: 'https://'+ process.env.DOMAIN + '/tokeninfo',
5 ...
```

Dieses Vorgehen bei der Authentisierung (siehe Abbildung 4.11) basiert auf dem Vertrauen zwischen Cloudanbietern. Der Nutzer kann für mehrere Anwendungen die gleichen Zugangsdaten benutzen. Die Anmeldung erfolgt über den Identitätsanbieter (wie z.B Google oder Github).

Der autorisierter Zugriff kann nun gewährleistet werden, da die Verwaltung bzw. die persistente Speicherung von Nutzerprofilen möglich ist.

7 Auswertung

Zur Skalabilität Die Partitionierung und das Replizieren von Datenbanken bringt zwar Vorteile (wie in Abbildung 6.2 gezeigt wurde), aber auch Herausforderungen bei der Synchronisation und Konsistenz der Daten mit sich. Dies geschieht auf Grund des konsistent zu haltenden Zustandes der Datenbanken.

Die Relevanz und Bedeutung der Daten hängt von dem Standpunkt ab. Z.B ein Schüler möchte schnell zu erfahren, ob der Hochladevorgang seiner Arbeit erfolgreich war (Schreibzugriff vorziehen). Ein Professor überprüft, ob alle Schüler ihre Arbeit hochgeladen haben (Lesezugriff vorziehen). Daher lässt sich eine technologisch orientierte Strategie unklar definieren, da weder alle Faktoren bekannt, noch beachtet werden können.

Die Unmöglichkeit, alle Perspektiven zu berücksichtigen, erfordert eine Entscheidung ob ein Service entweder eine Funktion oder eine Datenbank ist; auf die Gefahr hin, dass nicht alle Benutzerinteressen umgesetzt werden, obwohl der Begriff Service in „Service Oriented Architecture“ als ein Mehrwert für die Benutzer verstanden wird.

Um eine möglichst hohe Performanz zu erzielen, wird die Anwendung entsprechend der Benutzungsmuster gestaltet. Die funktionale Datenbankpartitionierung erfüllt, wenn korrekt implementiert, diese Forderung.

Da ein Event beispielsweise eine Interaktion zwischen dem Kunden und der Anwendung repräsentiert, eignet sich die EDA für die Steuerung von Benutzungsfällen besonders gut. Deren Orchestrierung lässt sich mittels einer endlichen Zustandsmaschine definieren und einem Petrinetz darstellen, in dem Events die Kanten und FaaS die Ecken sind. Dieser Ansatz ähnelt dem Domain getriebenen Entwurf (Domain Driven Design [MA07]), in dem Benutzungsfälle in Kontextgrenzen (Bounded Context) zusammengefasst werden.

Frameworks und FaaS Frameworks helfen bei der Entwicklung von komplexen Systemen, in dem sie Abstraktionsschichten hinzufügen und damit komplexe Berechnungen wie bei der „Dependency Injection“ von Java Enterprise Edition (JEE) übernehmen.

Für KOMA wurde kein Framework benutzt, da die Implementierung der Funktionen dem SRP folgte und der Abstraktionsschichten nicht bedurften. Hingegen wurden die unten beschriebenen Tests durchgeführt, um die Verwendbarkeit von Frameworks innerhalb einer Lambdafunktion auszuwerten.

Das ausgewählte Framework ist Apache Jena [Apa17], es wird als ein eingebetteter Server eingesetzt. Dieser unterstützt den Ausbau von Semantic Web und Linked Data Anwendungen.

Das Listing 7.1 zeigt die vorgenommene Konfiguration für den Aufbau des Servers. Er wird während des Tests die schon vorgeladene KOMA Ontologie durchgehen und ausloggen, um anschließend wieder zu stoppen.

Listing 7.1: Embedded Server in Lambda

```
1 Dataset ds = DatasetFactory.createTxnMem();
2 server = FusekiServer.create()
3 .setPort(3030)
4 .add("/dataset", ds)
5 .build();
6 server.start();
```

Die Abbildung 7.1 zeichnet die Ergebnisse der oben genannten Tests auf.

