Technology
Arts Sciences
TH Köln

Microservices-Architektur für webbasierte Enterprise-Anwendungen

Guided Project B09 - Evaluationsbericht

ausgearbeitet von

Christopher Beyerlein
Martin Breidenbach
Sebastian Domke
Pascal Dung
Daniel Felten
Jonas Frenz
Daria Grafova
Anatoli Neuberger
Mohammed Obaidi
Christian Olsson
Christoph Stephan
Yasa Yener

TH Köln
Campus Gummersbach
Fakultät für Informatik

im Studiengang

Computer Science Master

vorgelegt bei: Prof. Dr. Stefan Bente

Gummersbach, 30. August 2016

Inhaltsverzeichnis

Teil I.

Einleitung

Organisation des Projekts

1. Ziel des Projektes

Der Gegenstand dieses Projektes ist die praktische Evaluierung des Microservices-Architekturparadigmas. Es sollen die Vorteile und die Nachteile einer Microservice Architektur ermittelt und aufgezeigt werden. Zu diesem Zweck wurde mit theoretischen Überlegungen begonnen. Im Anschluss wurden diese Überlegungen an einem Proof-of-Concept überprüft. Die Anwendung des Proof-of-Concept ist Teil eines fiktiven Restaurants und ist das Resultat vergangener Projekte. Der Fokuspunkt der Arbeit am Proof-of-Concept war die Aufteilung der Restaurantanwendung in Microservices. Außerdem sollte die Anwendung durch neue Microservices mit eigenen Funktionalitäten erweitern werden. In vorherigen Projekten basierte die Restaurantanwendung auf OASP (siehe Kapitel 4), sodass OASP ebenfalls als Technologieplattform für das Projekt gewählt wurde. In diesem Zusammenhang soll auch ermittelt werden, in wie weit OASP für die Umsetzung von Microservices geeignet ist.

2. Vorgehensweise

Das Projekt wurde nach agilen Entwicklungsmethoden durchgeführt. So wurde Scrum als geeignete Methode angesehen, um zum einen organisiert arbeiten und zum anderen sehr flexibel auf Ereignisse reagieren zu können. Da davon ausgegangen wurde, dass der zunächst relativ unbekannte Ansatz einer Microservice Architektur nicht ohne Probleme für die bestehende Anwendung übernommen werden könne, wurde die agile Methode dem klassischen Wasserfallvorgehen vorgezogen. Die Arbeit wurde in vier Sprints aufgeteilt. Die ersten zwei Sprints hatten jeweils eine Länge von drei Wochen, die anderen beiden dauerten jeweils vier Wochen. Die Sprintdauer wurde im Verlauf des Projekts verändert, da durch die abgehaltenen Workshops einige Arbeitstage nicht vollständig genutzt werden konnten und die Resultate, die innerhalb des Sprints hätten erreicht werden können, sehr knapp ausgefallen wären.

3. Aufteilung in Teams

Im Verlaufe des Projektes hat das Team aus 12 Studenten verschiedene Untergruppen gebildet, um verschiedene Ziele zu verfolgen. Zu Beginn wurden vier Expertenrunden mit jeweils drei Personen gebildet. Diese Expertenrunden haben sich in die Themen Fachlicher Schnitt, Datenmodell und Kommunikation, User Interface (UI)-Integration und DevOps-Umsetzung eingearbeitet. Anschließend wurden für die weitere Arbeit drei neue Gruppen gebildet, die aus jeweils einem Experten der vorherigen Expertenrunden bestanden. Das erste Team hatte den Auftrag die Autorisierung und Authentifizierung in die Restaurantanwendung einzuführen beziehungsweise zu erweitern. Im Verlauf des Projektes wurde dieser Arbeitsauftrag erweitert, sodass Team 1 einen neuen Microservice für die Benutzerprofilverwaltung erstellen sollte. Das zweite Team beschäftigte sich hauptsächlich mit der gegebenen Restaurantanwendung und sollte diese in kleinere Microservices zerschneiden. Team drei wurde damit beauftragt einen neuen Microservice zu erstellen, der unter anderem die Bewertung von Rezepten und ein Empfehlungssystem umfassen sollte. Unterstützt wurden die Teams von jeweils einem Professor beziehungsweise wissenschaftlichen Mitarbeiter. Zudem bestand ein enger Kontakt zur Firma Capgemini, welche den Product Owner für unser Projekt stellte.

4. Workshops für teamübergreifende Themen

Während der Arbeit wurden zwei Workshops abgehalten, um sich teamübergreifend mit einem Thema auseinandersetzen zu können. Der erste Workshop fand in Sprint 3 statt. Er hatte das Ziel die Teams in OASP einzuführen und sie so für die Programmierung eigener Softwareelemente vorzubereiten. Der zweite Workshop wurde im vierten Sprint abgehalten. Hier wurde das Autorisierungs- und Authentifizierungssystem von Team 1 den anderen Teams vorgestellt und zusätzlich in die anderen Microservices eingebaut, sodass die gesamte Restaurantanwendung mit dem Autorisierungs- und Authentifizierungssystem arbeitet.

Microservices-Architektur

Die Microservices-Architektur ist ein moderner Ansatz Software aufzubauen und wurde zwischen 2005 und 2012 ins Leben gerufen. Der Aufstieg der Microservices hängt dabei eng mit der Art der Softwareentwicklung zusammen. In der Vergangenheit wurde Software viel nach Wasserfallmodellen entwickelt. Bei diesen Modellen wird ein Softwareprodukt erstellt und nach Abschluss der Entwicklungsarbeiten wird eine komplette Anwendung mit den gewünschten Features abgeliefert. Mit dem Aufkommen der agilen Entwicklungsmethoden um die Jahrtausendwende änderte sich die Idee der Softwareentwicklung. So wird zwar noch immer ein Produkt entwickelt, allerdings wird der Entwicklungsprozess in Etappen (Sprints) eingeteilt. Zu Beginn der Entwicklung gibt es eine Gesamtplanung, jeder Sprint wird jedoch einzeln geplant, wodurch gut auf Änderungen und den aktuellen Entwicklungsstand eingegangen werden kann. Ziel ist es, ein fertiges Produkt am Ende eines jeden Sprints abzuliefern und dieses mit den folgenden Sprints auszubauen, bis es vollständig ist. Microservices versprechen für dieses Vorgehen eine angemessene Architektur. Ein einzelner Microservice ist dabei eine kleine eigenständige Anwendung. Ein Microservice enthält daher nicht selten eine eigene Benutzerschnittstelle und andere Elemente zur Ausführung und Benutzung. Ein Micoservice kann relativ schnell, zum Beispiel während eines einzelnen Sprints entwickelt und in ein Gesamtsystem integriert werden. Dadurch kann ein Kunde nach jedem Sprint das eigene Produkt aktualisieren und unmittelbar mit den neuen Features arbeiten. Gleichzeitig hat eine Microservices-Architektur den Vorteil, dass einzelne Microservices schnell und einfach ausgetauscht, entfernt oder hinzugefügt werden können, ohne dass das gesamte System ausfällt, da alle Teile in der Microservices-Architektur alleine lauffähig sind. Diese Eigenschaft führte allerdings dazu, dass Microservices oftmals als Wegwerf-Software bezeichnet werden, da sie nur für einen bestimmten Zweck entworfen werden und bei Änderungen wieder weggeschmissen beziehungsweise neu erstellt werden. Ein sehr prominentes Beispiel für eine Microservices-Architektur ist die Website von Amazon, die auch für dieses Projekt oft als Vorbild genutzt wurde.
Neben der Ersetzbarkeit, der starken Modularisierung und die Unterstützung agiler Prozesse nennt [Wolff, 2015] weitere Stärken der Microservices-Architektur wie die Ergänzung von Legacy-Systemen, Technologiefreiheit und Continuous Delivery. Anstatt den unübersichtlichen und oft veralteten Quellcode eines monolithischen Systems zu

ergänzen, kann die neue Funktionalität in einem Microservice umgesetzt werden. Der Vorteil dabei ist, dass die Umsetzung mit Hilfe beliebiger Technologien erfolgen kann, sodass moderne Ansätze ohne das Risiko, dass das Gesamtsystem ausfallen kann, ausprobiert werden können. Außerdem erlaubt die Größe eines Microservices ein schnelleres und von anderen Teilen des Systems unabhängiges Deployment, was die Umsetzung einer Continuous-Delivery-Pipeline einfacher macht.

OASP

OASP ist die Technologieplattform, die für dieses Projekt gewählt wurde. OASP steht für *Open Application Standard Platform* und ist eine frei zugängliche Plattform, die zum erstellen von Software genutzt wird. Charakteristisch für OASP ist die Architektur einzelner Anwendungen, die mit OASP erstellt werden. Jede Komponente in OASP wird auf drei Ebenen aufgeteilt: Den Data-Access-Layer, den Logic-Layer und den Service-Layer. Der Service-Layer liegt zu oberst und ist die Schnittstelle zum Frontend beziehungsweise anderen externen Komponenten. Der Logic-Layer ist das Zentrum jeder Anwendung und enthält die Use Cases, die eigentliche Logik, der Anwendung. Der Data-Access-Layer ist in erster Linie für den Zugriff auf persistente Daten zuständig.

Teil II.

Hauptteil

1. Fachlicher Schnitt

Der erste Teilbereich der Evaluation beschäftigt sich mit Aufteilung der bestehenden Anwendung und der vorliegenden Anforderungen in fachlicher Komponenten, welche als eigenständige Microservices implementiert werden könnten. Dafür wird ein kurzer Überblick über den Stand der Anwendung zu Beginn des Projektes gegeben und Ansätze für die Aufteilung vorgestellt. Anschließend werden die Ergebnisse der vorgenommenen Aufteilung diskutiert und eine Wertung für diese vorgenommen.

1.1. Stand des Monolithen

Der Monolith, welcher mittels Microservices aufgeteilt werden sollte, ist die Beispielanwendung für OASP und wurde im Zuge vergangener Projektarbeiten erstellt und erweitert. Dazu wurde ein Client in AngularJS mit einer Java-basierten Serveranwendung in Form von OASP4J verwendet.
Als eine Restaurant-Beispielanwendung beinhaltet sie eine Reihe von Services mit unterschiedlichen Funktionalitäten. Als erster nicht angemeldeter Besucher wird die Landingpage aufgerufen, die als Loginseite dient und drei zufällig ausgewählten Rezepte als Beispielgerichte aufweist.
Nach einer Anmeldung können im Bereich für Tische (*Tables*) diese reserviert und Kellnern zugeordnet werden. Zu jedem Tisch können zudem Details wie die Bestellungen, die für die Gäste des jeweiligen Tisches zubereitet werden müssen, aufgegeben werden. Diese Bestellungen erscheinen automatisch im Bereich der Küche (*Kitchen*), wo sie sich ein Koch zuweisen kann. Bei der Bestellung wird aus den angelegten Gerichten gewählt, die im Rezeptbereich (*Recipes*) angelegt und mit den zur Zubereitung notwendigen Zutaten und weiteren Details gefüllt werden können. Die Details können getrennt sowohl auf Deutsch als auch auf Englisch vorliegen.

1.2. Ansätze zur Aufteilung

Der in Abschnitt 1.1 vorgestellte Monolith soll bearbeitet und fachlich in mehrere Microservices geteilt werden. Hierzu gibt es unterschiedliche Empfehlungen aus der Literatur.

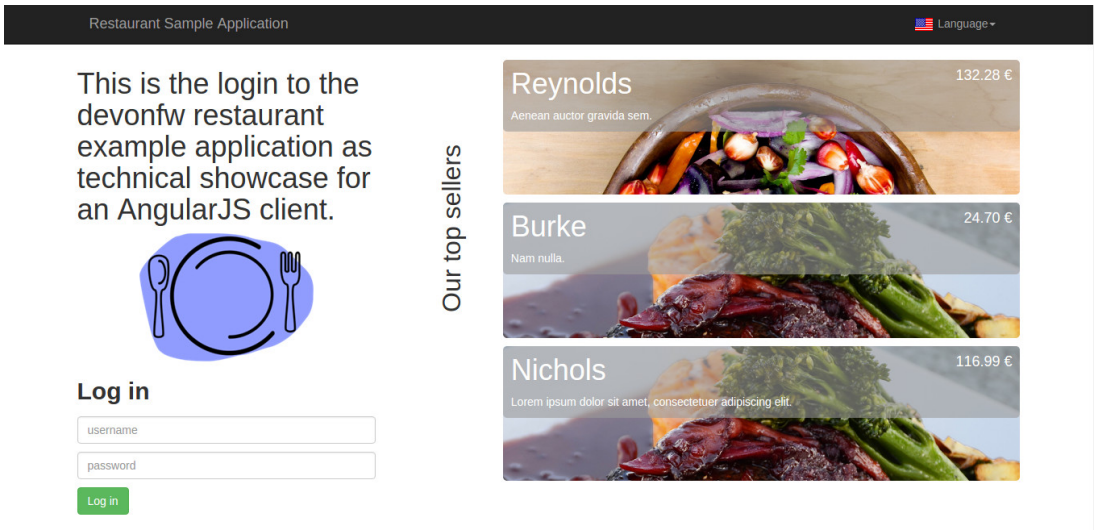

Abbildung 1.1.: Landingpage der OASP4J-Beispielanwendung

1.2.1. Microservices-Empfehlungen

Bei der Aufteilung der Services in Microservices gibt es eine große Auswahl unterschiedlicher Empfehlungen und Möglichkeiten. Ein wirkliches Patentrezept gibt es jedoch nicht.
Bei der Unterteilung der Services ist einiges zu beachten, was einen Microservice unter anderem ausmacht. Wie in Kapitel 4 bereits angesprochen, ist ein Microservice eine eigenständige Anwendung, der mit anderen Services kollaboriert und ihnen zuarbeitet. Er ist auf eine bestimmte Aufgabe spezialisiert und kann unabhängig von anderen Microservices geändert sowie bereitgestellt werden. Bei der Erstellung eines Microservices muss entschieden werden, welche Informationen gemeinsam genutzt, also anderen Services mitgeteilt werden sollen. Nur das unbedingt notwendige sollte den jeweils anderen Services bekannt sein. Wie der Name schon sagt, sollte ein Microservice möglichst klein sein. Die Definition wie klein und woran dies gemessen werden kann, ist jedoch wieder nicht fest vorgegeben.[Vgl. Newman, 2015]

Microservice und UI

Häufig wird empfohlen, einen Microservice mit einer User Interface (UI), also einer Benutzerschnittstelle, auszustatten. Dies ist allerdings bei einer technischen Aufteilung nicht zwingend notwendig, sollte ein Microservice nur Logik beinhalten, die von mehreren Frontends genutzt wird. Bei einer fachlichen Aufteilung dagegen wird meist

empfohlen, die User Interface (UI) mit in den Microservice aufzunehmen und nicht auszulagern.[Vgl. Wolff, 2015]

Anwendungsfälle

Ein häufig gewählter Ansatz ist die Aufteilung der Microservices in Kontexte beziehungsweise Anwendungsfälle. So hat jeder Microservice eine bestimmte thematische Zuständigkeit und zusammenhängende Funktionalitäten lassen sich so ergänzen und wieder entfernen.
Dies ist auch Bestandteil des sogenannten *Domain-Driven-Designs (DDD)s.* Jede Funktionalität kann ein eigener Microservice sein. Entsprechend soll jeder Microservice eine eigene fachliche Einheit bilden, durch die bei Änderungen oder neuen Funktionen nur dieser Microservice geändert werden muss. Ein solcher Microservice soll eine eigene fachliche Domäne modellieren, die auf Basis von *Bounded Context* entworfen werden kann. *Bounded Context*, besagt, dass ein Domänenmodell nur in bestimmten Grenzen innerhalb eines Systems sinnvoll ist. So können Teams unabhängig voneinander an ihren Microservices arbeiten ohne sich das Datenmodell teilen zu müssen. Ein Microservice sollte jedoch nicht mehrere *Bounded Contexts* mit entsprechend mehreren Features enthalten, da dieser dann meist in einen weiteren Microservice aufgeteilt werden könnte.[Vgl. Wolff, 2015]

Unterschiedliche Technologien

Ein großer Vorteil bei Microservices ist die Möglichkeit, unterschiedliche Technologien zu verwenden. So ist eine mögliche Aufteilung, dass bestimmte Teile einer Anwendung, die beschleunigt werden sollen, als Microservice mit einer passenderen Technologie ausgelagert werden. Beispielsweise bei Interaktionen von Benutzern in einem sozialen Netzwerk sind, anders als in der restlichen Anwendung, Graphdatenbanken wie Neo4J sinnvoll, da sie die verknüpften Verbindungen in Form eines Social Graphs speichern können. So könnte ein eigener Microservice für die Interaktionen mit einer solchen Datenbank erstellt werden.
Die Auswahl der unterschiedlichen Technologien sollte auf Grund der Kompatibilität und einem damit einhergehenden Mehraufwand gegebenenfalls eingeschränkt werden.[Vgl. Newman, 2015]

Programmbibliotheken

Eine weitere Möglichkeit zur Aufteilung sind Programmbibliotheken. Mit Hilfe von solchen werden Funktionalitäten ausgelagert. Die Voraussetzungen dabei ist, dass sie in derselben Sprache programmiert sind, oder zumindest auf derselben Plattform laufen

müssen. Bei den Bibliotheken müsste zudem ein Monolith oder zumindest ein Programmrahmen existieren, der diese einbindet.[Vgl. Newman, 2015]

Logging

Eine Möglichkeit, das Gesamtsystem zu überwachen muss auch mit den unterschiedlichen Microservices gewährleistet sein. So ist es empfehlenswert, einen eigenen Microservice mit passenden Schnittstellen zu erstellen, der von den anderen Services verwendet wird.

Übergreifende Praktiken und Prinzipien

Bei einer Verwendung mehrerer Microservices können Leitlinien sowie Praktiken und Prinzipien hilfreich sein, die dabei helfen, dass die jeweiligen Services kompatibel zueinander sind und gut miteinander funktionieren. Diese übergreifenden Praktiken und Prinzipien gelten dann für das gesamte System mit allen Microservices.
Bei der Kommunikation der Services zum Beispiel ist es hilfreich, sich im Vorfeld auf bestimmte Möglichkeiten zu beschränken. Ansonsten könnte es bei vielen verschiedenen Kommunikationsformen kompliziert werden, da unterschiedliche Schnittstellen vorhanden sein müssten. Weiterhin können Mindestvorgaben festgelegt werden, die besagen, welche Fähigkeiten ein Microservice haben muss. Viele kleine Teile mit eigenständigen Entwicklungszyklen stellen sicher, dass ein Service, der ausfällt, nicht das gesamte System lahmlegt. Codebeispiele für die Mindestvorgaben können zudem helfen.[Newman, 2015]

1.3. Ergebnisse

Es gibt zwischen der Planung, als Ergebnis des Kick-Off-Meetings und der Expertenrunde, bis zur tatsächlichen Umsetzung zum Teil erhebliche Differenzen beim fachlichen Schhnitt. Dies ist dem Umstand geschuldet, dass viele Teilnehmer noch nicht mit dem Framework OASP gearbeitet hatten und insbesondere nur über theoretisches Wissen über Microservices verfügten. Aus diesem Grund ist das Ergebnis in drei Unterkapitel unterteilt. Das Erste beschäftigt sich mit der Erarbeitung und der grundsätzlichen Konzeption und Vorgehensweise beim fachlichen Schnitt - der Planung. Das zweite Kapitel mit dem, wie es am Ende realisiert wurde und das Letzte damit, wie hoch die Abweichung war.

