Hochschule Wismar
University of Applied Sciences
Technology, Business and Design
Fakultät für Ingenieurwissenschaften, Bereich EuI

Bachelorarbeit

Nutzung von Self-Contained Systems in der elektronischen
Antragstellung des MV-Serviceportals

von: Patrice Matz
geboren am 17.07.1997
in Neubrandenburg

Betreuer: Prof. Dr.-Ing. Matthias Kreuseler
betrieblicher Betreuer: Jörg Birkholz

Zusammenfassung

Das Online Zugangsgesetz (OZG) wurde im Jahr 2017 verabschiedet und hat zum Ziel über 500 Verwaltungsleistungen pro Bundesland bis zum Jahr 2022 Bürgern digital bereitzustellen. Das DVZ Mecklenburg-Vorpommern wurde mit der Implementierung einer OZG-konformen digitalen Plattform beauftragt. Diese Plattform ist das MV-Serviceportal (MVSP). In dieser Arbeit wird eine Makroarchitektur zur Einbindung von Self-Contained-System (SCS) in das MVSP entworfen und eine prototypische Implementierung vorgestellt. Diese Architektur ist lose gekoppelt und überträgt Daten nur verschlüsselt. Zusätzlich erfolgt eine symmetrische Authentifizierung zwischen allen Komponenten. Um dieses Ziel unter Einhaltung der Anforderungen zu erreichen wird ein Service-Mesh-Ansatz genutzt. Es werden zwei Reverse-Proxys untersucht: Apache HTTPD und NGINX. Ebenso werden Client-Side-Rendering und Server-Side-Rendering zur Einbindung der jeweiligen User-Interfaces untersucht. In einer Reihe von Tests zu Sicherheit, Ladezeiten und Latenzen durch die Verwendung von Sidecars und symmetrischer Authentifizierung zeigen Entwurf und prototypsche Implementierung ihre Eignung für den Einsatzzweck.

Abstract

The online accessibility act was written into law in 2017. It has the goal of digitizing over 500 administrative services by the year 2022. The DVZ Mecklenburg-Vorpommern was tasked with creating a compliant platform. This platform is the ”MV-Serviceportal”. The goal of this project was design a macroarchitecture which would allow for the integration of SCS into the MVSP and to implement a prototyp. The architecure is loosely coupled and only transfers encrypted data over the network. Additionally a symmetrical authentication of the components was implemented. A service-mesh approach is used to achieve this goal while adhering to requirements. For the implementation two reverse proxies: Apache HTTPD and NGINX as well as client side rendering and server side rendering are evaluated. Finally a series of test regarding the security, loading times and latency introduced by the sidecars and the symmetrical authentication show the suitability of the implemented design.

Inhaltsverzeichnis

1 Einleitung **5**
1.1 Problemstellung 5
1.2 Lösungsansatz 5

2 Fachliches Umfeld **7**
2.1 Onlinezugangsgesetz 7
2.2 Ist-Stand 7

3 Anforderungen **10**
3.1 Anforderungen an die Architektur 10
3.2 Anforderungen an Dienste 11

4 Grundlagen **12**
4.1 Architekturen im Überblick 12
4.1.1 Monolith 13
4.1.2 Service-Oriented Architecture 13
4.1.3 Microservices 14
4.1.4 Self-Contained-Systems 14
4.2 RESTful API 15
4.2.1 Pfade 15
4.2.2 HTTP-Methoden 16
4.3 Verteilte Architekturen 17
4.3.1 Microservices 17
4.3.2 Service-Mesh 18
4.3.3 API-Gateway 18
4.4 Docker 19
4.5 Kubernetes 20
4.6 Sicherheitsaspekte 21
4.6.1 Anfrageauthentifizierung 21
4.6.2 Service-Authentifizierung 22
4.6.3 Weitere Sicherheitsüberlegungen 23

5 Entwurf **24**
5.1 Architektur 24
5.2 API 26
5.2.1 MVSP API 26
5.2.2 Dienst-API 28
5.3 Abläufe 29
5.3.1 Erstellen 29
5.3.2 Lesen und Ändern 30