Die Ergebnisse zeigen, wie sich die JVM in der Lambdalaufzeitumgebung initiiert (siehe Abschnitt 5). Da die Laufzeitumgebung für jede Anfrage eine neue Lambdafunktion instantiiert, muss der Server neu gestartet werden. Hierzu beweist die Grafik, dass IaaS und PaaS besser für längerfristige Prozesse als Lambda geeignet sind (siehe Tabelle 3).

Die bereitgestellte Speicherkapazität beschleunigt die Ausführung und dadurch verkürzen sich die Antwortzeiten. In diesem Szenario nutzt die Transaktionsverwaltung und die „In-Memory“ Caching des Servers nicht.

Bei zu niedrigen vorkonfigurierten Speicherkapazitäten sind die in der Tabelle 7.1 zusammengefassten Fehler zu beobachten:

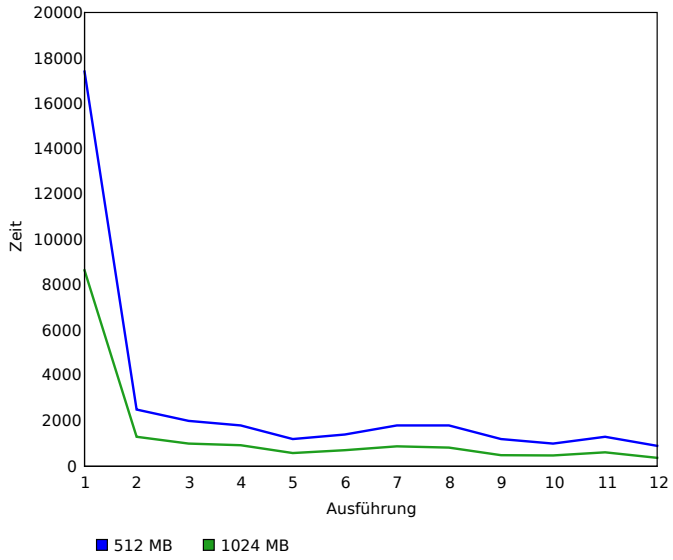


Abbildung 7.1: Ausführungszeiten für Fuseki Server in Lambda

Ressourcen (MB)	Ausführungszeit (ms)	Benutzte Speicher (MB)
256	26100	101
128	Fehler	java.lang.OutOfMemoryError

Tabelle 7.1: Tests mit Frameworks in Lambda Ergebnisse

Allerdings verursacht die Rekonfiguration keinen neuen Einsatz (Deployment), damit werden Ausfallzeiten vermieden.

An dieser Stelle empfiehlt es sich, dass in Abbildung 4.5 vorgestellte Legacy API Architekturmuster zu benutzen. Somit würden Anfragen, die eine Transaktionsverwaltung benötigen, an den Server weitergeleitet und die Latenz verkürzt.

Durch die Ersetzbarkeit der FaaS ist hier erreicht, dass die Entwickler und die Anwendung von der darunterliegenden Technologie entkoppelt werden.

Die Abbildung 7.2 zeigt die Ausführungszeiten der Abfrageverarbeitung des Servers und der Jena ARQ Bibliothek.

Die serverlose Variante (Jena ARQ, in Orange) ist erst ab der zweiten Ausführung deutlich schneller als mit Fuseki (in Blau).