1.3.1. Ergebnisse der Planungsphase

Noch vor dem Kick-Off Meeting wurde eine Expertengruppe unter anderem für den fachlichen Schnitt gegründet. Diese sollte zum Kick-Off Meeting bei Capgemini in Troisdorf unter anderem das theoretische Hintergrundwissen für den fachlichen Schnitt vermitteln und Vorschläge einreichen, was Bestandteil des Monolithen bleibt, was als möglicher Microservice in Betracht kommt und welche zusätzlichen Microservice entwickelt werden sollten.
Vorgeschlagene Microservices:

- Zutaten
- Rezepte (inkl. Suche und Filter)
- Bewertungen

Vorgeschlagene Bestandteile des Monolithen:

- Tische
- Küche
- Logging
- Benutzerverwaltung

Damit der fachliche Schnitt für die einzelnen Teams verständlicher ist, wurden Umsetzungsstrategien und Orientierungshilfen, auf Grundlage von mehreren Fachliteraturen und Internetquellen, erstellt:

Partitionierungsstrategien:

- Services nach Verben oder Anwendungsfällen partitionieren
- Services nach Nomen oder Systemen partitionieren
- Services nach Unix Werkzeugen designen
- Kleine Sammlung von Verantwortungen

Orientierungshilfen:

- CRUD ist zu klein (enthält keine Geschäftslogik)
- Kein Big-Bang

- Nicht in *Lines of Code* bemessen

Darüber hinaus wurde ein strategischer Leitfaden zur weiteren Entwicklung entlang des fachlichen Schnittes entwickelt:

Schritt 1: Keine Weiterentwicklung des Monolithen
Neue Module sollen direkt als Microservice entwickelt werden. Diese sollen dann mittels eines API-Gateways oder eines Request Routers mit dem Monolithen kommunizieren. Hierbei gibt wurden drei mögliche Szenarien angeboten:

- Einen direkten Aufruf der API des Monolithen
- Einen direkten Datenbankzugriff auf dem Monolithen
- Eine automatische Spiegelung und Synchronisation der Daten

Schritt 2: Teilung von Front- und Backend
Hier wird die Strategie verfolgt, die Präsentationsschicht von der Geschäftslogik und den Daten zu trennen. Die Präsentationsschicht soll dabei die User Interface (UI)-Komponenten über HTTP-Requests dynamisch erstellen. Nachdem das Frontend der Komponente vom Monolithen getrennt ist, sollen nun noch die Schichten der Geschäftslogik und des Datenzugriffs voneinander getrennt werden. Diese Strategie gilt für den Monolithen, neue Microservices können entweder monolithisch oder direkt aufgeteilt entwickelt werden.
Der erhoffte Vorteil ist hierbei ein vereinfachtes A/B testen für die Frontend-Entwickler.

Schritt 3: Service Extrahieren
Der Monolith enthält meist bereits einige Module, die häufig mit dem Bounded Context übereinstimmen. Um in dem zeitlich sehr begrenzten Rahmen die wertvollsten herauszulösen, wurden folgende mögliche Kriterien zur Bewertung der Prioritäten entworfen:

- Mit einfachen Services anfangen
- Services nach Vorteilen einstufen
- Suchen nach existierenden grobkörnigen Schnittstellen

Die genannten Ergebnisse der Expertengruppe zum fachlichen Schnitt wurden in das Kick-Off Meeting bei Capgemini genommen. Als Resultat eines demokratischen Entscheidungsprozesses wurden die Microservices „Rezeptemanagement“, „Bewertungen und Kommentare“, „Authentifizierung und Autorisierung“, „Benutzermanagement“ und „Zutaten- und Lieferantenmanagement“ bestimmt, um sie zu entwickeln.

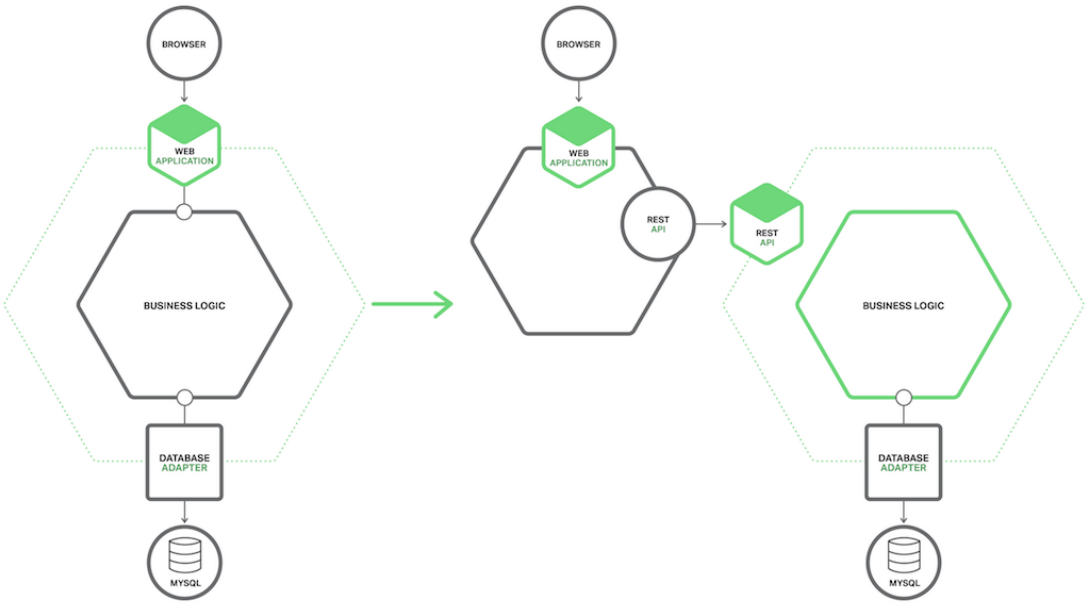

Abbildung 1.2.: Fachlicher Schnitt nach [Richardson, 2016]

1.3.2. Ergebnisse zum Abschluss der Entwicklungsphase

Es gab zwischen der Planung und dem vermittelten Wissen bis zur Realisierung der Microservices einige Differenzen. Hinsichtlich der Expertenmeinung wurden alle Microservices und Monolithmodule übernommen wie geplant, mit Ausnahme der Benutzerverwaltung. Der Microservice „Zutaten- und Lieferantenmanagement“ wurde nicht realisiert, da es sich schematisch um einen neuen Microservice handelt, wie es auch der Microservice „Bewertung und Kommentare“ war. Nach längerer Diskussion wurde entschieden, dass die Entwicklung viel Zeit in Anspruch nehmen würde, ohne dass dabei neue Erkenntnisse gewonnen werden könnten.

Als Partitionierungsstrategie wurden Services nach Anwendungsfällen in grob- (Authentifizierung und Autorisierung, Rezeptemanagement, Benutzerverwaltung) und mittelgranularer Größe (Bewertung und Kommentare) gewählt. Hinsichtlich der Orientierungshilfe kann man am ehesten bei A&A von einem BigBang sprechen, da es alle anderen MS ebenfalls betraf, ansonsten wurden die Orientierungshilfen gut berücksichtigt. Der Leitfaden zur Entwicklung entlang des fachlichen Schnittes wurde ebenfalls berücksichtigt. So wurde der Monolith nicht weiterentwickelt. Wenngleich die Teilung des Front- und Backends nicht ohne Probleme realisiert wurde. Der letzte Schritt, die

Services zu extrahieren, wurde aufgrund des Vorgängerprojektes und des demokratischen Entscheidungsprozesses stark beeinflusst. Der Lerneffekt und die Evaluation standen hier im Vordergrund.

1.4. Wertung

Um einen Microservice hinsichtlich seiner Qualität des fachlichen Schnittes beurteilen zu können wurden folgende Kriterien festgelegt:

- „Separation of concerns“, „High cohesion, low coupling“, „Bounded Context“
  Beim domain-driven Design ist häufig die Rede vom Bounded Context. In anderen Zusammenhängen die „Seperation of Concerns“ und in wieder anderen von „High cohesion, low coupling“ die Rede. Ihre Bedeutung ist ähnlich bis identisch. Entscheidend ist das ein Modul eine möglichst geringe Abhängigkeit zu anderer Software, bei uns dem Monolithen und den MS, hat. Darüber hinaus muss der Microservice eine möglichst hohe Kohäsion, also verantwortlich für eine wohldefinierte Aufgabe sein und selbst eine logische Einheit bilden.

- Größe des Microservice
  Hier sind sich die Experten uneinig wie klein ein Microservice sein soll. Die Maximalgröße hingegen ist meist unumstritten, jeder Einzelne im Entwicklungsteam des Microservices sollte diesen vollständig erfassen und verstehen können.

- unabhängig deploybar
  Innerhalb des Application Lifecycles sollten Microservices ausgerollt werden können, ohne das dabei andere Microservices oder der Monolith ebenfalls neu deployt werden müssen.

- Ersetzbar
  Ein Microservice sollte immer durch einen neuen Microservice schnell und unkompliziert ersetzt werden können. Dies soll das A/B Testen erleichtern.

Die Wertung, im Schulnotensystem, der einzelnen Microservices sieht dabei wie folgt aus:

Kriterien & Microservice	Domäne	Größe	Deployment	Ersetzbarkeit
A&A	befriedigend	gut	sehr gut	befriedigend
Benutzerverwaltung	befriedigend	gut	sehr gut	befriedigend
Bewertung & Kommentare	sehr gut	sehr gut	sehr gut	sehr gut
Rezeptemanagement	sehr gut	sehr gut	sehr gut	sehr gut

Tabelle 1.1.: Bewertung in Schulnoten

Die Domänen und Ersetzbarkeit der Microservice für Benutzerprofile und A&A sind nicht leicht zu realisieren. Von der Größe sind sie mit Abstand komplexer als alle anderen MS.

2. Authentifizierung und Autorisierung

Authentifizierung (engl. *authentication*) und Autorisierung (engl. *authorisation*) sind wesentliche Konzepte, die die Sicherheit eines Systems unterstützen, in dem interne oder externe Zugriffe auf Ressourcen entsprechend behandelt werden können. Authentifizierung kümmert sich um die Benutzeridentität. Mittels verschiedener Verfahren, wie zum Beispiel einem Anmeldeformular, kann die Identität des Benutzers[1] überprüft werden. Dabei werden die Anmeldedaten des Benutzers (engl. *credentials*) an das System übermittelt und ihre Authentizität geprüft. Nachdem die Identität des Benutzers erfolgreich bestätigt wurde, erhält das System einen Zugang auf die Benutzerdaten, die ihm helfen, den Benutzer zu autorisieren. Das heißt feststellen zu können, welche Zugriffsrechte auf welche Ressourcen ihm erlaubt sind.
In der Microservices-Architektur wird in der Regel ein zentraler Server für die Authentifizierung benutzt. In manchen Fällen macht auch eine zentrale Verwaltung der Benutzergruppen Sinn. Jedoch sollte jeder Microservice selber entscheiden können, welche Funktionalitäten welchen Benutzern zugänglich sind [Vgl. Wolff, 2015, S.154]. Mehr zu den Umsetzungsmöglichkeiten und verwendeten Technologien in den nächsten Abschnitten.

2.1. A&A in Microservices-Architekturen

Authentifizierung und Autorisierung (kurz A&A) auf dem neuesten technischen Stand zu halten und effektiv umzusetzen ist bereits in herkömmlichen, monolithischen Systemen eine herausfordernde und niemals endende Aufgabe. In Microservices-Architekturen wird diese Aufgabe aufgrund der größtenteils autark agierenden Microservices noch komplexer. Welche spezifischen Herausforderungen in diesem Umfeld existieren und wie ihnen begegnet werden kann, soll in diesem Abschnitt beschrieben werden.

2.1.1. Herausforderungen

Ein monolithisches System in Microservices zu zerlegen, führt zu einer erhöhten technischen Komplexität, welche neue Herausforderungen entstehen lässt. Vor allem für den

[1] Ein Benutzer kann nicht nur ein Mensch, sondern auch ein anderes System, also eine Maschine sein.

Bereich Sicherheit müssen viele Aspekte beachtet werden. In der Regel reicht die bestehende Sicherheitsarchitektur für die adäquate Absicherung von verteilten Systemen nicht mehr aus. Es müssen viele grundlegende Entscheidungen getroffen und ungeklärte Fragen beantwortet werden, wie beispielsweise ob man eine zentrale oder dezentrale Lösung implementieren möchte. Entscheidet man sich für die zentrale Variante, wird man mit Fragen wie „Wie können Rollen und Gruppen zentral verwaltet werden?“ oder „Was passiert bei einem Ausfall der Zentrale?“ konfrontiert. Um zu verhindern, dass das gesamte System ausfällt, müssen die anderen Microservices in der Lage sein, den Ausfall zu kompensieren.
Nicht zu vernachlässigen ist die erschwerte Domination der Infrastruktur. Da Microservices verteilte Systeme sind, kommunizieren sie asynchron über das Netzwerk. Die ausgewählte Lösung sollte in der Lage sein, diese Kommunikation konsistent abzusichern. Wie in Kapitel 4 bereits genannt, enthält ein Microservice neben der eigentlichen Anwendungslogik auch die Datenhaltungs- und Präsentationsschicht, die ebenfalls von dem Sicherheitskonzept berücksichtigt werden müssen. Darüber hinaus kann jeder Microservice seine verwendeten Technologien frei wählen. Diese unterschiedliche technologische Basis erhöht zusätzlich die Komplexität des gesamten Systems. Insbesondere für den Bereich Sicherheit ist jedoch ein grundlegendes Verständnis des Gesamtsystems notwendig. Ohne ein Mindestmaß an Standardisierung ist eine übergreifende Sicherheitsarchitektur kaum vorstellbar. Als praktisches Beispiel sei hier das zentrale Logging genannt, mit dessen Hilfe eine Mícroservices-übergreifende Analysierbarkeit des Gesamtsystems sichergestellt werden soll.
Ein großer Vorteil einer Microservice-Architektur ist das unabhängige Deployment der einzelnen Services. Um diesen Vorteil nicht zu entkräften, müssen weitere Überlegungen erfolgen. Insbesondere die Authentifizierung von Benutzern ist in der Regel eine zentrale Funktion, die von allen Microservices genutzt wird. Sollte nun eine Bibliothek für die Interaktion mit dem Authentifizierungsdienst notwendig sein, entsteht eine ungewollte Abhängigkeit zwischen der Authentifizierung und den einzelnen Services, was bei Änderungen des Authentifizierungsdienstes letztendlich zu einem abhängigen und aufwendigen Deployment führt. Das Deployment muss dann entsprechend koordiniert werden, sodass keine Probleme im Betrieb auftreten.
Aus organisatorischer Sicht gibt es ebenfalls Herausforderungen. Die von der Microservice-Philosophie propagierte unabhängige Entwicklung der Services wird durch ein Microservices-übergreifendes Sicherheitskonzept beeinträchtigt. Es wird mehr Koordination und Kommunikation zwischen den Teams notwendig, was einen negativen Einfluss auf die Produktivität und die Agilität der Entwicklung zur Folge haben kann.

2.1.2. Best Practices

Zur Umsetzung von Single-Sign-On (SSO) in Microservices-Architekturen werden in den meisten Fällen etablierte Implementierungen wie SAML oder OpenID Connect verwendet. Dies ist insbesondere zu empfehlen, da sicherheitskritische Implementierungen nur von absoluten Experten auf diesem Gebiet vorgenommen werden sollten, da es sonst schnell zu Schwachstellen kommen kann. Hauptunterschied zwischen SAML und OpenID ist die Nutzung von SOAP durch SAML während OpenID Representational State Transfer (REST) nutzt. Daneben wird OpenID oft als einfacher empfunden, da es weniger komplex ist.
Unabhängig davon ob SAML oder OpenID genutzt wird, wird ein Identity Provider (IP) benötigt. Dieser führt die eigentliche Authentifizierung durch, zum Beispiel durch eine Verifizierung eines Benutzers durch ein Passwort. Hier wird grundsätzlich zwischen öffentlichen IPs, zum Beispiel Google oder Facebook, oder aber eigenen IPs unterschieden. Während ein öffentlicher IP vergleichsweise einfach über eine API eingebunden werden kann und für Nutzer einer öffentlich zugänglichen Seite gegebenenfalls einladender wirkt, sollten für interne Systeme eigene IPs in Betracht gezogen werden. Diese führen zu mehr Kontrolle über die Authentifizierungsdaten sowie der Umsetzung und Infrastruktur der Authentifizierung. Bekannte Implementierungen zur Umsetzung eigener IPs sind OpenAM, JOSSO und Crowd.
Neben einem Identity Provider wird auch ein Service Provider benötigt. Dieser übernimmt die Autorisierung der Benutzer indem diesen zum Beispiel Rollen und Gruppen zugewiesen werden, die festlegen welche Aktionen diese ausführen dürfen. Oftmals sind Service Provider mittels eines Directory Service umgesetzt. Aktuelle Implementierungen sind LDAP (Leightweight Directory Access Protocol), Active Directory oder OpenIDM.
Im speziellen Anwendungsfall der Microservices existieren zwei grundsätzlich verschiedene Ansätze zu deren Umsetzung. Beim ersten Ansatz wird den einzelnen Microservices die Authentifizierung und Autorisierung vollständig überlassen, sodass jeder Microservice selbst dafür zuständig ist. Zur praktischen Umsetzung kann hierfür beispielsweise eine öffentliche Bibliothek genutzt werden. Zu den offensichtlichen Nachteilen dieses Konzepts gehört, dass eine entsprechende Bibliothek für jede genutzte Technologie neu geschrieben werden muss und durch die Einbindung in jeden Service eine erhebliche Mehrarbeit entsteht. Der zweite Ansatz beinhaltet den Einsatz eines Gateways, das vor alle Services geschaltet wird und alle Authentifizierungs- und Autorisierungsaufgaben übernimmt. Der Vorteil dieses Konzepts ist, dass jegliche Authentifizierung und Autorisierung lediglich einmal implementiert werden muss. Ebenso wird die Anzahl der Anfragen an Identity und Service Provider erheblich reduziert, da

die Authentifizierung und Autorisierung pro Nutzer nur noch einmal stattfindet und nicht für jeden Service einzeln. Der große Nachteil dieser Methode ist jedoch, dass ein Single Point of Failure eingeführt wird. Sollte dieser ausfallen, so beeinträchtigt dieser das komplette System, hat also einen Effekt der gerade bei Microservices-Architekturen nicht gewünscht ist. Zusätzlich verleitet eine Gateway-Lösung dazu, dass zwischen den Sevices und innerhalb des Microservices-Verbunds nur noch wenig oder gar nicht mehr authentifiziert und autorisiert wird, was bei einem Angriff von innerhalb des Verbunds fatale Auswirkungen auf die Sicherheit haben kann.
Zur technischen Umsetzung von Authentifizierung und Autorisierung existieren verschiedenste Möglichkeiten, die ihre jeweiligen Schwächen und Stärken haben. Zu den verbreitetsten zählen die Nutzung von Cookies, HTTPS Headern, Client-Zertifikaten und API-Schlüsseln. Um den Umfang dieser Arbeit auf ein vernünftiges Maß zu begrenzen wird auf die einzelnen Technologien an dieser Stelle nicht weiter eingegangen, sondern diese nur aus Gründen der Vollständigkeit genannt. In den nachfolgenden Abschnitten wird bei Bedarf deren konkrete Funktionsweise erläutert.

2.2. A&A in OASP

Da eines der Ziele des Projektes war, OASP hinsichtlich seiner Eignung zur Umsetzung von Microservices zu untersuchen und Authentifizierung und Autorisierung ein wichtiger Aspekt für Microservices-Architekturen sind, werden in diesem Abschnitt zunächst die Sicherheitsmaßnahmen von OASP vorgestellt. Nachfolgend wird das Projekt „Enterprise-Security“ analysiert und auf seine Verwendbarkeit hin geprüft. Abschließend werden die Schritte die zur Umsetzung von Authentifizierung und Autorisierung unternommen wurden beschrieben.

2.2.1. OASP Security

OASP wird mit mehreren fertigen Modulen geliefert, die die Entwicklung eines Systems vereinfachen sollen. Eines davon ist das Security Modul. Für den Authentifizierungsprozess wird hier als grundlegende Technologie Spring Security in der Version 3.2.5 benutzt. Die Autorisierung basiert auf einer, vom OASP-Team selbst erarbeiteten, Lösung, die aus zwei Komponenten besteht: *Identity- and Access-Management* und *Application-Security* (siehe Abbildung 2.1). Identity- und Access-Management wird durch einen Directory Server umgesetzt, der von außen zugänglich ist und an dem ein Administrator Benutzer, Gruppen und ihre zugehörigen Daten verwalten kann. Hier empfiehlt OASP eine auf dem Markt etablierte Lösung zu verwenden, die auf Standards wie zum Beispiel LDAP basiert. Application Security stellt einen sekundären Mechanismus für

die Zugriffskontrolle bereit. Er erlaubt eine interne Vergabe von Rechten an die im Directory Server definierten Gruppen. Dies passiert indem die bestehenden Gruppen in neue organisiert werden und für diese bestimmte Zugriffstrechte definiert werden. Diese Informationen werden in einer Datei (access-control-schema.xml) in der Anwendung aufbewahrt. In jedem Use Case der Anwendung (wie zum Beispiel „Find Table“ oder „Delete Order“) kann mittels JSR250 Annotationen definiert werden, welche Gruppe beziehungsweise Rolle den Use Case verwenden darf.