5.3.3 Löschen 32

6 Voruntersuchung 33
6.1 Docker 33
6.2 Test verschiedener UI-Fragment-Composition Konzepte 33
6.3 Interaktion zwischen Client und Servern 37
6.4 Authentifizierung 39
6.5 Auswertung 42

7 Implementierung 43
7.1 Online-Dienst-Prototyp 43
7.1.1 OD-API 44
7.2 eAST 45
7.2.1 UI-Fragment Composition 46
7.2.2 eAST-API 47
7.3 Deployment 49

8 Tests 50
8.1 Erfüllung der Anforderungen 50
8.2 Abgrenzungstest 51
8.3 Ladezeiten und Latenz 51
8.4 Latenz durch Zertifikate 53
8.5 Auswertung 54

9 Fazit und Ausblick 55

Literaturverzeichnis 56

Abbildungsverzeichnis 60

Listings 61

Tabellenverzeichnis 62

A Weitere Listings 65

Selbstständigkeitserklärung 72

1 Einleitung

Die Digitalisierung stellt Deutschland vor neue Herausforderungen. Diese sind vielfältig in Art und Ausprägung. Durch den Breitbandausbau und die weiter steigende Verbreitung von Smartphones sind so viele Bürger online wie nie zuvor [1]. Die Landesregierungen haben die hier entstehenden Potentiale erkannt. Zur Nutzung dieser wurde im Jahr 2017 das OZG verabschiedet [2].

1.1 Problemstellung

Das OZG soll Bürgern die Nutzung von Verwaltungsleistungen zugänglicher machen. Um das Entstehen von verschiedenen und parallelen Einzellösungen im Land MV zu vermeiden, wurde das MVSP entwickelt. Das MVSP stellt einen Teil der IT-Infrastruktur des Landes MV dar, welcher es erlaubt verschiedenen Stellen der öffentlichen Verwaltung einen zentralen elektronischen Antragsassistenten bereitzustellen.

In seiner bisherigen Form ist das MVSP in seinem Funktionsumfang durch die Nutzung von Formularen eingeschränkt. Hier soll eine neue SCS-basierte Architektur Abhilfe schaffen.

1.2 Lösungsansatz

Zu den Charakteristika von SCS gehört, dass sie meist Dienste[1] mit genau einer Aufgabe sind. Jedes SCS enthält die gesamte Fachlogik, Daten und User Interface (UI), die zur Erfüllung ihrer Aufgabe nötig ist. Ein SCS wird von einem Team betreut, welches für alle Teile des Dienstes vom Entwurf bis zum Deployment verantwortlich ist. Jeder Dienst kann vollkommen andere Technologien verwenden, da die Dienste untereinander nur über ihre APIs kommunizieren. [3]

[1] Dienste (englisch Service oder Web-Service) sind Teile eines oder ganze Web-Services.

Mehrere SCS können zusammen einen größeren Dienst bilden. Die einzelnen Dienste sind lose gekoppelt und möglichst unabhängig voneinander. Dadurch entsteht ein großer heterogener Dienst, wobei die beteiligten Teams ihre Autonomität behalten. Dies bietet einige allgemeine Vorteile: Dienste können einander nicht negativ beeinflussen, Teams bleiben autonom und effizient, zudem betreffen Bugs einer bestimmten Technologie nur einige Dienste und nicht alle.

Bei einem föderalen Projekt wie der Umsetzung des OZG scheinen SCS passend. Jedes Portal könnte aus einer Menge von SCS bestehen, welche jeweils genau ein Verfahren implementieren. Analog hierzu könnte jedes Portal innerhalb eines Portalverbundes ebenfalls als SCS angesehen werden. Dies könnte dafür sorgen, dass eine Vielzahl von Entwicklerteams parallel an einem Portal arbeiten könnten, wobei jedes Team autonom und unbeeinflusst von anderen Teams arbeiten kann.

Diese grundsätzliche Idee wird in den folgenden Kapiteln auf Ihre Anwendbarkeit innerhalb des MVSP untersucht.

2 Fachliches Umfeld

Dem MVSP liegt das OZG zu Grunde. Das OZG hat das Ziel, Verwaltungsleistungen digital bereitzustellen. Somit können Prozesse optimiert und Bürgern und Wirtschaft Zeit und Geld gespart werden.