Sowohl zwischen der 5. und der 7. als auch der 10. und 11. Ausführung steigt die Latenz. Nach Ergebnissen von Herrn McGrath und Brenner lässt sich dieses Phänomen auf „kalte Startzeiten“ (siehe Abschnitt 5) zurückführen . Dabei zeigten Google Cloud

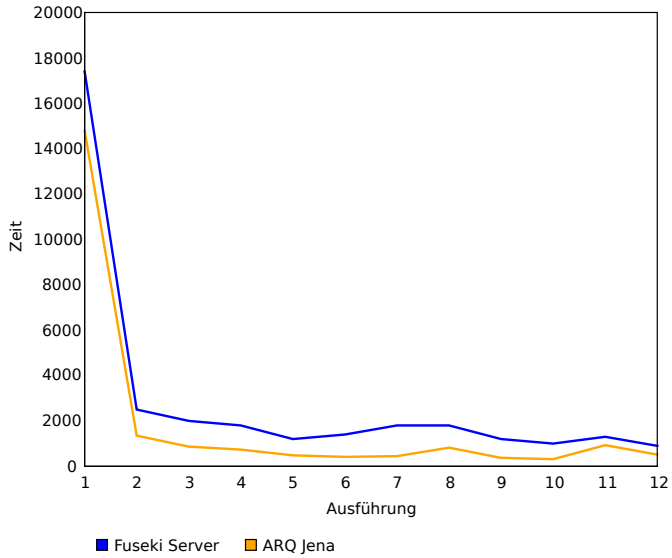


Abbildung 7.2: Ausführungszeiten für Fuseki und Jena ARQ

Funktions und Lambda deutlich weniger Schwankungen bei den Latenzmessungen. [MB17]

Große Bibliotheken Die gebaute JAR Datei der oben erwähnten Lambdafunktion, Jena ARQ, beträgt 19 MB. Die Ladezeiten verlangsamen sich, je größer die auszuführende Lambdafunktion ist [OYH+17]. Wissenschaftliche Bibliotheken wie „SciPy“ (für die Programmiersprache Python) übersteigen die maximale Größe einer Lambdafunktion (50 MB). Daraus ergibt sich, dass eine Verwaltung für große Bibliotheken gerade dann nötig ist, wenn auch der Benutzungsfall sich nicht für containerbasierte Anwendungen gut eignet.

Das Verwaltung von großen Bibliotheken wird derzeit zwar nicht von AWS Lambda unterstützt, dafür aber von „Pipsqueak“ (ein package-bewusst Berechnungsplattform für OpenLambda [OYH+17]). Die Autoren schlagen den Ausbau einer „Shared-Cache“ innerhalb der Lambdalaufzeitumgebung vor, die während der Laufzeit Bibliothekenabhängigkeiten an Lambdafunktionen liefert.

DevOps Frameworks Die Anzahl von Diensten bzw. Funktionen kann in einem komplexen System unübersichtlich werden. Entsprechend steigt der Aufwand bei Deployments, der Verwaltung und der Kohäsion von Abhängigkeiten, sowie bei der Koordination des Entwicklerteams (z.B. Wer entwickelt was wann?).

Folgende Tabelle 7.2 [FIMS17] zeigt, dass die hohe Granularität des Einzatzes der Serverless Architekturen Automatismen unweigerlich mit sich führt.

	Vor Ort	VMs	Containers	Serverless
Bereitstellungszeit	bis Monaten	Minuten	bis Minuten	Millis.
Benutzung	Gering	Hoch	Hoher	die Höchste
Ladegranularität	Keine	Stunden	Minuten	Blöcke von Millis.

Tabelle 7.2: Vergleich zwischen von Verwendung und Deployment

Diese Schwierigkeit macht die Deploymentframeworks wertvoll, da sie nicht nur die oben genannten Probleme ansprechen, sondern auch den Umgang mit anbieterspezifischen Sicherheitsrichtlinien vereinfachen. Dazu sind „Serverless Framework“ und „Zappa“ die bekanntesten Beispiele.

AWS ermöglicht es, den Entwicklungsstand der Dienste zu bezeichnen, z.B „Dev“ oder „Produktion“ und Dienste in Browsern einzeln zu testen. Dies ist nur bedingt ausreichend, denn Systemtests sind noch nicht möglich. Somit gewinnt die Deployment-Strategie an Bedeutung.