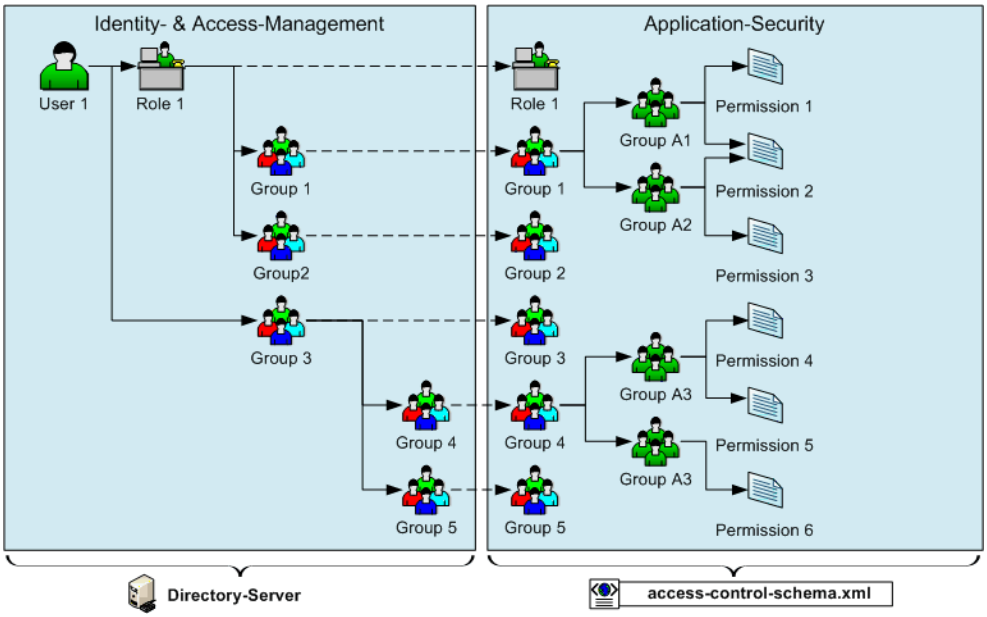

Abbildung 2.1.: OASP Security Model, entnommen aus https://github.com/oasp/oasp4j/wiki/guide-access-control.

2.2.2. OASP Enterprise Security

Neben den bereits standardmäßig in OASP vorhandenen Sicherheitsmaßnahmen existiert das Projekt „Enterprise-Security“ als Branch von OASP. Ziel des Projekts ist die Bereitstellung von Authentifizierungs- und Autorisierungstechnologien für Microservices-Architekturen, die mit OASP realisiert werden. Aufgrund einer Empfehlung seitens Capgemini wurde das Enterprise-Security-Modul auf seine Verwendbarkeit hin überprüft.

Das Enterprise-Security-Modul erfüllt laut Dokumentation alle Anforderungen, die an die Authentifizierung und Autorisierung gestellt wurden: Anstatt dass jeder Microservice selbständig für die Authentifizierung und Autorisierung der Nutzer zuständig ist, sollte ein Portal-Service alle sicherheitsbezogenen Aufgaben übernehmen. Dazu sollte der Portal-Service alle Anfragen mit Hilfe eines Single-Sign-On-Tokens (SSO-Token)

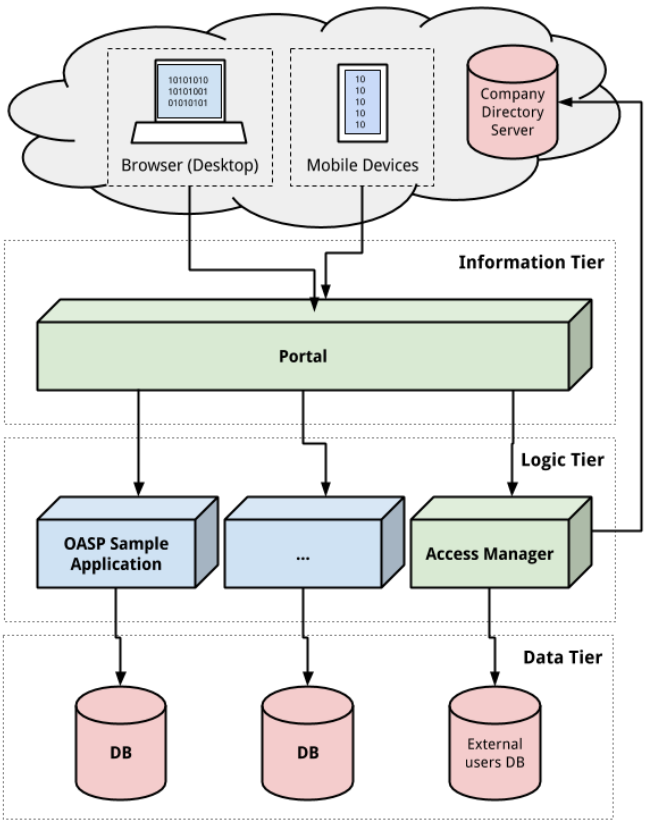

Abbildung 2.2.: Enterprise Security Architektur, entnommen aus https://github.com/oasp-forge/oasp4j-enterprise-security/wiki

validieren und bei Erfolg an den zuständigen Microservice weiterleiten. Die Realisierung des SSo-Tokens, sowie die Authentifizierung sollte mittels OpenAM und verschiedenen Directory-Diensten erfolgen. Die Autorisierung sollte von jedem Microservice selbständig erfolgen, indem dieser ein JSON-Web-Token(JWT) vom Portal-Service erhält, mit dem der Nutzer eindeutig identifiziert werden kann.
Ob die skizzierte Funktionalität tatsächlich vorhanden ist, konnte jedoch nicht belegt werden. Aufgrund der Komplexität des Enterprise-Sercurity-Projektes und der fehlenden Dokumentation konnte das Portal-Modul nicht in das existierende OASP-Projekt integriert werden. Da auch die Arbeiten am Entsprise-Security-Modul bereits seit über einem halben Jahr nicht weitergeführt wurden, wurde davon ausgegangen, dass auch keine weitere Hilfe oder Aktualisierungen zu erwarten seien. Somit wurde das Enterprise-Security-Modul nicht direkt für die Entwicklung der Authentifizierungs- und Autorisierungslösung genutzt. Lediglich die Authentifizierungslösung OpenAM wurde für die spätere Implementierung weiterhin verwendet.

Access Management

Das Projekt „Enterprise-Security“ definiert einen sogenannten Access Manager (AM). Der AM ist die zentrale Komponente innerhalb des SSO-Systems. Er generiert und validiert die SSO-Token. Vor der Generierung eines neuen Tokens, muss sich der Benutzer beim AM authentifizieren. Dieser ist an verschiedenen Datenquellen (zum Beispiel Verzeichnisdienste) gebunden, welche die, für die Autorisierung notwendigen, Benutzerdaten enthalten. In der Dokumentation des Enterprise-Security-Moduls werden zwei Implementierungen eines AM (OpenAM & JOSSO) evaluiert und miteinander verglichen. Im Rahmen dieser Evaluation wurde dieser Vergleich um eine weitere Variante erweitert. Die folgende Tabelle stellt die verschiedenen Produkte gegenüber:

Produkt	**Hersteller**	**Lizenz**
OpenAM	ForgeRock	Common Development and Distribution License (CDDL)
JOSSO	Atricore Inc.	GNU Lesser General Public License (LGPL)
Crowd	Atlassian Corporation plc	proprietär

Tabelle 2.1.: Verschiedene Implementierungen eines Access Managers

Das Projekt „Enterprise-Security“ empfiehlt OpenAM als eine umfangreiche AM-Implementierung. Aus diesem Grund wurde in einem weiteren Vergleich nur noch OpenAM und Crowd berücksichtigt. Nach einer ersten Recherche hat sich ergeben, dass OpenAM die mächtigere Lösung ist. Es bietet mehr Funktionalitäten und Integrationsmöglichkeiten als Crowd. Obwohl auch Crowd eine ausreichende Lösung für die meisten Anwendungsfälle bietet, ist es aufgrund seines proprietären Lizenzmodells weniger attraktiv als sein quelloffener Kontrahent.

SSO-Agent

Für die Integration der vorhandenen Microservices in die Sicherheitsarchitektur werden sogenannte SSO-Agents benötigt. Diese kommunizieren mit dem AM und setzen die von ihm vorgeschriebenen Regeln (Policies) um. OpenAM stellt bereits spezifische SSO-Agents für Web- und Applicationserver zur Verfügung. Die Policies für die Agents werden zentral von OpenAM verwaltet. Möchte man nicht unterstützte Anwendungen integrieren, kann man SSO entweder über einen Reverse Proxy realisieren, welcher die

Authentifizierung für die Benutzer übernimmt oder einen eigenen SSO-Agent implementieren. Für den letzteren Fall werden die Services von OpenAM über Java, C und Representational State Transfer (REST) APIs zur Verfügung gestellt. [Vgl. ForgeRock, 2016]
Es existieren zwei mögliche Architekturen für den Einsatz der SSO-Agents. Die erste Variante benötigt nur einen SSO-Agent, welcher zentral auf einem Reverse Proxy-Server installiert ist. Dieser ist für die Absicherung aller Microservices verantwortlich. Dieses Konzept wird auch vom Enterprise-Security-Modul mithilfe des Portals umgesetzt. Einer der Vorteile dieser Architektur ist der geringere Aufwand für die Installation und Konfiguration der benötigten Komponenten. Darüber hinaus findet die Netzwerkkommunikation mit dem AM ausschließlich über den zentralen Reverse Proxy-Server statt. Es existiert also nur eine Kommunikationsbeziehung mit dem AM, was das Gesamtsystem in Folge übersichtlicher macht. Dies führt wiederum zu einer höheren Sicherheit, da beispielsweise die Konfiguration der eingesetzten Firewalls vereinfacht wird. Die folgende Abbildung soll das beschriebene Konzept veranschaulichen. [Vgl. Oracle, 2010]

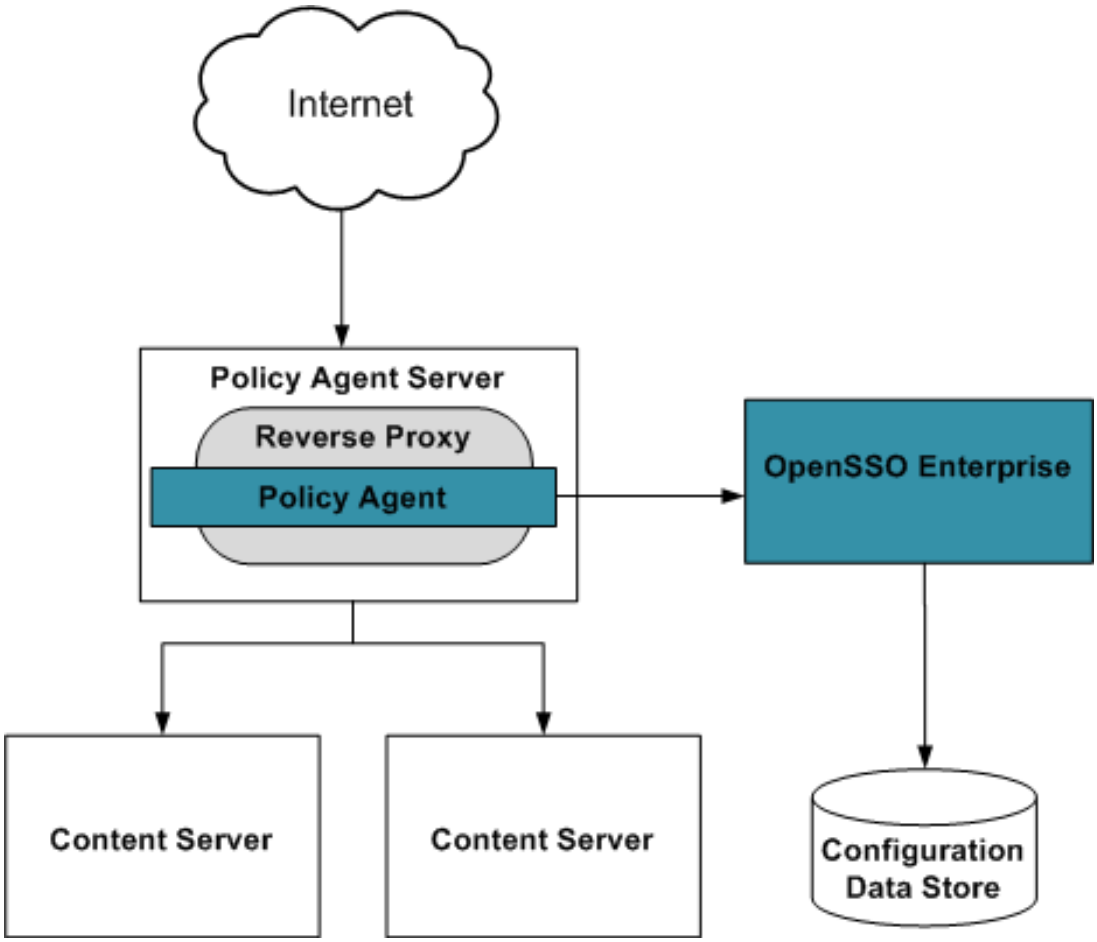

Abbildung 2.3.: Single Policy Agent, entnommen aus [Oracle, 2010]

Bei der zweiten möglichen SSO-Architektur wird für jeden Microservice ein dedizierter SSO-Agent installiert. Diese Variante erhöht zwar die Komplexität der Installation und Konfiguration der benötigten Komponenten, bietet jedoch die größere Flexibilität innerhalb des Gesamtsystems. So kann jeder SSO-Agent individuell auf den zu sichernden Microservice angepasst werden. Mögliche Szenarien sind zum Beispiel die Implementierung eines individuellen Loggings, Anpassungen der HTTP-Header und der Bereitstellung eigener URLs für Fehlerseiten und die Abmeldung eines Benutzers. Der letzte Punkt ermöglicht es spezifische Prozesse zu integrieren, die den Umgang mit der Session beziehungsweise dem Session Cookie definieren. Ein weiterer Vorteil der verteilten Kommunikation gegenüber einer einzigen Kommunikationsbeziehung mit dem AM ist die Reduzierung der Folgen eines Session Hijacking-Angriffs, bei dem die bestehen-

de Kommunikationsbeziehung unrechtmäßig verwendet wird. Die folgende Abbildung stellt die vorgestellte SSO-Architektur dar. [Vgl. Oracle, 2010]

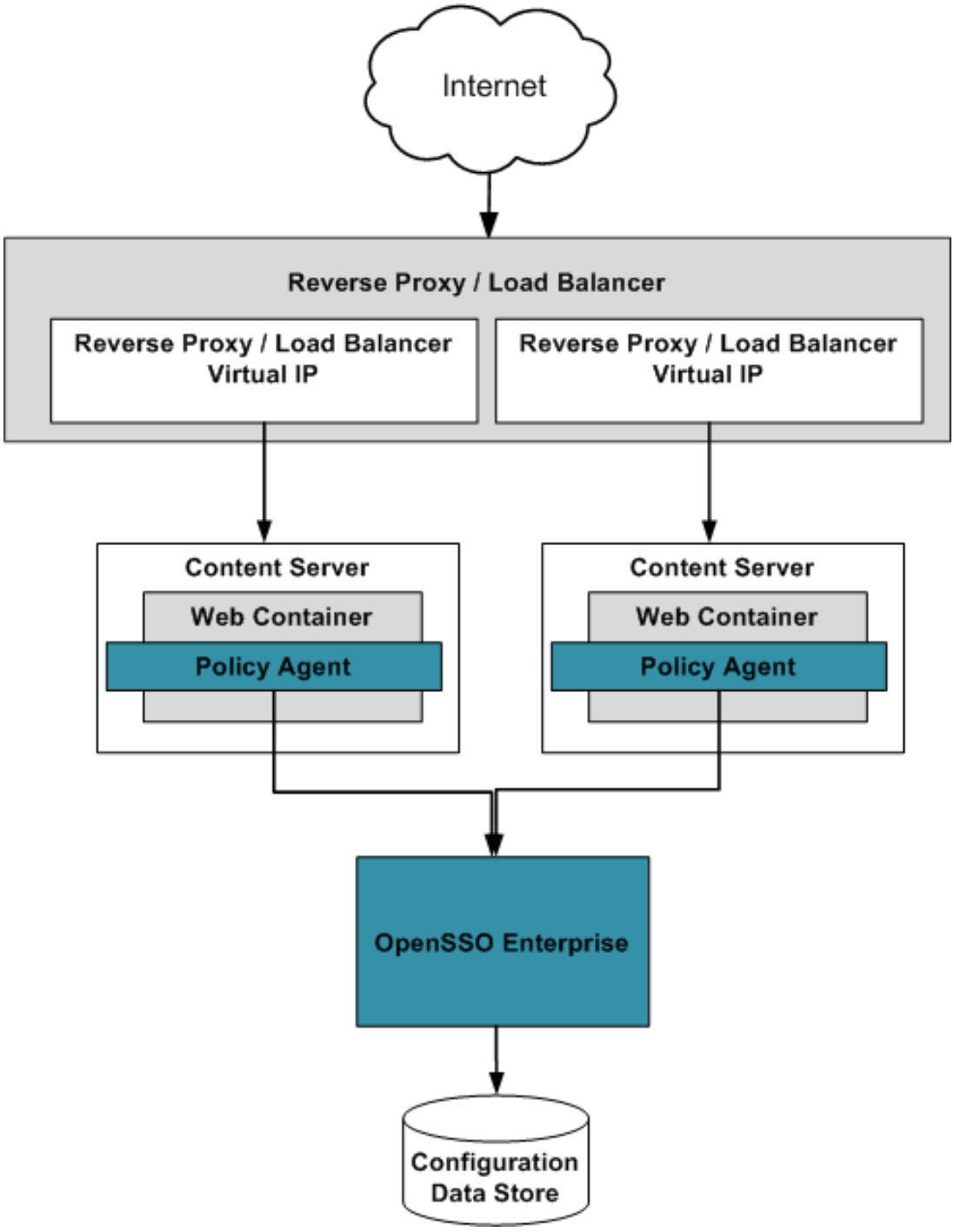

Abbildung 2.4.: Multiple Policy Agents, entnommen aus [Oracle, 2010]

2.2.3. Praktische Umsetzung

Das Projekt „Enterprise Security“ bietet ein umfangreiches Sicherheitskonzept, welches auch in Form eines nicht mehr weiterentwickelten Softwareprojektes heruntergeladen werden kann. Da es uns nicht möglich war die Software zu evaluieren, wurde ein eigenes Softwareprojekt erstellt, welches eine exemplarische Sicherheitsarchitektur abbilden soll. Dazu wurde OpenAM als Access Manager verwendet. Darüber hinaus wurde ein Microservice zur Verwaltung von Benutzerprofilen auf Basis von OASP4J erstellt, welcher für die Umsetzung des Konzeptes abgesichert werden sollte. Anschließend wurde eine Methode entwickelt, die ein einfaches Deployment der erstellten Lösung für weitere Microservices ermöglicht. Die Beschreibung dieser Methode ist im Kapitel 3 zu finden.

SSO-Agent

Für die Umsetzung des Sicherheitskonzeptes entschied man sich für den Ansatz mit mehreren SSO-Agents, da uns die Technologiefreiheit und Unabhängigkeit bei der Entwicklung von Microservices von großer Bedeutung war. Im Folgenden wird das Vorgehen während der Evaluation beschrieben.
In einem ersten Schritt wurde der Einsatz eines Web Policy Agents in Verbindung mit OpenAM evaluiert. Als Webserver wurde der etablierte Apache HTTP Server ausgewählt. Das Ziel dieser Untersuchung war die Absicherung der initialen Homepage des Webservers. Dazu wurde eine entsprechende Policy für den Agent angelegt. Diese Policy gewährt nur authentifizierten Benutzern Zugriff auf die gewünschte Ressource. Nach der Aktivierung des Agents wird man beim erstmaligen Aufruf der Homepage zu OpenAM umgeleitet. Dort muss sich der Benutzer authentifizieren. Bei Erfolg, wird von OpenAM ein gültiges SSO-Token generiert und als Session Cookie auf dem Client gespeichert. Dieser Cookie enthält eine verschlüsselte Referenz auf die bei OpenAM gespeicherte Session des Benutzers. Über diese kann OpenAM überprüfen, ob ein Benutzer bereits authentifiziert ist, ob die aktuelle Session noch Gültigkeit besitzt und über welche Rechte der Benutzer verfügt. Sobald die Session beendet wird (manuell oder bei Ablauf eines definierten Zeitintervalls), verliert das SSO-Token seine Gültigkeit.
Für die angestrebte Sicherheitsarchitektur zur Absicherung der Microservices, welche ausschließlich auf Applicationservern bereitgestellt werden, wird jedoch eine andere Variante eines Policy Agents benötigt. Der J2EE Policy Agent stellt eine Implementierung eines Policy Agents dar, der in etablierte Applicationserver integriert werden kann. In diesem Fall wurde die Variante für Apache Tomcat ausgewählt. Das Ziel der zweiten Untersuchung war die Absicherung des Microservice zur Verwaltung von Benutzerprofilen. Dazu wurde wieder eine entsprechende Policy für den Agent angelegt. Nach der Aktivierung des Agents und der Erweiterung des Deployment Descriptor des

Microservice mit einem Eintrag für den Agent, wird man beim erstmaligen Aufruf des Microservices zu OpenAM umgeleitet.