In diesem Kapitel wird zuerst der Inhalt des Gesetzes zusammengefasst. Anschließend wird der daraus resultierende Ist-Stand erläutert.

2.1 Onlinezugangsgesetz

Das OZG hat das Ziel bis zum Jahr 2022 über 500 Verwaltungsleistungen zu digitalisieren. Es sind barrierefreie Portale zu schaffen und mit den Portalen anderer Länder zu einem Portalverbund zusammenzuschließen. Jedes Portal muss über ein Nutzerkonto verfügen, welches der eindeutigen Identifikation der Nutzer und dem Speichern relevanter personenbezogener Daten dienen soll. Genaueres hierzu ist unter §8 [2] nachzulesen. Die Nutzung des Nutzerkontos ist für den Nutzer optional. Alternativ können Leistungen auch ohne Account beantragt werden. Das Durchführen der Digitalisierung, das Schaffen neuer und das Ändern bestehender Prozesse obliegt den Ländern. [2]

Für das Portal des Landes MV wurde das Datenverarbeitungszentrum Mecklenburg-Vorpommern (DVZ) mit der Schaffung einer Plattform nach den oben beschriebenen Vorgaben vom Ministerium für Energie, Infrastruktur und Digitalisierung beauftragt.

2.2 Ist-Stand

Der aktuelle Stand entspricht einer Service-Orientated-Architecture. Das MVSP siehe Abb. 2.1 besteht aus drei, von der OZG-Referenzarchitektur [4] vorgeschrieben, Kernkomponenten: dem Nutzerkonto, dem Content-Management-System (CMS) und der elektronischen Antragstellung (eAST). Des Weiteren werden die Infodienste M-V

(IDMV), das Formular-Management-System (FMS), die Zahlungsverkehrsplattform (ZVP) und die Virtuelle Poststelle (VPS) benötigt.

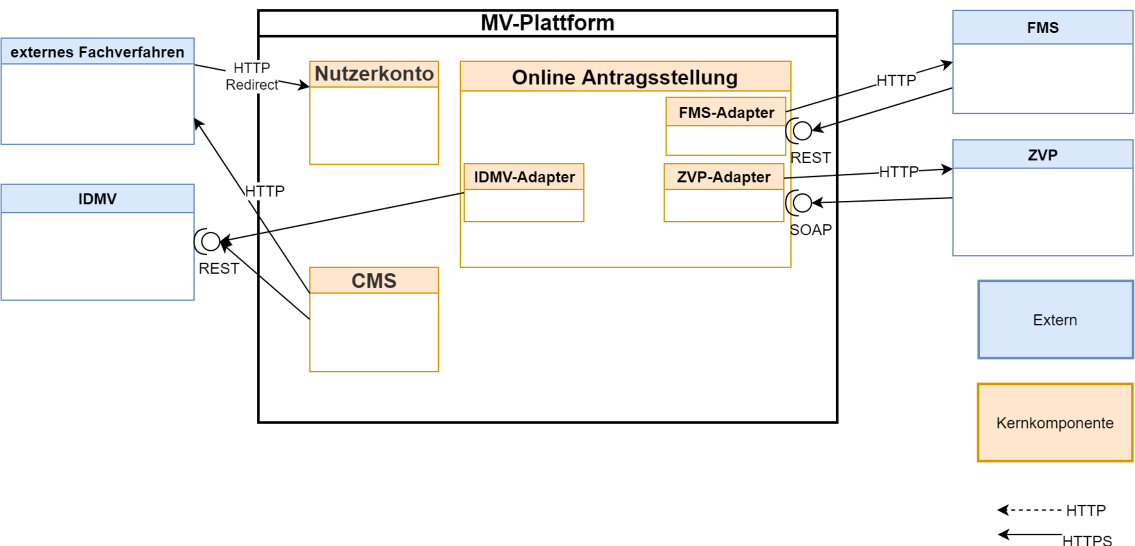

Bild 2.1: Ist-Architektur

Kernkomponenten:

Nutzerkonto Das Nutzerkonto dient der Authentifizierung des Nutzers unter anderem mit der eID[5]. Im Nutzerkonto werden sämtliche personenbezogenen Daten gespeichert, z. B. Name oder Adresse. Der Login der Nutzer erfolgt per OpenID Connect[1] [4]

Content Management System Das CMS dient, aus Sicht der Elektronischer Antragstellung (eAST), dem Suchen und Finden der zuständigen Stelle für die gewünschte Leistung im gewünschten Ort. Diese Stelle ist eine Behörde und kann einen Online-Antragsassistenten[2] bereitstellen. Das CMS ist der Einstiegspunkt für die eAST.