Zur Entwicklung Die starke Komponentisierung und Dezentralisierung von Software, die Variabilität von Programmiermodellen, Frameworks, Tools, Sprachen und deren Entwicklungsumgebung erhöht die Komplexität des Entwicklungszyklus und weckt einen Bedarf an Tools zur Automatisierung von Deployment, Konfiguration und Tests. Ein klar definierter Handlungsplan bei der Softwareentwicklung erleichtert das Abstrahieren von zu bearbeitenden Details.

Die DevOps Kultur gibt dazu Hilfestellungen. Neben dem Entwurf der Softwarearchitektur ist es nötig, um deren Umsetzung innerhalb des zeitlich vorgegebenen Rahmens zu gewährleisten, eine zum Projekt passende DevOps Strategie aufzustellen. Um einen Vorteil aus den neuen Technologien zu ziehen, ist die Recherche nach schon existierenden DevOps Frameworks besonders wichtig. Ihre Integration in die DevOps Strategie sind einer agilen Entwicklung dienlich.

Zu Serverless Die unterschiedlichen Interpretationen des Begriffs Serverless führen mitunter zu kreativen Ansätzen wie die Containerisierung einer Lambdalaufzeitumgebung. Dies ist vorteilhaft, weil dadurch alle Programmiersprachen unterstützt werden können.

Kosten und Performanz zwischen Lambda und Server Auf der kleinsten Ebene rechnet Lambda nach zeitlichen Abschnitten von 100 ms ab. Da viele RPC kürzer als 100 ms lang leben [Sri95], wird laut den Autoren von [HSH+16] ein 3.7 fache Zunahme an Abrechnungen verzeichnet.

Da die PaaS und IaaS pro Stunde abgerechnet werden, FaaS hingegen pro Anfrage, ergibt sich daraus, dass der Preis und die Performanz je Anfrage, umgekehrt proportional zu ihrer Anzahl ist.

Nachteile Da HTTP und Lambda zustandslos sind, können die Sitzungen der Nutzer derzeitig nur mit Hilfe von Datenbanken implementiert werden. Interaktionen, die auf älteren basieren, wurden im Front-End berücksichtigt. Obwohl Zustände von Lambdafunktionen innerhalb eines „Workers“(siehe Abbildung 5.1) sichtbar sind [HSH+16], wird die Kontrolle auf diese nicht gewährleistet.

Der Einsatz von großen Mengen von Quellcode verlangsamt die Ladezeiten des Servers (siehe Abbildung 5.1), u.d. verschlechtert sich die Latenz der HTTP-Response.

Die Hardwareressourcen von Lambda skalieren zwar vertikal, aber nur proportional. Entsprechend werden diese nicht genutzt, aber abgerechnet.

Die Latenz bei der Bereitschaft auf neue Anfragen ist bei Lambda höher als bei PaaS, daher sind Mechanismen zur Optimierung von Antwortzeiten nötig, z.B: eine periodische Ausführung zur Wiederverwendung der Lambdalaufzeitumgebung, oder das Ableiten einer Message Queue (Nachrichtenschlange) von einer überlasteten PaaS Anwendung zu Lambdafunktionen.

Es besteht ein Lock-in Risiko, wenn die Portierung von Quellcode auf unterschiedliche Cloudanbieter nicht im Voraus eingeplant wurde.

Der Kontrollverlust über das Verhalten der Serverless Dienste kann die Entwurfsentscheidungen erschweren.

Die AWS-Umgebung lässt das Debuggen von Lambdafunktionen nicht zu, obwohl Werkzeuge zum lokalen Testen und Debuggen diese Umgebung emulieren.

Je spezifischer sich die Anforderungen gestalten, desto mehr Eigenschaften der zugrundeliegenden Technologie sind zu beherrschen, z.B.: eine Benutzer definierte Erweiterung der Funktionalität einer Datenbank durch Lambda.

Vorteile Die automatische Skalierung erleichtert das Wachstum einer Webanwendung, denn sowohl deren Erfolg als auch die Anzahl von Anfragen ist nicht einfach vorherzusehen.