Bibliothekslösung

Nach der erfolgreichen Einbindung des J2EE Policy Agents wurde er auf seine künftige Verwendbarkeit hin überprüft. Während, wie bereits beschrieben, die Authentifizierung auf Anhieb einwandfrei funktionierte, war eine Autorisierung in der gewünschten Form nicht realisierbar. Ziel war, dass Benutzer in der OpenAM-Umgebung einmalig angelegt werden können und dort, je nach den für sie vorgesehenen Rechten, Gruppen zugeordnet werden. Wie die einzelnen Microservices mit diesen Gruppen umgehen, sollte ihnen selbst überlassen sein. Die von OpenAM bereitgestellten Autorisierungsmechanismen, wie das vollständige Blockieren des Zugriffs auf einzelne Kontexte oder die Gestattung von bestimmten Representational State Transfer (REST)-Befehlen, wurde als zu grob bewertet. Vielmehr sollten die Microservices in der Lage sein die Ausführung einzelner Methoden zu autorisieren. Dazu wurde im nächsten Schritt versucht die Gruppen von OpenAM auf die mittels JSR250-Annotation definierten Rollen von OASP mit Hilfe des J2EE Policy Agents zu mappen. Nach mehreren erfolglosen Versuchen entschied man sich schließlich dazu eine eigene, auf dem Bibliotheksprinzip basierende, Lösung zu entwickeln, die im Folgenden vorgestellt werden soll.
Die Authentifizierung wurde über die Representational State Transfer (REST)-API von OpenAM umgesetzt. Die vom Benutzer in eine simple Eingabemaske eingegebenen Credentials wurden per Representational State Transfer (REST) an den OpenAM Server geschickt und von diesem überprüft. Bei erfolgreicher Authentifizierung wurde eine verschlüsselte, für die jeweilige Sitzung gültige ID vom Server zurück gesendet, die vom Agent-Modul als Cookie gespeichert wurde. Durch die ID konnte zum einen die SSO-Funktionalität realisiert werden, da diese auch zum Authentifizieren beim OpenAM-Server benutzt werden konnte. Zum anderen konnte mit Hilfe der ID die Gruppe und alle weiteren auf dem OpenAm-Server gespeicherten Merkmale des Nutzers abgefragt werden. Die so erlangte Gruppe innerhalb von OASP auf die JSR250-Rollen zu mappen war trivial. Mit Hilfe der JSR250-Annotation konnte nun jeder Microservice für jede Methode einzeln festlegen, welche Nutzer berechtigt sein sollten diese auszuführen. Wie bei Bibliothekslösungen üblich musste jeder Microservice die Lösung als Bibliothek einbinden und musste dafür etwas Eigenarbeit aufwenden.

2.3. Wertung

Das Kapitel *Wertung* soll die im Abschnitt 2.2.3 erarbeiteten Ergebnisse zusammenfassen und eine abschließende Empfehlung geben.

2.3.1. Dokumentation

Die Dokumentation zum Enterprise-Security-Modul wirkt leider unvollständig. Vor allem fehlt eine Beschreibung des Projektes beziehungsweise des Sicherheitskonzeptes. Es wurden nur ein paar Abbildungen veröffentlicht, die sich der Leser selbst erklären muss. Hingegen bewerten wir es positiv, dass zu Beginn ein Vergleich zwischen zwei AM-Implementierung vorgenommen wurde, was den Einstieg in das Thema erleichtert. Des Weiteren wird auf die Installation sowie Konfiguration des AM und der für die Umsetzung des Sicherheitskonzeptes benötigten Komponenten eingegangen. Abschließend wird die Integration der einzelnen Komponenten in das Gesamtsystem beschrieben. Die Dokumentation zu OpenAM und den verwandten Technologien ist hingegen sehr ausführlich.

2.3.2. OASP Security

Die in OASP vorgesehener Aufteilung von Security Komponenten auf die Identity- and Access Management und Application Security erlaubt dem Entwickler eine freie Auswahl von Technologien für ID und AM. Die in OASP Security benutzte Spring Security Bibliothek ist heutzutage sehr verbreitet und es gibt bereits viele Beispiele, wie bestimmte Produkte mit Spring Security integriert werden können. Die zweite Komponente von OASP, Application Security, erleichtet die Implementierung von Autorisierung, indem sie für das automatische Mapping von, in einem Identitymanagement definierten, Rollen sorgt. Nur das Access Control Schema muss entsprechend angepasst werden, um die JSR250-Annotation in Use Cases verwenden zu können.
Im Großen und Ganzen wurde eine solche Aufteilung als „microservicefreundlich“ bewertet. Mit Hilfe von OpenAM konnte ein zentrales Identity- and Access Management implementiert werden. Und mittels Access Control Schemata konnte jeder einzelne Microservice für sich selbst entscheiden, welche Funktionalitäten welchen Benutzern zugänglich sind.

2.3.3. OASP4JS Security

Um die Authentifizierungsmechanismus in die User Interface (UI) zu integrieren, wurde versucht das OASP4JS Security Modul in das neu generierte Template des Profile

Microservices aus der Sample Application zu übertragen. Diese Integration konnte leider nicht erfolgreich durchgeführt werden. Wollte ein unangemeldeter Benutzer auf das Profile-Service über SPA zuzugreifen, meldete Angular einen unbekannten Fehler und konnte ihn nicht auf die von Spring Security erzeugte Logging-Page umleiten. Wegen der fehlenden Erfahrung im Bereich SPA Security und dem Zeitmangel konnte die Integration nicht abgeschlossen werden, was die Wertung in diesem Bereich unmöglich gemacht hat.

2.3.4. Access Management

OpenAM als AM-Implementierung zu verwenden hat sich im Laufe unserer Evaluation als eine gute Entscheidung herausgestellt. OpenAM bietet viele Funktionalitäten sowie eine umfangreiche Dokumentation. Die von ForgeRock bereitgestellten SSO-Agents erleichtern darüber hinaus die Integration in die bestehende Systemlandschaft. Die Möglichkeit eigene SSO-Agents zu implementieren unterstützt die Technologiefreiheit und Unabhängigkeit bei der Entwicklung von Microservices.

2.3.5. Bibliothekslösung

Als grundsätzliche Umsetzungsmöglichkeiten für Authentifizierung und Autorisierung bei Microservices existieren die Konzepte der Portallösung, wie sie das Enterprise-Security-Modul vorsieht, und die von uns als Bibliothekslösung benannte Alternative. Die bereits im Abschnitt 2.1.2 aufgelisteten Vor- und Nachteile der beiden Prinzipien wurden gegeneinander abgewogen und es wurde sich schließlich für die Bibliothekslösung entschieden. Hauptargument für die schließlich verwendete Bibliothekslösung war die Vermeidung eines Single Point of Failure, der dem Microservice-Gedanken entgegen steht. Aus diesem Grund wird die Entscheidung, die Bibliothekslösung zu nutzen positiv bewertet. Die in Abschnitt 2.2.3 beschriebene Umsetzung konnte ohne größere Probleme mittels OASP umgesetzt werden. Auch die spätere Einbindung der Bibliothek verlief reibungslos, da hauptsächlich bereits existierende Authentifizierungs- und Autorisierungsklassen erweitert oder neu geschrieben wurden. Abschließend kann OASP zur Umsetzung empfohlen werden.

3. Continuous Integration und Continuous Deployment

Nachdem das letzte Kapitel bereits ein Querschnittsthema behandelte, beschäftigt sich auch das nachfolgende Kapitel mit einem Aspekt, welcher für jeden Microservice von Bedeutung ist, Continuous Integration und Continuous Delivery/Deployment. Dabei wird zunächst auf allgemeine Aspekt des Continuous Integration und Continuous Delivery/Deployment-Ansatzes eingegangen, anschließend spezielle Eigenschaften beleuchtet und die Umsetzung und der Einsatz im Projekt aufgezeigt. Des Weiteren erfolgt eine Beschreibung und Evaluation des Deployments mit Docker und in die Cloud, sowie eine abschließende Wertung für die Umsetzung im Projekt und die Unterstützung durch OASP.

3.1. CI und CD im Allgemeinen

Um Softwareauslieferungsprozesse besser steuern zu können wird eine Sammlung von Werkzeugen, Techniken und Prozessen zur Unterstützung genutzt, die auch als Continuous Delivery/Deployment (CD) bezeichnet werden. Um dem Entwickler ein schnelles Feedback über seinen geschriebenen Code zu geben, wird ihm in der Continuous Integration, auch CI genannt, mitgeteilt, ob der von ihm entwickelte Programmiercode einen Fehler verursacht. Hierfür wird der neu geschrieben Quelltext aller Entwickler mehrmals täglich mit einer Mainline-Version des Projekts gemerged und durchläuft daraufhin die CI-Pipeline. Die nachfolgende Abbildung veranschaulicht das Verfahren.

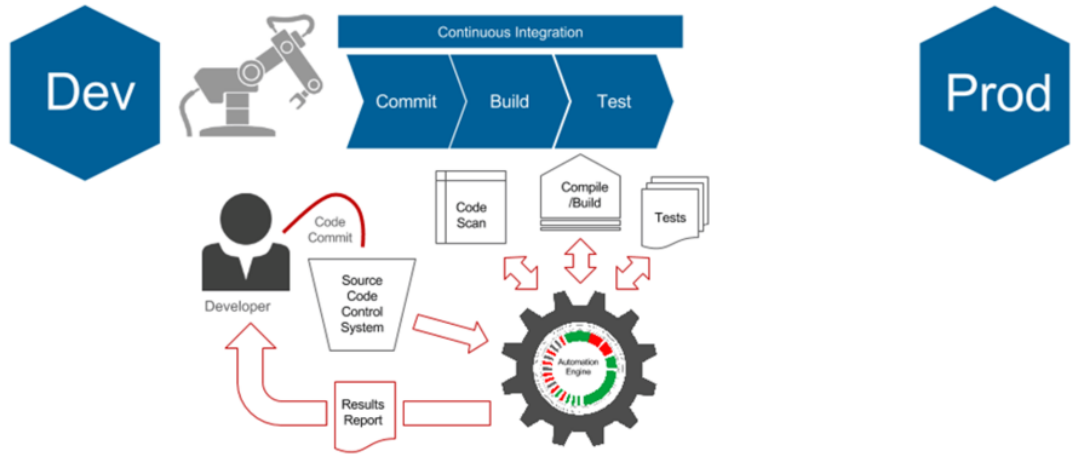

Abbildung 3.1.: CI -Pipeline [Cygan, 2015]

Beim Continuous Delivery/Deployment wird die Applikation für den Einsatz unter realen Einsatzbedingungen vorbereitet. Dabei wird berücksichtigt, wie sich das Verhalten des Benutzers auf im Applikation im echten Betrieb auswirkt. Dies ist ein wesentlicher Unterschied zum CI-Prozess, der dem Entwickler nur mitteilt, falls die Integrationstest aufgrund der fehlgeschlagenen Validierung seines Quellcodes fehlschlagen. [Cygan, 2015]

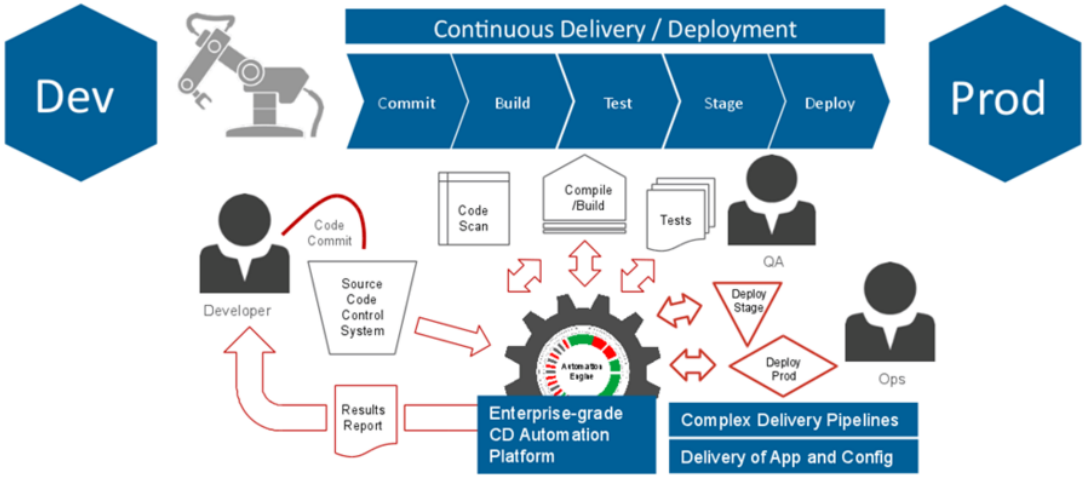

Abbildung 3.2.: Continuous Delivery/Deployment [Cygan, 2015]

3.1.1. Vorteil CD gegenüber CI

Einen Organisationsrahmen für die Entwicklungsteams wird durch CI zur Verfügung gestellt, da auf diese Weise eine sinnvolle Aufgabenteilung erreicht werden kann. Weiterhin spielt die Softwarefreigabe eine wichtige Rolle, da der geschriebene Code die nötigen Tests, Qualitätskontrollen und den betrieblichen Einsatz durchlaufen muss. Dies hat zur Folge, dass die umgesetzten Erneuerungen und Features nach der Realisierung nicht direkt eingesetzt werden und den Benutzern nicht sofort zu Verfügung stehen. Ein zeitnahes Feedback, ob die geforderten Anforderungen ihre definierten Aufgaben erfüllen, kann der Entwickler aufgrund der langen Zykluszeiten nicht erhalten.
Um neue Features und Verbesserungen schneller auf den Markt zu bringen, ist eine Erweiterung von CI zu CD sinnvoll. Dies bringt den Vorteil mit sich, dass kleine inkrementelle Änderungen an der Software direkt mit messbarem Feedback durch den Benutzer erfolgen. Änderungen an der Software oder vorherige Stände wiederherzustellen, falls die Entwicklung nicht nach Plan verläuft, wird hierdurch ebenfalls vereinfacht. Bevor die Software in das nächste Stadium übergeht, werden alle Testes ausgeführt. Dadurch wird insgesamt eine bessere Qualität des Codes erreicht. Nichtfunktionale Anforderungen können durch Einführung von „Quality Gates“ in wichtigen Stadien der Pipeline sichergestellt werden. CD und CI verwenden die gleichen Softwareprinzipien. Dies ermöglicht eine bessere Nachvollziehbarkeit des Quellcodes bis zur Lieferung der Software-Applikation. Somit kann nachvollzogen werden wer welche Teile des Codes wann umgesetzt hat.
Bei CI und CD spielen sowohl Tools als auch Mitarbeiter eine wichtige Rolle. Die Implementierung einer Continuous Delivery/Deployment-Pipeline erzeugt eine gemeinschaftliche Antriebskraft und eine einheitliche Sichtweise auf das Projekt bei den beteiligten Teams. [Cygan, 2015]

3.1.2. CI-Pipeline für Java mit Maven

In diesem Abschnitt wird erläutert, wie eine typische CI-Pipeline für Java mit Maven ausgeführt wird. Diese sieht wie folgt aus:

1. Mit dem befehl „maven compile“ wird im ersten Schritt der Quellcode kompiliert.
2. Im zweiten Schritt werden, mit dem Befehl „maven test“, alle definierten Unit-Tests ausgeführt.
3. Anschließend wird der kompilierte Code in eine jar-Datei gepackt. Dies geschieht mit dem Befehl „maven package“.
4. Der Befehl „maven install“ fügt die Artefakte zum lokalen Repository hinzu.

5. Zu guter Letzt werden mit dem Befehl „maven deploy“ die Artefakte zu einem Remote-Repository hinzugefügt.

6. Die entstandene WAR-Datei kann nun in ein Testsystem hochgeladen und getestet werden.

Continuous Integration-Bedingungen
Bei der Einführung von Continuous Integration sollten die Punkte Organisation des Source Codes, Integration von Änderungen und Pflege des Source Codes beachtet werden.

Organisation des Source Codes Bei der Organisation des Source Codes sollte der aktuell entwickelte Code stets in einem Repository liegen. Elemente die dem Projekt zugehören sollten ebenfalls im gleichen Repository zu finden sein. Zusätzlich sollte der Entwickler über eine möglichst aktuelle Version des Repository lokal in seinem Workspace verfügen. Jedes Team, Projekt beziehungsweise Microservice soll über eine eigene Mainline verfügen. So können Merge-Konflikte vermieden werden.

Integration von Änderungen Es sollte die Möglichkeit bestehen, durch bestimmte Kommandos automatisiert Builds zu erzeugen. Ebenfalls sollte eine Möglichkeit existieren den Code lokal zu kompilieren. Um Merge-Konflikte zu vermeiden sollten die Änderungen mindestens einmal pro Tag Committed werden. Die durchgeführten Änderungen sollten möglichst automatisiert getestet werden. Die eingesetzte Testumgebung sollte der realen Umgebung möglichst stark ähneln.

Pflege des Source Codes Falls Probleme in der Mainline auftreten, so sollten diese hoch priorisiert und anschließend schnellstmöglich behoben werden. Die Kommunikation zwischen den Teams sollte stets aufrecht gehalten werden, um wichtige Fragen, wie beispielsweise „Ist der Mainline-Build funktionstüchtig“ schnell beantworten zu können. Eine Historie der Änderungen erleichtert die Nachvollziehbarkeit des entwickelten oder überarbeiteten Codes.

3.1.3. CI/CD im Projekteinsatz

Um den eingecheckten Code regelmäßig zu testen und auf bestimmte Merkmale zu prüfen, wurde im Projekt eine Continuous Integration-Pipeline sowie eine Continuous Deployment-Pipeline eingerichtet. So war es möglich Fehler im Code frühzeitig zu erkennen und zu beheben.
Bei jeder Änderung im zentralen GitLab-Repository, die durch einen Commit ausgeführt wurde, soll Jenkins das Projekt auschecken, bauen und die enthaltenen Tests aus-

führen. Die Einhaltung der OASP-Coding Conventions wurde mit Hilfe von Checkstyles überprüft. Hierfür musste folgendes konfiguriert werden:

Buildverfahren

- Der Befehl „clean install“ im Bereich Maven Goals aufrufen. Dieser Befehl führt automatisch Tests aus.
- Ebenfalls im Maven Goals den checkstyle:checkstyle -Dcheckstyle.config.location="checkstyle.xml" aufrufen.
- Der Bereich „Post-Build-Aktionen“ dient hierbei für die Veröffentlichung der Ergebnisse der Checkstyle Analyse. Dieser kann leer bleiben.