Durch die garantierte Lieferung von Nachrichten (siehe Abschnitt 5) können auch stark skalierte Webanwendungen von der Fehlertoleranz profitieren.

Der zeitliche Aufwand bei dem Planen von Ressourcenbereitstellung hat sich, wegen der automatischen Skalierung und fast exakten nutzungsabhängigen Abrechnung, aufgelöst.

So lange ein Architekturentwurf vorhanden ist, können sich die Entwickler auf den Kern des Quellcodes konzentrieren.

8 Ausblick

Da der Architekturentwurf komplexer wird, bietet sich die Implementierung von unterstützenden Werkzeugen an, die einen kohärenten Entwurf z.B. durch Petrinetzmodellierung erleichtern.

Die Komposition von unbekannten Inhalten in einer Benutzeroberfläche, wie es der Fall bei einer „Linked Data Driven“ Webanwendung ist, lässt sich mittels REST Schnittstellen gestalten. Da Dienste von Drittanbieter aufgerufen und deren Inhalte eingebettet werden können. Die Konfiguration ihrer Darstellung kann mit Parametern in der Anfrage festgelegt werden.

Der Bedarf an Werkzeugen für die Entwicklung, den Einsatz, das Testen usw. besteht, wie in 7 erläutert wurde, auch mit dem Serverless Ansatz weiterhin.

Die Vereinigung von Edge- mit Cloud-Computing durch die Verwendung von FaaS kann ein neues Potenzial für die Entwicklung von IoT darstellen. Der Begriff „Edge-Computing“ bezeichnet die Verlagerung von Rechenleistung, Anwendungen, Daten und Services an die Endpunkte eines Netzwerkes [Wan17]. Diese Verlagerung wurde bei der Umsetzung von KOMA in Abschnitt 6.3 in Form einer SPA implementiert.

Die starke Entkopplung (siehe Abbildung 4) von Diensten, die aus der Umwandlung von einer monolithischen in eine dezentralisierte Architektur entsteht, bringt mehr Komplexität, mehr Flexibilität und die Entkopplung des Entwicklers von der zugrundeliegenden Technologie mit sich. Daher gewinnt die Entwurfsphase eines Projekts bei Serverless Architekturen an Bedeutung, um die Kohäsion von Diensten zu gewährleisten. Als Konsequenz tritt die Modellierung und Entwicklung der Problemdomain in den Vordergrund.

An dieser Stelle ist es hilfreich zwischen einer zeitlichen und einer technologischen Flexibilität zu differenzieren. Die Zeitliche kann erreicht werden, in dem die Kernfunktionalität einer Anwendung in FaaS Dienste kapselt, um einen konzeptionellen Beweis

durchzuführen; die Technologische durch die oben vorgestellten Architekturentwurfsmuster (siehe Kapitel 5) . Da die unter EDA integrierten Dienste von einander stark entkoppelt sind, lassen sie sich einfach aggregieren.

Listings

Tabellenverzeichnis

Literaturverzeichnis

[AMMH07] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 411–422. VLDB Endowment, 2007.

[Apa17] Apache. Apache jena framework, 2017.

[AWS17a] AWS. Aws elastic beanstalk - paas application management, 2017.

[AWS17b] AWS. Aws lambda – cloudwatch, 2017.

[AWS17c] AWS. Aws lambda – häufig gestellte fragen, 2017.

[Bob13] DuCharme Bob. Learning sparql. sl, 2013.

[Cha14] K Chandrasekaran. *Essentials of cloud computing*. CRC Press, 2014.

[CLSS14] Marco Console, Domenico Lembo, Valerio Santarelli, and Domenico Fabio Savo. *Graphical Representation of OWL 2 Ontologies through Graphol*, volume 1272. 10 2014.

[Con17a] ConnectWise. Connectwise, 2017.

[Con17b] World Wide Web Consortium. World wide web consortium, 2017.

[DCAB17] Diego Duran, Gabriel Chanchí, Jose Luis Arciniegas, and Sandra Baldassarri. A semantic recommender system for idtv based on educational competencies. In *Applications and Usability of Interactive TV*. Springer, January 2017.

[EQF17] EQF. Eqf, 2017.

[FIMS17] Geoffrey C Fox, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Status of serverless computing and function-as-a-service (faas) in industry and research. *arXiv preprint arXiv:1708.08028*, 2017.

[Fou17] The Apache Software Foundation. Arq - a sparql processor for jena, 2017.

[Fow17a] M Fowler. Serverless architectures, 2017.

[Fow17b] M Fowler. Serverless architectures, 2017.

[Goo17a] Google. Docker, 2017.

[Goo17b] Google. Docker monitoring, 2017.

[GOS09] Nicola Guarino, Daniel Oberle, and Steffen Staab. What is an ontology? In *Handbook on ontologies*, pages 1–17. Springer, 2009.

[H+17] II Hunter et al. Advanced microservices: A hands-on approach to microservice infrastructure and tooling. 2017.

[HSB+14] A. Homer, J. Sharp, L. Brader, M. Narumoto, and T. Swanson. *Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications*. Patterns & practices. Microsoft Developer Guidance, 2014.

[HSH+16] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. *Elastic*, 60:80, 2016.

[HW04] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[Ini17] Initializr. Initializr, 2017.

[Jhi17] Jhipster. Jhipster, 2017.

[Kin16] W. King. *AWS Lambda: The Complete Guide to Serverless Microservices - Learn Everything You Need to Know about AWS Lambda!* AWS Lambda for Beginners, Serverless Microservices Series. CreateSpace Independent Publishing Platform, 2016.

[Kli03] Eckhard Klieme. ua: Zur entwicklung nationaler bildungsstandards–eine expertise. *Berlin 2003*, 2003.

[LOD17] LOD. Lod, 2017.

[MA07] Floyd Marinescu and Abel Avram. *Domain-driven design Quickly*. Lulu. com, 2007.

[MB17] Garrett McGrath and Paul R Brenner. Serverless computing: Design, implementation, and performance. In *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on*, pages 405–410. IEEE, 2017.

[Net17] Netflix, 2017.

[OYH+17] Edward Oakes, Leon Yang, Kevin Houck, Tyler Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Pipsqueak: Lean lambdas with large libraries. In *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on*, pages 395–400. IEEE, 2017.

[Ray13] Nilanjan Raychaudhuri. *Scala in action*. Manning Publications Co., 2013.

[RB] Jürgen Dunkel Ralf Bruns. *Event-Driven Architecture: Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse*.

[RMG14] Kalthoum Rezgui, Hédia Mhiri, and Khaled Ghédira. Extending moodle functionalities with ontology-based competency management. *Procedia Computer Science*, 35:570–579, 2014.

[SA17] Sachsen-Anhalt. Sachsen-anhalt, 2017.

[Sba17] P. Sbarski. *Serverless Architectures on AWS: With Examples Using AWS Lambda*. Manning Publications Company, 2017.

[SBF98] Rudi Studer, V Richard Benjamins, and Dieter Fensel. Knowledge engineering: principles and methods. *Data & knowledge engineering*, 25(1-2):161–197, 1998.

[Sri95] Raj Srinivasan. Rpc: Remote procedure call protocol specification version 2. 1995.

[Sta07] Gernot Starke. *SOA-Expertenwissen: Methoden, Konzepte und Praxis serviceorientierter Architekturen*. dpunkt, 2007.

[UNL17] UNLESS, 2017.

[Wan17] Wolfgang Wanner. Cloud- versus edge-computing, 2017.

[Wat17] Watson. Watson, 2017.

[Wei02] F.E. Weinert. *Leistungsmessungen in Schulen*. Beltz Pädagogik. Beltz, 2002.

[You15] Marcus Young. *Implementing Cloud Design Patterns for AWS*. Packt Publishing Ltd, 2015.