Abbildung 3.3.: Einstellungen Buildverfahren

Im Folgenden sind ein Report, sowie ein Verlauf der Checkstyle-Warnings abgebildet:

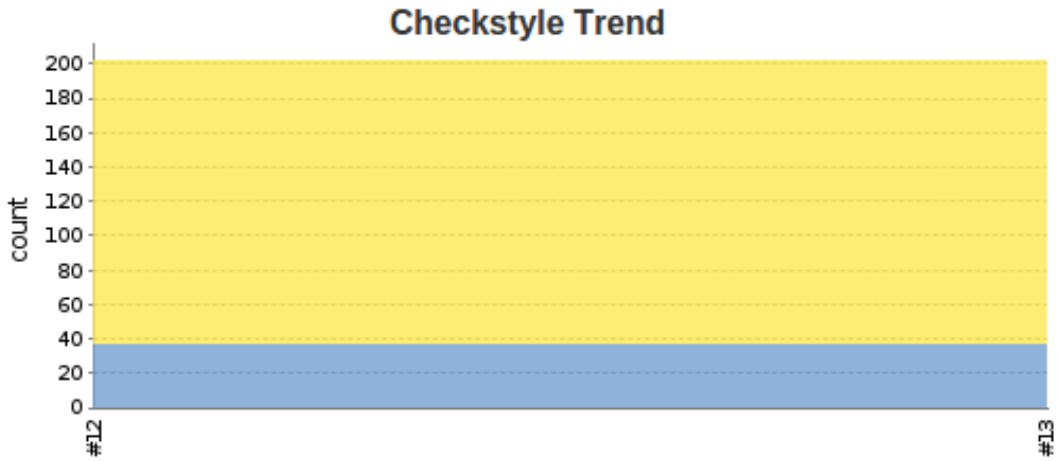

Abbildung 3.4.: Checkstyle Trend

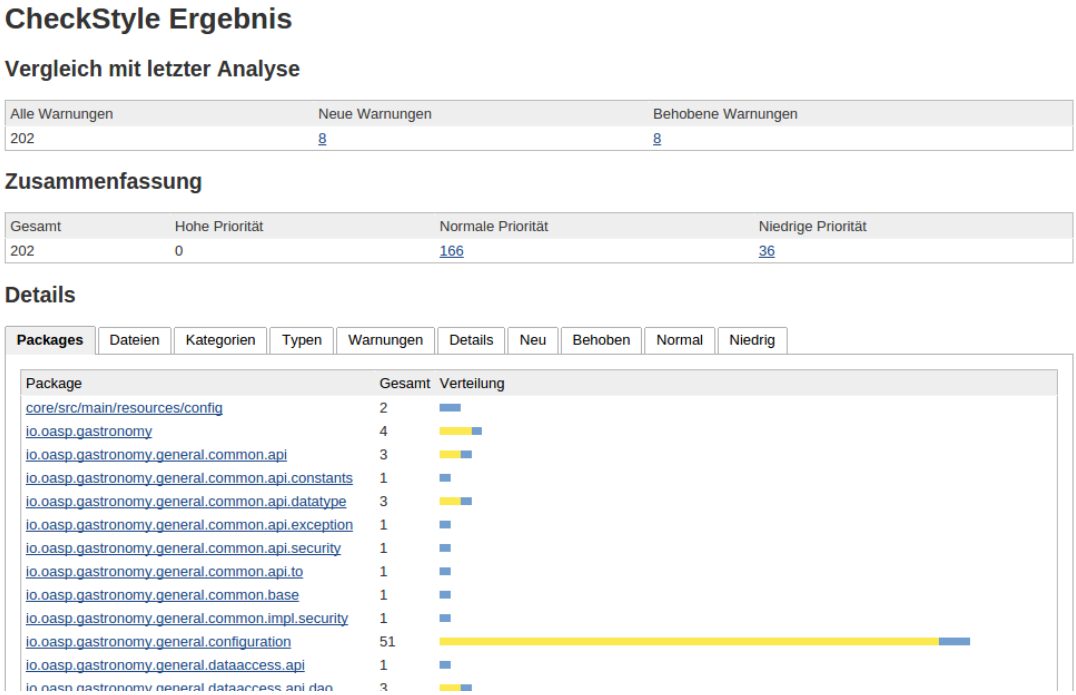

Abbildung 3.5.: CheckStyle Ergebnis

Die Continuous Deployment-Pipeline wurde wie folgt konfiguriert:

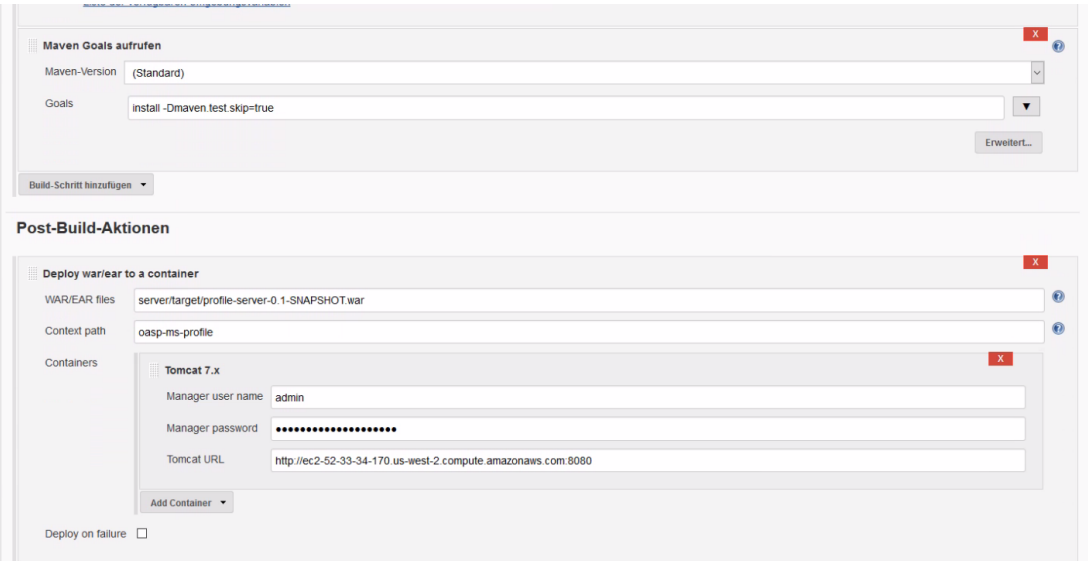

Abbildung 3.6.: Konfiguration der Continuous Deployment-Pipeline

Im Gegensatz zu der CI wurden hier im Bereich „Maven Goals aufrufen“ keine Befehle eingetragen.
Der Bereich „Post-Build-Aktionen“ dient zum Deployment auf einen Server oder in die Cloud. Hier müssen der Benutzername, das Passwort sowie die URL des Servers beziehungsweise der Cloud-Instanz angegeben werden.

3.2. Deployment mit Docker

Das Deployment der Microservices über die eingesetzte Continuous Deployment-Pipeline und insbesondere der, in Kapitel 2 beschriebenen, A&A-Lösung sollte mit Hilfe von Docker erfolgen. Dies begründet sich durch die, im Rahmen des Microservices-Ansatzes verfolgten, möglichst autonomen Entwicklung der einzelnen Services sowie der angestrebten unabhängigen Freigabe und Inbetriebnahme dieser durch die jeweiligen Teams. Aus diesem Grund passt Docker durch das verwendete „Single-Process“-Modell ideal zu der angestrebten Vorgehensweise. [Pressler u. Tigges, 2015]
Docker ist ein Werkzeug zur Virtualisierung auf Betriebssystemebene. Anders als bei der herkömmlichen Hardwarevirtualisierungen läuft die virtuelle Umgebung hierbei direkt auf dem Kernel des Host-Systems. Hierdurch ist diese deutlich leichtgewichtiger, benötigt weniger Arbeitsspeicher und somit ist es möglich mehr Instanzen auf einer identischen Hardware zu betreiben. Die zentrale Komponente einer Docker-Installation ist der sogenannte Docker Daemon. Dieser verwaltet die lokalen Container und Images

sowie die Netzwerkkomponenten und das Dateisystem. Docker Images können als Blaupause einer virtuellen Umgebung betrachtet werden und enthalten das Dateisystem, welches vom Kernel benötigt wird, um die jeweilige Anwendung auszuführen. Container bilden bei Docker die eigentliche virtuelle Umgebung. Wenn ein Container gestartet wird, wird das Dateisystem, welche die unveränderlichen Daten enthält, erzeugt und eingebunden und mit einem Read/Write-Layer kombiniert. Anschließend wird ein Prozess durch den Docker Daemon gestartet, der vom restlichen Hostsystem isoliert ist und die Anwendung des Images ausführt. Der Anwender kann mit Hilfe eines Docker Clients den Docker Daemon steuern und hierbei werden die Kommandozeilenbefehle über eine Representational State Transfer (REST)-API an den Docker Daemon weitergeleitet. [Pressler u. Tigges, 2015]
Die nachfolgende Abbildung 3.7 zeigt die Unterschiede zwischen der Virtualisierung über Docker-Container (rechts) und über herkömmliche virtuelle Maschinen (links). Dabei wird die zuvor angesprochene Einsparung auf Betriebssystemebene und der damit verbundene geringere Bedarf an Ressourcen deutlich, welcher durch die Docker Engine ermöglicht wird. [Pressler u. Tigges, 2015]

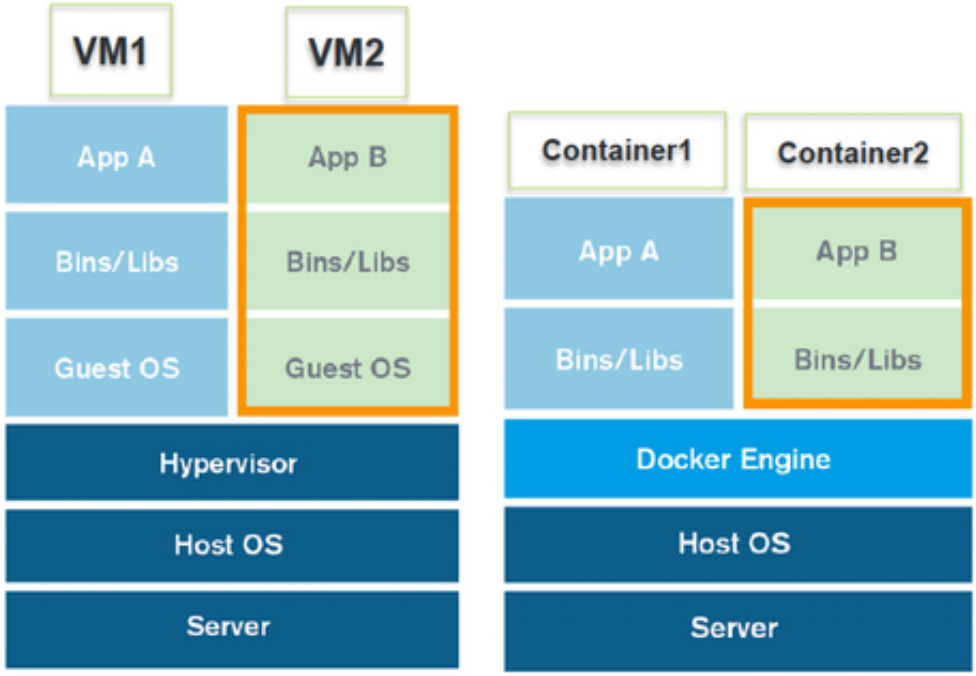

Abbildung 3.7.: Virtuelle Maschinen und Docker-Container im Vergleich [Tinatigertech, 2015]

3.2.1. Evaluation von Docker

Die Evaluation von Docker als Deployment-Konzept und Docker-Container als Organisationseinheit für die Microservices erfolgte zunächst in einem kleinen Rahmen auf einem lokalen Rechner. Hierzu wurde Docker auf einem Windowsrechner installiert. Da Docker ursprünglich nur unter Linux ausgeführt werden kann, musste zu diesem Zweck auf eine offizielle Lösung für Windows zurückgegriffen werden. Diese ist als native Windows-Anwendung verfügbar und benutzt Hyper-V zur Virtualisierung der Docker-Umgebung und des Linux-Kernel für den Docker Daemon. [Docker, 2016]
Zu Evaluierungszwecken wurde die zuvor ausgewählte A&A-Lösung OpenAM als Docker-Image installiert. Hierzu wurde zunächst ein lauffähiges Docker Build File von OpenAM ausgesucht. Dies erwies sich als schwierig, da sehr viele fehlerhafte oder veraltete Docker Build Fils oder Images zu finden waren. Anschließend wurde aus dem Docker Build File von OpenAM ein Docker Image erstellt. Dieses konnte daraufhin unter Angabe der passenden Ports und einer spezifischen Hostadresse ausgeführt und konfiguriert wer-

den. Darüber hinaus mussten die Host-Dateien von Windows und Docker angepasst werden, da aufgrund der doppelten Virtualisierung im Rahmen der Linux-Instanz und des Docker Images nur auf diese Weise eine Netzwerkweiterleitung vom Hostrechner zum Docker Container und umgekehrt erfolgen konnte.
Auf eine Umsetzung und Evaluierung von Docker-Images als Entwicklungs-, Build-, Test- und Deployment-Umgebung wurde, auf Grund der bereits bestehenden CI/CD-Landschaft, verzichtet.

3.3. Deployment in die Cloud

Im Laufe des Projekts wurde durch die Projektbeteiligten entschieden, dass das Deployment der Microservices und der A&A-Lösung in der Cloud erfolgen sollte, um eine möglichst realistische und flexible Bereitstellung der Services zu erzielen. Nachdem bereits zuvor erste Erfahren mit einer lokalen Umsetzung mit Docker erfolgt war, wurde nun im Anschluss die Umsetzung und Evaluation über einen Docker Hosting Service realisiert. Hierzu wurden zunächst mehrere Docker Hosting Services verglichen. Unter den möglichen Kandidaten für die spätere Umsetzung waren Tutum, sloppy.io, Quay.io, die Google Cloud Platform, Orchard sowie Amazon Web Services. Letztendlich fiel die Entscheidung auf die Amazon Web Services (AWS) begründet damit, dass zum einen das Hosten in einem gewissen Rahmen hier kostenlos erfolgen konnte, AWS über eine sehr umfangreiche Dokumentation verfügt und es sich bei AWS um den weltweit größten Cloud-Hosting Service handelt. [Ward, 2016]
Nachdem die Entscheidung auf AWS gefallen war, fand die eigentlich Evaluation des Deployment in die Cloud statt. Hierzu wurde eine virtuelle Maschine, in diesem Fall eine Amazon Linux AMI, im Container-Management-Service des Amazon Elastic Compute Cloud (EC2) erstellt, mit Hilfe dessen es möglich ist Docker-Container in einer virtuellen Maschine in der Cloud zu betreiben. An der Instanz der Amazon Linux AMI wurden zunächst die gewünschten Konfigurationen vorgenommen. Dabei musste in den Security Group-Einstellungen eine neue Regel für eine http-Verbindung (TCP, Port 80) hinzugefügt und zudem ein SSH-Key-Pair erzeugt und gespeichert werden, um im Anschluss auf die Instanz über eine SSH-Verbindung zugreifen zu können. [Amazon, 2016b]
Im Anschluss an die Erstellung der Instanz wurde über die zuvor erwähnte SSH-Verbindung auf die Amazon Linux AMI zugegriffen und diese daraufhin konfiguriert. Dabei erfolgte zunächst die Installation von Docker anhand einer von Amazon bereitgestellten Anleitung. [Amazon, 2016a]
Anschließend wurde, wie in Abschnitt 3.2.1 bereits beschrieben, das Docker-Image von OpenAM auf der Amazon Linux AMI installiert. Abweichend zu dem lokalen Test

musste die Host-Datei der Amazon Linux AMI-Instanz mit der entsprechenden Public DNS ergänzt werden und der Start des Docker-Containers von OpenAM erfolgte unter Angabe des gewünschten Ports und der Public DNS der Instanz.
Um die Evaluation und das beispielhafte Deployment des Profile-Microservice über die bestehende CD-Pipeline in die Cloud zu ermöglichen, musste die bestehende Konfiguration der CD-Pipeline geringfügig angepasst werden. Zudem wurde eine weitere Instanz einer Amazon Linux AMI über einen zweiten Amazon-Account erstellt und konfiguriert. Hierbei wurde, wie auch zuvor, zunächst Docker installiert und eine Tomcat-Instanz als Docker-Container ausgeführt. Dabei musste darauf geachtet werden, dass dieser Docker-Container im Detach-Mode gestartet wird, damit dieser als Hintergrundprozess ausgeführt wird und somit immer zur Verfügung steht. Daraufhin fand die Anpassung der CD-Pipeline statt, bei der ein neuer Jenkins-Job erstellt wurde, der als Post-Build-Aktion den lauffähigen und durch die CI-Pipeline getesteten Microservice in den laufenden Docker-Container der Tomcat-Instanz auf der Amazon Linux AMI deployed.

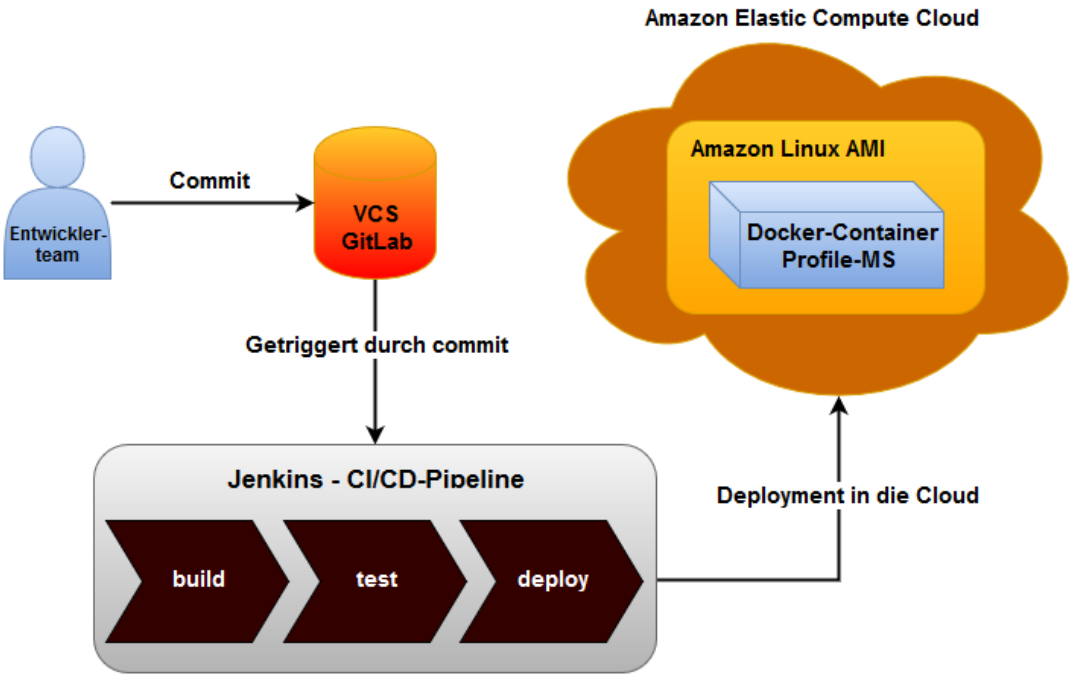

Abbildung 3.8.: Abstrakte Darstellung des Ablaufs beim Deployment in die Cloud

Die vorangegangene Abbildung 3.8 zeigt den abstrakten Ablauf beim Deployment des Profile-Microservice in die Cloud. Der Ablauf beginnt hierbei beim Entwicklerteam (oben links). Wenn ein Commit durch einen Entwickler in das Repository des Profile-

Microservice erfolgt, wird daraufhin die CI- und CD-Pipeline angestoßen. Jenkins, welcher die CI/CD-Landschaft bildet, zieht sich daraufhin den aktuellen Stand des entsprechenden Repository und führt anschließend den Build, die konfigurierten Tests und das Deployment durch. Das Deployment erfolgt dabei in die Amazon Elastic Compute Cloud und der Tomcat-Instanz, welche als Docker-Container auf der Amazon Linux AMI ausgeführt wird.

3.4. Wertung

Im nachfolgenden Kapitel soll die im Projekt verwendete Continuous Integration- und Continuous Deployment-Pipeline in Hinblick auf die Verwendung im Projekt, bei der Entwicklung von Microservices und in Verbindung vom Einsatz von Docker bewertet werden.

3.4.1. Entwicklung von Microservices in Verbindung mit CI/CD

Die Entwicklung von Microservices sieht die Aufteilung eines Monolithen in fachlich definierte Microservices vor. Dabei soll die Entwicklung der einzelnen Services durch die Entwicklerteams möglichst unabhängig voneinander erfolgen. Außerdem sollen die Services möglichst schnell in den Betrieb übergehen können, um Änderungen zeitnah dem Kunden zur Verfügung zu stellen. Auf diese Weise besteht die Möglichkeit zeitnah Feedback durch den Benutzer einzuholen. Um den damit verbundenen Anforderungen gerecht zu werden, wurde, wie bereits in Abschnitt 3.1 beschrieben, eine Continuous Integration und Continous Deployment-Landschaft genutzt. Somit war es im Projekt möglich, die, durch die einzelnen Teams erstellten, Microservices mit Hilfe der CI/CD-Pipeline nach dem einchecken von Änderungen in das VCS automatisch auf Fehler hin zu untersuchen und nach erfolgreichem Test automatisch bauen zu lassen. Im späteren Projektverlauf war es darüber hinaus durch die CD-Pipeline möglich den Microservice nach erfolgreichem Durchlauf durch die CI-Pipeline automatisch bereitzustellen. Durch den, in Abschnitt 3.2 und 3.3 beschriebenen, Ausbau der bestehenden CI/CD-Landschaft konnten zudem erste Erfahrungen gesammelt werden, wie ein dynamisches Deployment über Servergrenzen hinweg realisiert werden kann. Zudem wurde sich damit beschäftigt, wie eine Skalierung der Microservices in Verbindung mit Docker und dem Cloud-Ansatz zu realisieren wäre. Es konnte durch das Einhalten der Vorgaben und Empfehlungen, wie Beispielsweise den stetigen Commits von kleinen Änderungen, einer Mainline im Git-Repository pro Microservice und der Funktionstüchtigkeit der Mainline als Hauptpriorität bei der Entwicklung ein gewisser Grad der Automatisierung im Bereich des Deployments und der Verfügbarkeit der Microservices erreicht werden.

3.4.2. Docker in Verbindung mit CI/CD

Docker erwies sich beim Verfolgen des Microservices-Ansatzen im Projekt als sehr hilfreich. Durch den Einsatz von Docker und der damit verbundenen ressourcenschonenden Virtualisierung sowie der Empfehlung einen Docker-Container als Organisationseinheit für einen Microservice zu nutzen, war es möglich die Microservices und deren Entwicklungen gut voneinander abzugrenzen. Theoretisch könnte man somit dynamisch auf Schwankungen der Last auf einzelne Microservices reagieren, indem weitere Instanzen des Microservice als Docker-Container hinzugeschaltet werden. Auch wenn der Einsatz von Docker in Verbindung mit dem Deployment in die Cloud nur begrenzt im Projekt bei zwei Microservices, dem Profile-Microservice und OpenAM, zum Einsatz kam, konnte so bereits das mögliche Potenzial aufgezeigt und in einem gewissen Rahmen analysiert und evaluiert werden.

4. Inter-Service Kommunikation

Dieses Kapitel beschäftigt sich mit der Kommunikation zwischen den Microservices. Es werden Punkte wie Representational State Transfer (REST)-Schnittstelle und Formate für die Kommunikation zwischen Microservices näher betrachtet und durch Evaluationskriterien wie Modifizierbarkeit oder Konsistenz beurteilt. Im Gegensatz zu den anderen Kapiteln, wird es hier keine Wertung für OASP und den Microservices-Ansatz geben, da diese Punkte nicht tief genug behandelt wurden, um eine fundierte Wertung geben zu können.
Für die Kommunikation zwischen den Services sind folgende Evaluationskriterien relevant:

- **Ausfallsicherheit** - funktioniert der Service beim Ausfall eines abhängigen Services? Wie aussagekräftig sind die Daten ohne den abhängigen Service?
- **Konsistenz** - wie stellt der Service sicher, dass die Daten immer konsistent sind?
- **Modifizierbarkeit** - wie schwer ist es, die Antwort eines Services für eigene Zwecke anzupassen?
- **Funktionalität** - bietet der Service alle Möglichkeiten an, die es vor der Abtrennung vom Monolithen hatte? Haben die Funktionen den selben Umfang oder sollten diese angepasst oder erweitert werden?
- **Effizienz** - ist der Service genau so schnell wie vor der Abtrennung vom Monolithen? Wie viel Kommunikation benötigt der Service?

4.1. API-Spezifikation

Damit zwei Microservices miteinander kommunizieren können, müssen sie wissen, wie sie einander erreichen können. Aber die Kenntnis über die URL ist noch lange nicht aussagekräftig, wenn man nicht weiß, was man überhaupt zurückbekommt.
Durch eine API-Spezifikation gibt man den Gesamtüberblick über alle Möglichkeiten, die ein Microservice bietet. Die Beispielspezifikation einer URL wird in Tabelle 4.1 dargestellt.

Titel	**Beispiel**
Path	recipe/rating/id
Method	GET
Kurze, aussagekräftige Beschreibung, was man durch die URL erreicht	Gibt eine bestimmte Bewertung zu einem Rezept zurück
Request header	application/json
Request body	“id”: id
Responce header	html
Responce body	HTML-Snippet

Tabelle 4.1.: Beispiel Spezifikation.

Der **Path** definiert die URL mit der man eine Funktion des Services erreichen kann. Im Beispiel übergibt man durch die URL eine Variable, die ein bestimmtes Element anspricht. Die Übergabe einer Variablen durch den Path ist nur bei GET-Methode möglich.
Mit der **Methode** definiert man das Vorhaben, das man durch den Aufruf erreichen will:

- **POST** schickt die Daten an den Server, um diese zu speichern oder auf Richtigkeit zu prüfen. Zum Beispiel wird bei erfolgreichem Login ein Token zurück geschickt, sonst eine Fehlermeldung.
- **GET** holt einen Datensatz oder eine Liste von Daten vom Server.
- **DELETE** löscht einen Datensatz auf dem Server.
- **PUT** legt neuen Datensatz auf dem Server an.

Request und **response header** beschreiben den Datentyp der Daten die man sendet, beziehungsweise vom Server empfängt.
Der **body** enthält die Daten, im angegebenen Format.

Kann man eine URL mit mehreren Methoden erreichen, muss jede einzeln definiert werden.
Damit kann ein Team schnell und einfach Anfragen an den Microservice anderer Teams erstellen und senden ohne Kontakt aufzunehmen.
Um andere Teams zu unterstützen, muss die Spezifikation immer aktuell gehalten und auf Änderungen in der Spezifikation hingewiesen werden.

4.2. Übertragungsformate

Um Daten zwischen den Services zu übertragen, muss man sich auf ein bestimmtes Transportform (Format) einigen. Bisher wurden bereits zwei Formate erwähnt (JSON und HTML-Snippet).

JSON ist standardisiert. Man kann es auf beliebigen Plattformen erstellen und auslesen. Es ist auch sehr leicht umzusetzen, denn es hat nur zwei Datentypen: JSONObject und JSONArray. Beide können nur primitive Datentypen beinhalten, wie Zahlen, Text und boolean-Werte. Außerdem können JSONObject und JSONArray in sich weitere JSONObjects und JSONArrays beinhalten, womit ein JSON unendlich ineinander geschachtelt werden und eine beliebige Menge an Information transportieren kann. Dabei bleibt es sehr gering in der Größe und ist damit auch für große Datenmengen gut geeignet. Die geringe Größe spiegelt sich in der schnellen Datenübertragung wieder.
Doch da JSON nur einfache Daten übertragen kann, ist er für Stream-Übertragungen nicht geeignet. Außerdem wird JSON immer als Baumstruktur gelesen: von oben nach unten, und kann die Daten nicht sortieren oder zuordnen. Somit ist es auch für die Übertragung von Metadaten oder Kommentaren nicht geeignet.
Wenn man die mit JSON übertragenen Daten dem Nutzer anzeigen möchte, so muss man dann eine Ansicht dafür bilden, was Zeit kostet. Dies ist aber nicht der Fall bei HTML-Snippets.

HTML-Snippets liefern im Response eine fertige Ansicht, die direkt dem Nutzer präsentiert werden kann. Hat man im Snippet Interaktionsmöglichkeiten, so sind diese bereits in dem Snippet implementiert und man benötigt keinen zusätzlichen Aufwand. Doch wenn die Ansicht angepasst werden soll, ist dies nur schwer möglich und verursacht viel Aufwand.
Um ein HTML-Snippet zu lesen braucht man außerdem eine Bibliothek, wobei man Zeit investieren muss, um eine für eigene Zwecke passende Bibliothek zu finden.

Message Queue ist eine weitere Form des Transports der Daten. (siehe Abbildung 4.1)
Message Queues arbeiten asynchron. Das bedeutet man kann eine Nachricht mit Informationen an einen Service schicken, definieren was als Antwort erwartet wird und dann weiter arbeiten, ohne auf den Antwort zu warten. Sobald die Antwort kommt, wird diese verarbeitet und dem Nutzer bereitgestellt. Die Message Queue arbeitet in Stapelverarbeitung. Man kann mehrere Nachrichten übermitteln und die Message Queue stellt sicher, dass auch diese in der gleichen Reihenfolge verarbeitet werden. Der Consumer

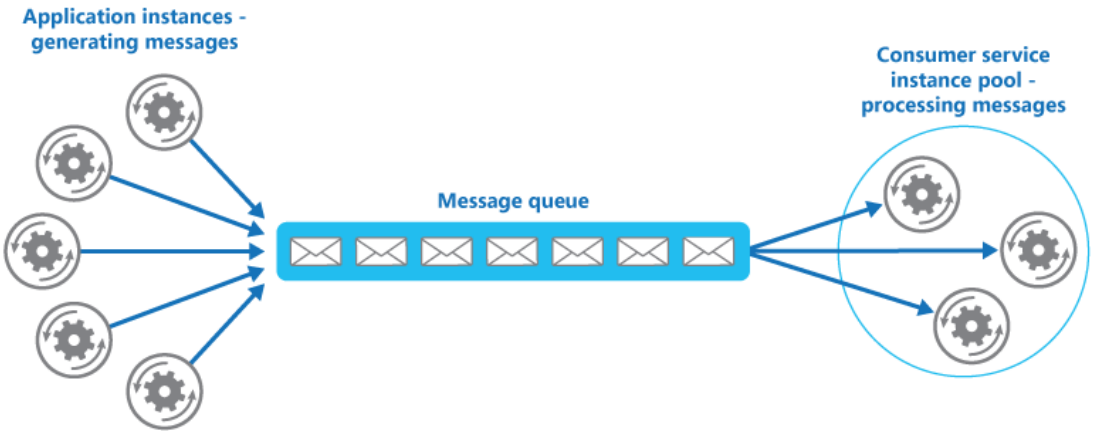

Abbildung 4.1.: Message Queue

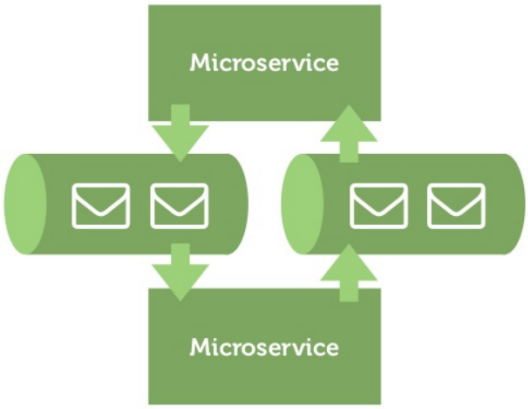

Abbildung 4.2.: Message Queue Microservices

Service verarbeitet die Nachrichten synchron nacheinander und schickt per Message Queue Eintrag eine Antwort an die Instanz zurück. Sobald die Applikation die Nachricht bekommt, wird diese für den Nutzer bereitgestellt.
Die Nachrichten in einer Message Queue bleiben persistent, bis sie verarbeitet werden. Ein Fehler während der Verarbeitung erhöht nur die Wartezeit auf die Antwort.
Message Queueing ist ein Framework, das zwischen den Microservices agiert. Entscheidet man sich für den Einsatz von Message Queues, so sind alle Microservices von diesen abhängig. Bei einem Ausfall der Message Queue werden alle Anfragen zwischen den Microservices nicht bearbeitet. Außerdem kann eine hohe Anzahl an Messages die Antwortzeiten für alle Microservices stark verlängern und dadurch das ganze System verlangsamen.

4.3. Datenkonsistenz

Arbeiten zwei Microservices miteinander, muss einer was von dem anderen Wissen. Beispielsweise werden in einem Rezepte gespeichert und in einem anderen die Bewertungen zu den Rezepten. Um die Bewertungen nun den richtigen Rezepten zuzuordnen, brauchen die Bewertungen die IDs der Rezepte.
Egal für welches Transportformat man sich entscheidet, bei jeder Anfrage an den Bewertungsservice muss der Rezeptservice die ID des Rezeptes mitschicken, das abgefragt oder manipuliert wird. Ist eine Abfrage erfolgreich, so bekommt man die Daten über die Bewertung. Bei einer Manipulation, wie Erstellung, Änderung oder Löschung einer Bewertung, braucht man eine Rückmeldung, ob die Abfrage erfolgreich war, denn sonst kann dies zu einem Fehler auf Seiten des Bewertungsservices führen, da die Daten nicht mit den vorliegenden Daten übereinstimmen. Ein anderes Beispiel kann es deutlicher machen.
Der Bewertungsservice kann anhand eines Rezeptes dem Nutzer die Rezepte vorschlagen, die man probieren sollte. Wird im Rezeptservice ein Rezept gelöscht ohne eine Benachrichtigung darüber an den Bewertungsservice zu geben, so kann der Bewertungsservice die Vorschläge mit Rezepten zurückgeben, die im Rezeptservice nicht mehr existieren. So muss der Rezeptservice immer bei relevanten Änderungen mit Bewertungsservice kommunizieren.
Es kann auch vorkommen, dass bei der Kommunikation der Bewertungsservice unerreichbar ist. In einem solchen Fall muss die Anfrage gespeichert und später wieder ausgeführt werden, sonst kommt es wieder zu Dateninkonsistenzen.

5. User Interface - Integration

Anders als bei anderen Architekturstilen verfolgt der Microservices-Ansatz das Vorgehen, keine dedizierte Benutzerschnittstelle für die Anwendung zu haben. Wie in Kapitel 4 bereits beschrieben, soll auch diese Schicht klassischer Architekturen aufgeteilt und in den zugehörigen Microservice integriert werden. Dafür werden in diesem Kapitel allgemeine Umsetzungsmöglichkeiten für diese Integration dargestellt werden, um anschließend anhand von konkreten Beispielen das Vorgehen im Projekt zu detailieren. Zum Abschluss des Kapitels soll das Vorgehen allgemein und der Einsatz von OASP in diesem Aspekt bewertet werden.

5.1. Umsetzungsmöglichkeiten

Die Integration von Microservices auf der User Interface (UI)-Ebene kann auf verschiedene Wege realisiert werden. [Wolff, 2015, Kap.9.1] unterscheidet zwischen zwei Umsetzungsmöglichkeiten der User Interface (UI) im Kontext von Microservices: Single Page Application (SPA) und HTML-basierte Anwendungen.
Bei der SPA wird ein Teil der Anwendungslogik in das Frontend ausgelagert. Hier handelt es sich meistens um Bibliotheken wie AngularJS, Ember.js oder ExtJS, die solche Umsetzung erlauben. Im Gegensatz zur SPA muss bei den HTML-basierten Oberflächen nicht die ganze Anwendung vom Server geladen werden, sondern nur zum Zeitpunkt benötigte HTML-Seiten.
Da OASP im Frontend bereits einen SPA-Ansatz verfolgt, wurde dieser für unser Projekt als erstes in Betracht gezogen. Hier gibt es zwei Umsetzungsmöglichkeiten:

1. **SPA pro Microservice**: jeder Microservice bekommt eine eigene unabhängige SPA. Mittels Links können dann diese SPAs verbunden werden. Wie auf der Abbildung 5.1 zu sehen ist, passt dieser Ansatz nur dann gut, wenn die Microservices von unterschiedlichen Benutzergruppen verwendet werden oder wenn das Umschalten zwischen Microservices eher eine Ausnahme ist.

2. **Eine SPA für alle Microservices**: die User Interface (UI) eines Microservices besteht aus einem oder mehreren Module, die dann in eine SPA zusammengeführt werden. Ein solcher Ansatz lässt sich zum Beispiel mit dem Modul-Konzept

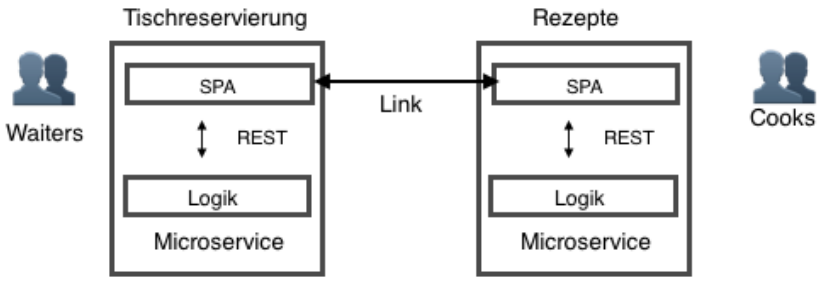

Abbildung 5.1.: SPA pro Microservice, erstellt von den Autoren basierend auf [Wolff, 2015, S.169].

von AngularJS 1 realisieren werden. Dies hat aber den Nachteil, dass die SPA nur als vollständige Anwendung deployt werden kann. Fehlt ein Modul ist die Gesamtanwendung funktionsunfähig.

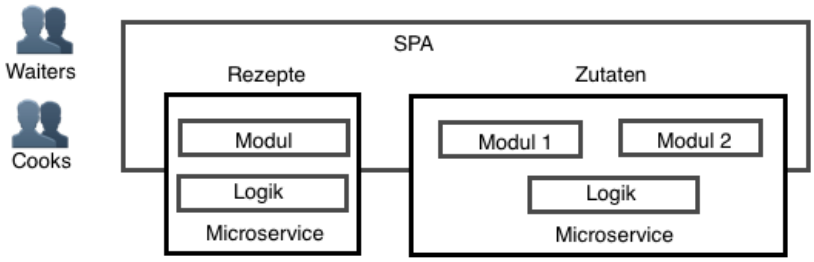

Abbildung 5.2.: Eine SPA für alle Microservices, erstellt von den Autoren basierend auf [Wolff, 2015, S.171].

Da der Rezepte- und der Bewertung Microservices eine engere Integration mit einander benötigen, ergab eine SPA-Lösung hier weniger Sinn als bei dem Profil Microservice (siehe dazu Abschnitt 5.2). Deswegen wurden auch weitere Umsetzungsmöglichkeiten wie Resource Oriented Client Architecture (ROCA) und Frontend-Server betrachtet.
Beim ROCA handelt es sich um User Interface (UI), die aus HTML-Seiten besteht und die Kopplung von Microservices durch Links gewährleistet. Damit sich die Seiten von der Gestaltung nicht unterscheiden, wird ein Asset-Server verwendet, der gemeinsame Stylesheets und JavaScripts beinhaltet. Für eine gemeinsame URL ist dann ein Router zuständig. Dadurch, dass diese Lösungsmöglichkeit nicht nur die Microservices-Prinzipien wie unabhängiges Deployment und unabhängige Entwicklung verletzt, sondern eine schwer einschätzbare Implementierungskomplexität eines Router mit sich bringt, wurde auf dieses Konzept verzichtet.
Der Kernmechanismus der Frontend-Server-Architektur bildet ein Frontend Server, der

in der Lage ist aus mehreren HTML-Snippets, die er von verschiedenen Microservices bekommt, eine HTML-Seite zu erstellen. Da dieser Ansatz eine potenziell einfache Lösungsmöglichkeit für das Einbetten von Microservices darstellt, wurde entschieden eine HTML-Snippets basierte Lösung für den Bewertung Microservice zu entwickeln. Mehr dazu kann im Abschnitt 5.3.

5.2. SPA für Profile Microservice

Der Profile-Microservice sollte nur dazu dienen, angemeldeten Benutzern die Möglichkeit zu geben, ihre Profildaten zu bearbeiten. Da diese Funktionalität relativ selten im Kontext der Restaurant-Anwendung benutzt wird, wurde entschieden, diese als eigenständigen Microservice mit User Interface (UI) zu implementiert. Dabei sollte er mit den anderen Microservices über einem Link gekoppelt werden. Eine SPA erschien eine gute Lösungsmöglichkeit für das User Interface (UI) zu sein, weil sich die Logik für das Inline-Editing, Speichern und die Validierung von den Eingabefeldern mittels AngularJS im Frontend leicht realisieren lässt.
Für das Erstellen einer neuen Anwendung bietet OASP4JS ein Application Template, das mit Hilfe von Yeoman Generator generiert werden kann. Im Vergleich zur OASP4JS Sample Application beinhaltet das Template keine zusätzlichen Module wie Security oder internationalization, sodass diese entweder selbständig entwickelt oder von der Sample Application übernommen werden müssen. Jedoch sind alle benötigten modernen Werkzeuge für Testing und Entwicklung mit AngularJS bereits installiert.
Ein Nachteil einer solchen Umsetzung war die Verletzung des einheitlichen Designs. Damit der Benutzer beim Umschalten zwischen Microservices keinen Unterschied feststellt, musste das HTML-Grundgerüst der SPA sowie ihre Styling-Dateien in die neue Anwendung übernommen werden. Bei einer kleinen Anzahl der Microservices ist dies kein großes Problem. Bei mehr als fünf Microservices, kann das Ändern vom Design aber zu einem Albtraum werden, sodass ein Assert-Server in Frage kommen kann, was einem unabhängigen Deployment gefährdet.

5.3. Einbettung von HTML - Snippets

Wie in den vorherigen Abschnitten beschrieben, eignet sich der SPA-Ansatz nicht mehr wenn eine engere Bindung zwischen User Interface (UI)-Elementen verschiedener Microservices, welche noch innerhalb einer Seite geeint werden sollen, besteht. Daher wurde im Rahmen des Projektes ein weiterer Ansatz verfolgt bei dem die einzelnen User Interface (UI)-Elemente in Form von HTML-Snippets ausgelagert werden. Diese HTML-Snippets sollen im Rahmen eines Services angeboten werden damit diese auch

von anderen Services verwendet werden können. Wenn ein neues Projekt mit OASP erzeugt wird[1], bieten sich zunächst zwei potentielle Anlaufstellen zur Implementation eines solchen Services:

Spring MVC Ein neu erstelltes Projekt in OASP verwendet Spring MVC zur Darstellung des Logins sowie einer Beispiel-Landingpage. Die OASP-Dokumentation geht jedoch bei der Beschreibung des Service-Layers nicht weiter darauf ein.

Apache CXF + JAX-RS Das Erstellen eines Representational State Transfer (REST)-Services auf Basis von Apache CXF und JAX-RS. Dies entspricht der Empfehlung von OASP[2] für Representational State Transfer (REST)-Services.

Da nicht unbedingt der Microservice selbst zu einer Webapplikation werden soll, sondern lediglich HTML-Snippets als Ressourcen nach außen getragen werden sollen, erscheint die Implementation als Representational State Transfer (REST)-Service angemessen. Auch OASP empfiehlt die Nutzung von Representational State Transfer (REST) für *internal services*, der Kommunikation zwischen verschiedenen Applikationen eines Anbieters[3].
Representational State Transfer (REST)-Services die sich an die JAX-RS Spezifikation halten verwenden Annotationen um eine Klasse mit ihren Methoden als Service zu beschreiben. Bisher wurden die Services meist nur für JSON-Nachrichten verwendet, kenntlich gemacht durch die *@Produces(...)*-Annotation. Da *plain old java objects* (POJO) eindeutig auf JSON-Objekte gemappt werden können, wird dies von OASP automatisch vorgenommen, sodass die Annotation bereits genügt (siehe Abb.5.3).
HTML-Snippets enthalten jedoch eine Gestaltungskomponente die sich keineswegs aus dem Java-Objekt ableiten lässt, weshalb hier kein automatisches Mapping stattfinden kann. Nachdem in der ersten explorativen Phase der Implementierung HTML-Strings händisch mit den relevanten Eigenschaften eines POJO (in diesem Fall einer Rezeptbewertung) konkateniert wurden, wurde schnell klar das dieses Vorgehen keine langfristige Lösung sein kann wenn ein gewisses Maß an Modifizierbarkeit und Wartbarkeit gegeben sein soll. Um diese beiden Punkte zu adressieren bietet sich der Einsatz einer Templating-Engine an. Die meisten Templating Engines zeichnen sich dadurch aus das Gestaltung und Inhalt von einander getrennt werden können. In der Regel wird dies durch den Einsatz von Platzhaltern realisiert. In der darauf folgenden Recherchephase wurden einige solcher Templating-Engines identifiziert, diese waren jedoch meistens für den Einsatz mit Spring gedacht, eine speziell für Apache CXF geeignete Engine wurde zunächst nicht gefunden. Letztendlich fiel die Wahl auf Apache Freemarker, da sich

[1] https://github.com/oasp/oasp4j/wiki/tutorial-newapp
[2] https://github.com/oasp/oasp4j/wiki/guide-rest#jax-rs
[3] https://github.com/oasp/oasp4j/wiki/guide-service-layer#types-of-services

```
@Path("/v1")
@Named("RatingRestService")
@Transactional
@Service
public class RatingRestServiceImplAngular2 komplette
   Neuentwicklung mit TypeScript und WebComponents {
    // [...]
    @Produces(MediaType.APPLICATION_JSON)
    @GET
    @Path("/ratings/")
    public List<RatingCto> getAllRatings(@Context
       HttpServletRequest request) {
        return this.ratingManagement.findRatings();
    }
    // [...]
}
```

Abbildung 5.3.: Definition einer Serviceklasse mittels Java-Annotationen nach JAX-RS Spezifikation.

diese losgelöst von anderen Technologien nutzen lässt. Zusätzlich wurde eine Wrapper-Klasse für Freemarker implementiert um die Nutzung der Templating-Engine nochmals deutlich zu vereinfachen. Freemarker erlaubt neben dem Einsatz von Platzhaltern für Variablen auch spezielle Direktiven um bei Änderungen am Template Redundanzen zu vermeiden. So erlaubt die *<#list>*-Direktive das Iterieren durch eine Liste von Objekten, sodass wiederkehrende Elemente nur einmalig gestaltet werden müssen (siehe Abb.5.4). Häufig benötigte Teilkomponenten können auch in ein eigenes Template überführt werden und mit der *<#include>*-Direktive wiederverwendet werden.

Während die Template-Dateien nun getrennt vom Code im Dateisystem des Projekts liegen ergab sich auch direkt eine neue Herausforderung: Während der Entwicklung wird der Ratingservice mittels SpringBoot gestartet, beim eigentlichen Deployment für die anderen Microservice-Teams jedoch als Tomcat-Modul, wodurch sich die Dateistruktur stark unterscheidet. Als SpringBoot-Anwendung bleibt die gewohnte Dateistruktur, in der sich auch der Quellcode bereits befindet, erhalten. Beim Tomcat-Modul werden die kompilierten Klassen jedoch in ein eigenes Package zusammengefasst, lediglich der *WEB-INF* und *META-INF* Ordner wird seperat ausgelagert. Hinweise in der Dokumentation, wie sich das Filehandling plattformübergreifend lösen lässt oder gar ein file abstraction layer innerhalb von OASP selbst, hätten an dieser Stelle einiges an Entwicklungszeit einsparen können. Um das Problem kurzfristig zu lösen, werden die

```
<h2 class='ratingHeader'>Recipe ${recipeId}:</h2>
<div>Average rating:</div>
<#include "/ratingStars.html">
<h3 class='commentTitle'>Comments:</h3>
<div class='comments'>
    <#list ratings as rating>
        <#include "/commentStars.html">
        <div class='comment'>
            ${rating.rating.comment}
        </div>
        <div class='thumbs'>
            <i class="fa fa-thumbs-o-up" aria-hidden="true"></i>${rating.upvotes}
            <i class="fa fa-thumbs-o-down" aria-hidden="true"></i>${rating.downvotes}
        </div>
    </#list>
</div>
```

Abbildung 5.4.: Ausschnitt eines Freemarker-Templates für ein HTML-Snippet

Templates nun in einem Unterordner des *WEB-INF* Ordners ausgelagert und zur Laufzeit überprüft ob der Tomcat-spezifische Pfad existiert, ist dies der Fall wird der Pfad beibehalten, ansonsten wird der SpringBoot-spezifische Pfad zum Template-Ordner verwendet. Hier konnte von der vorherigen Implementation der Wrapper-Klasse profitiert werden, sodass diese Überprüfung nur einmalig implementiert werden musste um diese Problematik für alle bisherigen und zukünftig verwendeten Templates zu verhindern.
Da einige HTML-Snippets neben der statischen Anzeige von Informationen auch Interaktionsmöglichkeiten bieten, wie zum Beispiel das Abgeben einer Bewertung, wurde neben HTML als Beschreibungssprache und CSS zur Gestaltung auch JavaScript zur Umsetzung von Funktionalität verwendet. Da es keinen zentralen Frontendserver gibt der die einzelnen HTML-Snippets zusammensetzt, muss dies im Frontend des User Interface (UI) geschehen, welches das jeweilige Snippet nutzt. Da JavaScript die einzige Programmiersprache ist, die ohne die Nutzung weiterer Browserplugins im Frontend ausgeführt werden kann, empfiehlt es sich diese auch zum Zusammensetzen der Snippets zu verwenden. Hierzu wird das Snippet vom Service abgerufen und dann in ein leeres, als Container fungierendes, HTML-Element des bestehenden DOM eingebettet. Um den Inhalt eines HTML-Elements zu füllen wird in JavaScript die Objekteigenschaft *.innerHTML* verwendet. Der aktuelle HTML5-Standard verhindert aus Sicherheitsgründen die automatische Ausführung von weiterem JavaScript-Code der in solch

einen Container nachgeladen wird[4]. Dies hat zur Folge das HTML-Snippets mit Interaktionselementen nicht mehr korrekt ausgeführt werden.
Um diese Limitation umgehen zu können wurde der Service neben der Auslieferung der Snippets, noch um die Auslieferung eines SnippetLoaders erweitert. Beim SnippetLoader handelt es sich um eine JavaScript-Klasse welche das Einbetten von Snippets für den jeweiligen Konsumenten übernimmt (siehe Abbildung 5.5).

```
1 <script type="application/javascript" src="[url-zum-service]/
    oasp-ms-rating/services/rest/v1/snippetloader.js"></script>
2 <script type="application/javascript">
3     new SnippetLoader(<Dom-Element>, <URL zum Snippet>);
4 </script>
```

Abbildung 5.5.: Verwendung des SnippetLoaders beim Konsumenten des Services

Der SnippetLoader filtert vor der Einbettung des Snippets in das Container-Element alle *<script>*-Elemente heraus und erzeugt mittels *document.createElement('script')* neue *<script>*-Elemente welche wieder mit dem Code des ursprünglichen Snippets befüllt werden und anschließend dem DOM mittels *document.body.appendChild()* hinzugefügt werden. Dies funktioniert da das benötigte JavaScript vom Konsumenten des Snippets komplett neu erzeugt wird und auf die Nutzung von *.innerHTML* für den JavaScript-Code verzichtet wird.

5.4. Frontend Microservices mit Web Components

Im Rahmen eines Spikes sollte untersucht werden, welche weiteren Möglichkeiten für die Integration des UI genutzt werden können. Bereits nach kurzer Recherche wurde ersichtlich, dass das bisher im Einsatz befindliche Framework AngularJS aus diversen Gründen denkbar ungeeignet für eine solchen Zweck ist (siehe Abschnitt 5.1. Gleichzeitig wurde erkannt, dass dessen, sich noch in der Entwicklung befindender, Nachfolger Angular 2 deutlich besser geeignet ist. Deshalb, und weil ein Umstieg auf Angular 2 in absehbarer Zeit sowieso bereits geplant war, wurde beschlossen den Spike anhand von Angular 2 durchzuführen anstatt von vornherein veralteten Code zu entwickeln.
Angular 2 unterscheidet sich von seinem Vorgänger grundlegend durch zwei Aspekte: der Verwendung der Programmiersprache TypeScript, sowie dem Aufbau der Anwendung mittels Web Components. Web Components sind neue Funktionen, die bisher teilweise zur HTML-Spezifikation hinzugefügt wurden und es erlauben sollen, wiederverwendbare Webkomponenten zu entwickeln. Angular 2 baut dabei vor allem auf dem

[4] https://www.w3.org/TR/2008/WD-html5-20080610/dom.html#innerhtml0

Konzept von Custom Elements auf das erlaubt, neue DOM-Elemente zu definieren die über den HTML-Standard hinausgehen. [W3C, 2016]
Für die Evaluation wurden prototypisch zwei Module entwickelt, siehe 5.6. Zum einen die Hauptapplikation, welche die Startseite, die Konfiguration und das Routing beinhaltet, und zum anderen ein Modul für die Anzeige von Rezepten.

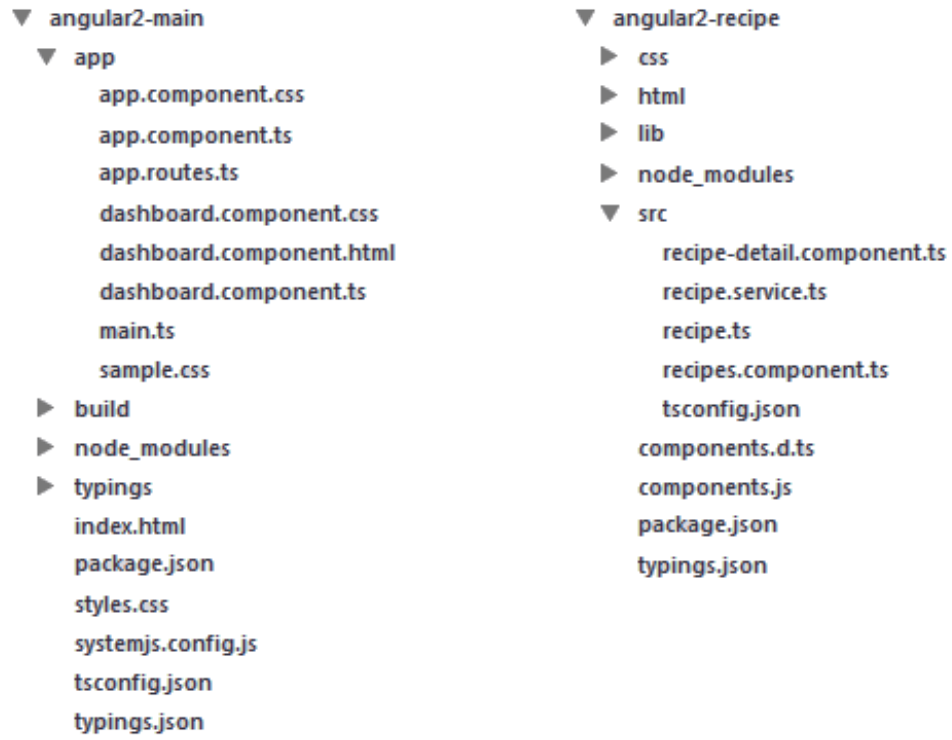

Abbildung 5.6.: Struktur der erstellten Module

Zum besseren Verständnis werden kurz die verschiedenen Teile der Applikation erklärt. Auf Seiten des Rezept-Moduls stellen die beiden Komponenten (recipe-detail.component.ts und recipes.component.ts) die Hauptbestandteile dar. Darin werden Darstellung und Logik der anzuzeigenden Seiten festgelegt. Die anzuzeigenden Rezepte, deren Struktur in recipe.ts modelliert ist, werden dabei aus dem Service recipe.service.ts bezogen, welcher die Verbindung zur bestehenden Datenbank realisiert. Um eine Einbindung und Nutzung des Moduls zu ermöglichen, werden die benötigten Zugriffspunkte in components.d.ts beziehungsweise components.js veröffentlicht, wie in Abbildung 5.7 zu sehen ist. Das Modul wird schließlich durch npm kompiliert, gepackt und exportiert.
Auf Seiten des Hauptmoduls wird das Rezept-Modul ebenfalls per npm als Dependency eingebunden. Dadurch können die beiden veröffentlichten Komponenten im Hauptmodul importiert werden. Welche Komponente wann verwendet werden soll, wird schließ-

```
export * from './lib/recipes.component';
export * from './lib/recipe-detail.component';
...
exports.RecipesComponent = require('./lib/recipes.component.
    js').RecipesComponent;
exports.RecipeDetailComponent = require('./lib/recipe-detail
    .component.js').RecipeDetailComponent;
```

Abbildung 5.7.: Inhalte von components.d.ts und components.js

lich im Routing in der Datei app.routes.ts bestimmt. Wie in Abbildung 5.8 zu sehen ist, wird dort festgelegt, dass die Pfade /recipes und /detail/id auf das Rezepte-Modul verweisen.
Auf diese Weise können beliebig viele Module eingebunden werden, deren Umfang beliebig groß sein kann. Geladen werden diese Module beim Ausführen von npm install, was üblicherweise vor dem Starten der Anwendung passiert. Ein getrenntes Deployment der einzelnen Module wird dadurch verhindert. Bei der Aktualisierung eines Moduls muss daher die ganze Anwendung neu gestartet werden. Trotz dieser Einschränkung können mit dieser Methode einzelne Teile des Frontendes vollkommen unabhängig voneinander entwickelt werden.
Es muss noch gesagt werden, dass Angular 2 zum Zeitpunkt des Spikes noch nicht final veröffentlicht wurde, sondern sich in der Version RC4 befand. Dementsprechend ist es möglich, dass sich einige Umstände die in diesem Abschnitt beschrieben wurden noch ändern. Es ist auch zu erwähnen, dass es neben Angular bereits einige weitere Frameworks gibt, die auf Web Components aufbauen und sich auf die einfache Modularisierung von Anwendungsteilen spezialisiert haben. Das prominenteste Beispiel ist hierbei Polymer.
Abschließend ist festzustellen, dass sich Web Components im allgemeinen und Angular 2 im speziellen durchaus für die Entwicklung von zentral verwalteten Frontend-Microservices eignen. Ist jedoch eine noch stärkere Unabhängigkeit zwischen den einzelnen Microservices gewünscht (wie beispielsweise Hot Deployment) muss auf andere Methoden ausgewichen werden.

5.5. Wertung

Befasst man sich mit der Integration von Microservices auf der Ebene der User Interface (UI), sieht man sich verschiedenen Dilemmas gegenüber gestellt. Beim Versuch die Vorteile der Microservices-Architektur beizubehalten, beeinträchtigt man oft die allgemeinen Prinzipien, wie einheitliches Design, unkomplizierter Aufbau der Logik oder

```
import { provideRouter, RouterConfig }  from '@angular/
    router';
import { DashboardComponent } from './dashboard.component';
import { RecipesComponent, RecipeDetailComponent } from '
    recipeMGMT/components.js';

const routes: RouterConfig = [
    {
      path: 'dashboard',
      component: DashboardComponent
    },
    {
    path: '',
    redirectTo: '/dashboard',
    terminal: true
  },
  {
    path: 'recipes',
    component: RecipesComponent
  },
  {
    path: 'detail/:id',
    component: RecipeDetailComponent
  },
];

export const APP_ROUTER_PROVIDERS = [
  provideRouter(routes)
];
```

Abbildung 5.8.: Inhalt von app.routes.ts

Freiheit von Redundanzen. Nicht zuletzt spielen bei der Integrationsentscheidung die im Frontend des Monolithen bereits verwendeten Technologien eine große Rolle.
Als nächstes wird versucht eine Wertung unter Berücksichtigung der oben genannten Vorteile, Prinzipien und Bedingungen zu geben. Da sich der Monolith in zwei Komponenten, dem AngularJS-basierten Frontend, sowie dem Java-basierten OASP4J aufteilt, wird die Heraustrennung dieser Komponenten und die anschließende User Interface (UI)-Integration entsprechend unterteilt.

5.5.1. OASP4JS

Dadurch, dass OASP im Frontend einen SPA-Ansatz verfolgt, gibt es starke Code Abhängigkeiten zwischen den Modulen der Anwendung. Nimmt man ein Modul aus dem Monolithen-Frontend raus und versucht ein neues mit der bestehenden SPA zu integrieren, müssen die JavaScript-Dateien mit den Anwendungs- und Modul-Definitionen entsprechend angepasst werden. Dies macht das unabhängige Deployment einzelner Microservices, die eine AngularJS 1 basierte SPA benutzen, unmöglich. Möchten Entwickler den Monolith mit einem unabhängigen Microservice ergänzen, kann er im Prinzip nur aus zwei Ansätzen wählen. Es wäre möglich eine eigene SPA für den Microservice zu schreiben und dabei das HTML-Grundgerüst und Styling-Dateien aus dem Monolithen zu übernehmen. Oder man kann komplett andere Technolgien benutzen, was die Gefahr mit sich bringt, dass sich der Interaktions-Fluss und das Design der neuen Microservices von dem Monolithen stark unterscheiden werden.
Zusammenfassend lässt sich sagen, dass die Aufteilung des Monolithen in die einzelne Module im Frontend an sich positiv bewertet wird. Dies erlaubt eine leichte Spaltung beziehungsweise Entfernung von Modulen wie „offer“, „sales“, „recipe“ oder „table“ Management. Die verwendete Technologie, also AngularJS 1, lässt den Microservice-Ansatz nur schwer, eventuell nur mit komplizierten Umwegen, die von unserem Team nicht entdeckt werden konnten, umsetzen.

5.5.2. OASP4J

OASP4J stellt das Java-basierte Backend zum AngularJS-basierten OASP4JS-Frontend dar. Da die Architektur des Monolithen bereits eine Modularisierung vorgesehen hat und somit bereits ein klarer Schnitt existiert, fiel es sehr leicht ein entsprechendes Modul herauszulösen und in einen neuen OASP Microservice zu überführen. Durch den Versionssprung von OASP mussten lediglich kleine Anpassungen durchgeführt werden. Nachdem der neue Microservice deployed ist, muss lediglich die neue URL vom Konsumenten des Backend, im Rahmen dieses Projekts war dies das monolithische OASP4JS Frontend, angepasst werden.
Auf der Serverseite wurde im Rahmen des Projekts eine sehr spezielle Art der User Interface (UI)-Integration angewandt, die Nutzung von HTML-Snippets. OASP liefert zwar Spring MVC mit, wodurch prinzipiell die Entwicklung von Webanwendungen möglich ist - jedoch soll der Backendservice beim Ansatz der HTML-Snippets nicht selbst die Rolle der gesamten Webanwendung, also auch des Frontends einnehmen. Stattdessen sind die HTML-Snippets nur als eine weitere Art von Ressource zu sehen, welche vom Representational State Transfer (REST)-Service zur Verfügung gestellt werden und OASP liefert für diesen speziellen Fall keine Hilfestellung.

Teil III.

Fazit

6. Fazit

Nachdem die vorherigen Kapitel gezielt Aspekte von Microservices-Architekturen betrachtet und bewertet haben, soll in diesem Kapitel ein übergreifendes Fazit gegeben werden. Dazu wird zuerst auf die Evaluation des Microservices-Architekturparadigma eingegangen. Der zweite Teil des Fazits beschäftigt sich anschließend mit der Eignung von OASP als Plattform für Microservices.

6.1. Einsatz von Microservices-Architekturen

Obwohl in diesem Projekt nicht alle Aspekte des Microservices-Architekturparadigmas behandelt werden konnten, haben die behandelten Aspekte einen guten Einblick in die Entwicklung einer Applikation nach diesem Paradigma gegeben. Zu Beginn des Projektes zeigte bereits die fachlicher Aufteilung einer bestehenden Anwendung, das die Identifikation einzelner Microservices keine triviale Aufgabe ist. Dies unterstreicht Aussagen wie die von [Fowler, 2015], welche empfehlen neue Projekte als Monolith zu beginnen und diesen später in Microservices zu unterteilen. Da in diesem Projekt weitere Argumente für diesen Ansatz, wie beispielsweise der hohe Aufwand für Infrastrukturarbeiten, nicht untersucht werden konnten, kann daher keine deutliche Empfehlung für dieses Vorgehen ausgesprochen werden. In einem Folgeprojekt könnte besonders der Aspekt der Infrastruktur (siehe Kapitel 4) näher betrachtet werden, um den Ansatz des „Monolith First“ zu bestätigen.
Einen weiteren guten Einblick lieferte die Auseinandersetzung mit dem Querschnittsaspekt der Authentifizierung und Autorisierung (siehe Kapitel 2). Hier wurde deutlich, das vor allem „Cross-Cutting Concerns“ wie Logging oder Sicherheit einen erheblichen Anstieg an Komplexität erfahren, wenn auf einen Microservices-Ansatz umgestellt wird. Diese Komplexität sollte bei der Abwägung über den Einsatz von Microservices nicht unterschätzt werden.
Neben diesen Schwierigkeiten, konnten aber auch Vorteile des Ansatzes festgestellt werden. Die Flexibilität, welche die Teams durch diesen Ansatz erhielten, ermöglichte es, losgelöst von den anderen Teams, Änderungen zu machen und neue Funktionalität schnell in eine produktionsähnliche Umgebung zu liefern. Hier können die Vorteile von Continuous Integration und Continuous Deployment (siehe Kapitel 3) noch stärker zur

Geltung kommen, als es bei monolithischen Anwendungen der Fall ist. Somit passt die Kombination des Microservices-Ansatzes mit Continuous Deployment hervorragend in eine agile Entwicklungsmethode wie Scrum, bei dem kleine Teams schnell Wert für einen Kunden generieren möchten.

Ein Problem, welches in diesem Projekt deutlich wurde, ist die Aufteilung der Benutzerschnittstelle und die Integration dieser in die entwickelten Microservices. Im vorliegenden Projekt wurde dieser Aspekt zwar behandelt, aber in seiner Komplexität unterschätzt. Hier ist es gegebenenfalls sinnvoll, ein dediziertes Team für die Zusammenführung der Benutzerschnittstellen-Fragmente abzustellen. Dies würde gewährleisten, dass eine kohärente Anwendung entwickelt wird, welche diese Kohärenz auch in der Benutzerschnittstelle widerspiegelt. Auch diese wäre ein Aspekt, der in Folgeprojekten stärker fokussiert werden sollte.
Abschließend ist festzuhalten, das sich der Microservices-Ansatz durch seine Vorteile zurecht großer Beliebtheit in der Industrie erfreut. Allerdings ist dieser Ansatz keine „Silver Bullet“. Bei der Abwägung über diesen Ansatz sollten auch die Nachteile betrachtet werden, welche je nach Art des Systems, sehr prominent sein können. Bei Systemen welche bereits eine hohe Komplexität aufweisen, kann der Umstieg auf eine Microservices-Architektur mit seiner Erhöhung der Komplexität in Querschnittsaspekten mehr Nachteile als Vorteile in das System tragen. Auch die Nutzung des Microservices-Ansatzes in „Greenfield“ - Projekten ist fraglich, da der Aufbau der benötigten Infrastruktur (zum Beispiel zur Kommunikation der Services oder zur Sicherung der Datenkonsistenz) viel Aufwand benötigt.

6.2. OASP als Plattform für Microservices-Architekturen

Neben der allgemeinen Eignung von Microservices-Architekturen, soll in diesem Abschnitt speziell die Eignung von OASP für diesen Architekturansatz beleuchtet werden. Diese Eignung konnte in diesem Projekt konkret durch die Aspekte der Authentifizierung und Autorisierung, sowie der User Interface (UI)-Integration untersucht werden. Für Querschnittsaspekte liefert OASP Standardlösungen, sodass bei der Enwticklung einer monolithischen Anwendung nicht sehr viel Aufwand in die Implementation einer Individuallösung investiert werden muss. Für den Einsatz in Microservices-Architekturen sind diese Standardlösungen allerdings nicht geeignet. Wie in Kapitel 2 beschrieben, musste daher vollständig auf eine Individuallösung zurückgegriffen werden, was einen erheblichen Mehraufwand bedeutete. Dieser Mehraufwand wurde mitigiert, da in OASP einige Vorkehrungen getroffen wurden, welche die Entwicklung von Individuallösungen leichter machen.

Im Bereich der User Interface (UI)-Integration wurden dahingegen mehr Schwächen festgestellt. OASP baut auf ein dediziertes Frontend, welches durch Aufrufe über das Netzwerk mit dem Backend kommuniziert. Dies entspricht dem Gegenteil des gewünschten Vorgehens in Microservices-Architekturen und benötigt sehr viel Anpassung um auf Microservices aufgeteilt zu werden. Vor allem die Nutzung von AngularJS in der Version 1.3 ist hier für einige Anwendungen negativ zu bewerten. Insgesamt unterscheidet sich der Anspruch von OASP jedoch stark vom Microservices-Ansatz. OASP versucht technologie- und implementationsunabhängig zu sein. Dies wird durch den hohen Abstraktionsgrad erzielt, welcher es ermöglicht, jederzeit einzelne Technologien auszutauschen. Microservices hingegen spezialisieren sich in der Regel auf einen Anwendungsfall oder eine einzelne Domäne. Statt eine einzelne Abhängigkeit oder Softwarekomponente auszutauschen wird bei größeren Änderungen einfach der gesamte Microservice ausgetauscht.
Unter diesen Gesichtspunkten ist eine Nutzung von OASP für die Entwicklung von Microservices zu empfehlen. Jedoch sollte berücksichtigt werden, dass andere Plattformen verstärkter den Microservices-Ansatz unterstützen (hier sei JHipster[1] als Beispiel zu nennen) und somit attraktiver sein können. Da OASP anstrebt, das Frontend auf Angular 2 umzustellen, wird zukünftig aber ein negativer Aspekt bei der Nutzung für Microservices entfallen (siehe Abschnitt 5.4 in Kapitel 5).

[1] https://jhipster.github.io/microservices-architecture/

Arbeitsmatrix

Legende

CB Christopher Beyerlein

MB Martin Breidenbach

SD Sebastian Domke

PD Pascal Dung

DF Daniel Felten

JF Jonas Frenz

DG Daria Grafova

AN Anatoli Neuberger

MO Mohammed Obaidi

CO Christian Olsson

CS Christoph Stephan

YY Yasa Yener

Kapitel	CB	MB	SD	PD	DF	JF	DG	AN	MO	CO	CS	YY
| Organisation des Projekts	0%	0%	0%	0%	0%	0%	0%	**20%**	0%	0%	**80%**	0%
| | - Ziel des Projektes	0%	0%	0%	0%	0%	0%	0%	**20%**	0%	0%	**80%**	0%
| | - Vorgehensweise	0%	0%	0%	0%	0%	0%	0%	**20%**	0%	0%	**80%**	0%
| | - Aufteilung in Teams	0%	0%	0%	0%	0%	0%	0%	**20%**	0%	0%	**80%**	0%
| | - Workshops für teamübergreifende Themen	0%	0%	0%	0%	0%	0%	0%	**20%**	0%	0%	**80%**	0%
| Microservice Architektur	0%	0%	0%	0%	0%	0%	**10%**	0%	0%	0%	**90%**	0%
| OASP	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	**100%**	0%
| Fachlicher Schnitt	0%	**53,6%**	0%	0%	**46,4%**	0%	0%	0%	0%	0%	0%	0%
| | - Stand des Monolithen	0%	0%	0%	0%	**100%**	0%	0%	0%	0%	0%	0%	0%
| | - Ansätze zur Aufteilung	0%	0%	0%	0%	**100%**	0%	0%	0%	0%	0%	0%	0%
| | - Ergebnisse	0%	**100%**	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
| | - Wertung	0%	**100%**	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
| Authentifizierung und Autorisierung	**48,5%**	0%	0%	0%	0%	**35,2%**	**16,3%**	0%	0%	0%	0%	0%
| | - A&A in Microservices-Architekturen	**42,6%**	0%	0%	0%	0%	**57,4%**	0%	0%	0%	0%	0%	0%
| | - A&A in OASP	**59%**	0%	0%	0%	0%	**30%**	**11%**	0%	0%	0%	0%	0%
| | - Wertung	**40%**	0%	0%	0%	0%	**23,3%**	**36,7%**	0%	0%	0%	0%	0%
| Continuous Integration und Continuous Deployment	0%	0%	**52,8%**	0%	0%	0%	0%	0%	**47,2%**	0%	0%	0%
| | - CI und CD im Allgemeinen	0%	0%	0%	0%	0%	0%	0%	0%	**100%**	0%	0%	0%
| | - Deployment mit Docker	0%	0%	**100%**	0%	0%	0%	0%	0%	0%	0%	0%	0%
| | - Deployment in die Cloud	0%	0%	**100%**	0%	0%	0%	0%	0%	0%	0%	0%	0%
| | - Wertung	0%	0%	**100%**	0%	0%	0%	0%	0%	0%	0%	0%	0%
| Inter-Service Kommunikation	0%	0%	0%	0%	0%	0%	0%	**100%**	0%	0%	0%	0%
| | - API-Spezifikation	0%	0%	0%	0%	0%	0%	0%	**100%**	0%	0%	0%	0%
| | - Übertragungsformate	0%	0%	0%	0%	0%	0%	0%	**100%**	0%	0%	0%	0%
| | - Datenkonsistenz	0%	0%	0%	0%	0%	0%	0%	**100%**	0%	0%	0%	0%
| User Interface - Integration	0%	0%	0%	0%	0%	0%	**37,4%**	0%	0%	**25,2%**	0%	**37,4%**
| | - Umsetzungsmöglichkeiten	0%	0%	0%	0%	0%	0%	**100%**	0%	0%	0%	0%	0%
| | - SPA für Profile Microservice	0%	0%	0%	0%	0%	0%	**100%**	0%	0%	0%	0%	0%
| | - Einbettung von HTML - Snippets	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	**100%**
| | - Frontend Microservices mit Web Components	0%	0%	0%	0%	0%	0%	0%	0%	0%	**100%**	0%	0%
| | - Wertung	0%	0%	0%	0%	0%	0%	**65%**	0%	0%	0%	0%	**35%**
| Fazit	0%	0%	0%	**88,4%**	0%	0%	0%	0%	0%	0%	0%	**11,6%**
| | - Einsatz von Microservices-Architekturen	0%	0%	0%	**100%**	0%	0%	0%	0%	0%	0%	0%	0%
| | - OASP als Plattform für Microservices-Architekturen	0%	0%	0%	**72,2%**	0%	0%	0%	0%	0%	0%	0%	**27,8%**
| Glossar	**4,5%**	**4,5%**	**4,5%**	**50%**	**4,5%**	**4,5%**	**4,5%**	**4,5%**	**4,5%**	**4,5%**	**4,5%**	**4,5%**
| Redaktion/Editoring	0%	0%	0%	**100%**	0%	0%	0%	0%	0%	0%	0%	0%

Index

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Amazon 2016a] Amazon, Amazon Web S.: *Docker Basics*. http://docs.aws.amazon.com/AmazonECS/latest/developerguide/docker-basics.html. Version: 2016. – Eingesehen am 23.08.2016

[Amazon 2016b] Amazon, Amazon Web S.: *Getting Started with Amazon EC2 Linux Instances*. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EC2_GetStarted.html. Version: 2016. – Eingesehen am 23.08.2016

[Cygan 2015] Cygan, Bernhard: *Von Continuous Integration zu Continuous Delivery mit Jenkins Workflow*. https://www.informatik-aktuell.de/entwicklung/methoden/von-continuous-integration-zu-continuous-delivery-mit-jenkins-workflow.html. Version: 2015

[Docker 2016] Docker: *Docker for Windows*. https://docs.docker.com/engine/getstarted/step_one/#/docker-for-windows. Version: 2016. – Eingesehen am 23.08.2016

[ForgeRock 2016] ForgeRock: *OpenAM Developer's Guide*. https://backstage.forgerock.com/#!/docs/openam/13.5/dev-guide#chap-api-overview. Version: Juli 2016. – Eingesehen am 09.08.2016

[Fowler 2015] Fowler, Martin: *Monolith First*. http://martinfowler.com/bliki/MonolithFirst.html. Version: 2015

[Newman 2015] Newman, Sam: *Microservices - Konzeption und Design*. 1. Aufl. Heidelberg : MITP-Verlags GmbH & Co. KG, 2015. – ISBN 978–3–95845–081–3

[Oracle 2010] Oracle: *Sun OpenSSO Enterprise 8.0 Deployment Planning Guide*. https://docs.oracle.com/cd/E19575-01/820-3746/gjbna/index.html. Version: 2010. – Eingesehen am 21.08.2016

[Pressler u. Tigges 2015] Pressler, Jerry ; Tigges, Oliver: *Docker – perfekte Verpackung von Microservices*. http://www.sigs-datacom.de/uploads/tx_mwjournals/pdf/preissler_tigges_OTS_Architekturen_15.pdf. Version: 2015. – Eingesehen am 23.08.2016

[Richardson 2016] Richardson, Chris: *Refactoring a Monolith into Microservices*. https://www.nginx.com/blog/refactoring-a-monolith-into-microservices/. Version: 2016

[Tinatigertech 2015] Tinatigertech: *Ein Docker-Container und deine Anwendung läuft überall.* http://www.tigertech.de/ein-docker-container-und-deine-anwendung-laeuft-ueberall/. Version: 2015. – Eingesehen am 23.08.2016

[W3C 2016] W3C: *Custom Elements.* https://www.w3.org/TR/custom-elements/. Version: Juli 2016

[Ward 2016] Ward, Chris: *The Shortlist of Docker Hosting.* https://blog.codeship.com/the-shortlist-of-docker-hosting/. Version: 2016. – Eingesehen am 23.08.2016

[Wolff 2015] Wolff, Eberhard: *Microservices - Grundlagen flexibler Softwarearchitekturen.* 1. Auflage. dpunkt.verlag GmbH, 2015

Glossar

Bounded Context

Definiert die Grenzen eines Kontexts, sodass ersichtlich ist, wann ein Kontext noch gilt und wann er seine Gültigkeit verliert. Jeder Bounded Context kann ein eigenes Domänenmodell besitzen. Bounded Context ist ein zentrales Pattern des Domain-Driven-Designs.

Continuous Deployment

Ansatz aus der Softwareentwicklung bei dem Methoden und Prozesse von agilen Vorgehensweisen mit der Produktion (IT-Betrieb) verknüpft werden. Somit besteht die Möglichkeit Software in einer hohen Geschwindigkeit iterativ zu entwickeln und bereits während der Entwicklung einsatzfähige Software zu liefern. Zu diesem Zweck werden hierbei möglichst alle Prozessschritte von der Entwicklung (Build und Tests) bis hin zum Deployment hochgradig automatisiert.

Continuous Integration

Beschreibt einen Prozess im Rahmen der Softwareentwicklung, bei dem Änderungen am Projekt kontinuierlich in das Gesamtsystem integriert werden. Die veränderten Komponenten werden hierbei mit den restlichen Bestandteilen zu einem lauffähigen Gesamtsystem zusammengefügt. Zur Unterstützung des Continuous Integration-Prozesses werden für die Verwaltung des Source Codes Version Control Systeme und für die Automatisierung der Build- und Testvorgänge entsprechende Werkzeuge eingesetzt.

Cookie

Ein Cookie ist eine Textinformation, die die besuchte Website (hier „Server“) über den Browser im Rechner des Betrachters („Client“) platziert. Der Cookie wird entweder vom Webserver an den Browser gesendet oder von einem Skript (etwa JavaScript) in der Website erzeugt. Der Client sendet die Cookie-Information bei späteren, neuen Besuchen dieser Seite mit jeder Anforderung wieder an den Server.

Deployment Descriptor

Ein Deployment Descriptor ist eine Konfigurationsdatei im Format XML. Im Umfeld der Java Platform, Enterprise Edition, beschreibt diese Konfigurationsdatei den spezifischen Bereitstellungsprozess und dazu benötigte Informationen. In Java-Webanwendungen muss der Deployment Descriptor den Namen web.xml tragen und sich im Unterverzeichnis WEB-INF zum Wurzelverzeichnis der Webanwendung befinden.

DevOps

DevOps (Development und IT Operations) kann als Mentalität oder Kultur beschrieben werden und dient zur vollständigen Automatisierung des Entwicklungsprozesses unter Zuhilfenahme von Technologien und Werkzeugen. Setzt eine starke Kommunikation und Bindung zwischen der Entwicklung und dem Betrieb voraus und bedarf bei der Einführung einer Neustrukturierung der vorhanden Strukturen und die Bereitschaft neue Methoden, Prozessen, Plattformen und Werkzeugen zu erlernen.

DOM

Das Document Object Model (DOM) ist eine plattformunabhängige Schnittstelle, die es Programmen und Skripten erlaubt, dynamisch auf den Inhalt, die Struktur und den Stil von Dokumenten zuzugreifen und diese zu bearbeiten.

Domain-Driven-Designs (DDD)

Eine Vorgehensweise um eine Software nach fachlichen Aspekten zu modellieren. DDD unterteilt ein großes System in eine Reihe von Bounded Contexts.

HTML

Die Hypertext Markup Language erlaubt die Beschreibung der Struktur einer einzelnen Webseite.

HTML-Snippet

Eine Teilmenge einer an den Browser gesendeten Webapplikation. Kann neben dem HTML-Gerüst auch weitere Webtechnologien wie CSS-Anweisungen und JavaScript-Code enthalten.

JSR250

Der Java Specification Request 250 legt fest wie Annotations in Java definiert und verwendet werden.

LDAP

Das Lightweight Directory Access Protocol ist ein weit verbreitetes Netzwerkprotokoll zum Verwalten verteilter Verzeichnisdienste.

npm

Der Node Package Manager (npm) ist ein Paketmanager für die JavaScript-Laufzeitumgebung node.js.

OASP

Die Open Application Standard Plattform (OASP) soll eine Lösung bieten, Anwendungen aufzubauen, die sowohl Best-in-Class-Frameworks und Bibliotheken, als auch in Industrie bewährte Praktiken und Code-Konventionen kombinieren.

Representational State Transfer (REST)

REST bezeichnet ein Programmierparadigma für verteilte Systeme, insbesondere für Webservices.

Reverse Proxy

Ein Proxy tritt im Falle des Reverse Proxys als vermeintliches Zielsystem in Erscheinung, wobei die Adressumsetzung dann in der entgegengesetzten Richtung vorgenommen wird und so dem Client die wahre Adresse des Zielsystems verborgen bleibt.

Session Hijacking

Session Hijacking ist ein Angriff auf eine verbindungsbehaftete Datenkommunikation zwischen zwei Computern. Authentifiziert sich einer der Kommunikationspartner gegenüber dem anderen innerhalb der Sitzung, stellt diese eine Vertrauensstellung dar. Ziel des Angreifers ist es, durch die „Entführung“ dieser Sitzung die Vertrauensstellung auszunutzen, um dieselben Privilegien wie der rechtmäßig authentifizierte Benutzer zu erlangen.

Single-Sign-On

Beschreibt das Prinzip, dass sich innerhalb eines Microservice-Verbunds nur ein einziges mal Authentifiziert werden muss, was mit einer zum Teil erheblich besseren User Experience einher geht.

User Interface (UI)

Die Schnittstellen, mit denen ein Nutzer mit einem System interagieren kann. Als GUI (Graphical User Interface) wird eine grafische Benutzerschnittstelle bezeichnet.