Elektronische Antragstellung Die elektronische Antragstellung wird als eAST abgekürzt und stellt dem Nutzer, die zur Leistung und Wohnort passenden, Antragsprozess bereit. Der Nutzer kann hier neue Leistungen beantragen, den Status beantragter Leistungen einsehen und sich über weitere Leistungen informieren. Über das Postfach wird der Nutzer über Änderungen zu seinen Anträgen informiert.

[1] OpenID Connect ist eine Authentifizierungsschicht, welche auf dem Autorisierungsprotokoll OAuth 2.0 aufbaut. Hiermit ist es möglich die Identität eines Nutzers zu prüfen und Daten über diesen Nutzer oder die Session ab zu rufen. [6]

[2] Im fachlichen Umfeld werden Online-Antragsassitenten als Online-Dienste bezeichnet, um Verwirrung zu vermeiden wird im Folgenden der Begriff Online-Antragsassistent verwendet

Externe Schnittstellen

Adaptoren Da die drei Kernkomponenten Daten von verschiedenen externen Diensten benötigen, haben sie Abhängigkeiten zu externen Schnittstellen dieser Dienste und Adapter zur Nutzung dieser.

IDMV Die Infodienste M-V sind die zentralen Informations- und Pflegesysteme, des Landes MV mit Informationen zu Orten, Zuständigen Stellen und den angebotenen Online-Antragsassistenten.

FMS Das Formular-Management-System stellt die nötigen Formulare bereit. Diese können in verschiedenen Formaten, wie HTML oder PDF vorliegen. Welche Formulare benötigt werden ist im IDMV hinterlegt.

ZVP Die Zahlungsverkehrsplattform dient dem sicheren Abwickeln von Zahlungsvorgängen für E-Government-Leistungen [7].

3 Anforderungen

In diesem Kapitel werden die technischen und fachlichen Anforderungen an mögliche SCS-basierte Lösungsentwürfe definiert. Diese wurden im Laufe eines mehrstufigen iterativen Reviewprozesses in Zusammenarbeit mit den Entwicklern und Architekten des MVSP gefunden. Es wird unterschieden zwischen Anforderungen an den Entwurf der Makroarchitektur und den Anforderungen an die verwendeten Dienste.

3.1 Anforderungen an die Architektur

Aufgrund des fachlichen Umfeldes gibt es allgemeine Anforderungen an den Entwurf, die erfüllt werden müssen. In diesem Unterkapitel werden diese erläutert.

Lose Kopplung In dieser Arbeit soll eine Erweiterung der Funktionalität der eAST untersucht werden. Diese Erweiterung darf die Stabilität des MVSP nicht beeinträchtigen. Es ist daher schon im Entwurf sicherzustellen, dass eingebettete Dienste im Fehlerfall keine weiteren Dienste negativ beeinflussen. Es gilt, daher Abhängigkeiten zu minimieren und optional zu gestalten.

Transport Layer Security (TLS) Die Sicherheit von Bürgerdaten hat eine sehr hohe Priorität. Das Rechenzentrum des DVZ ist nach ISO 27001 bis zum Datenbanklevel zertifiziert. Im unwahrscheinlichen Fall eines erfolgreichen Eindringens in das Netzwerk des DVZ darf die Sicherheit von Nutzerdaten dennoch nicht beeinträchtigt werden. Daher gilt es, Daten nur über verschlüsselte Verbindungen zu übertragen.

Service-Authentifizierung Eine weitere Maßnahme zur Absicherung von Nutzerdaten ist die Service-zu-Service Authentifizierung. Durch die Verwendung von HTTPS kann der Client die Identität des Servers validieren. Der Server kann aber keine Aussage über die Vertrauenswürdigkeit des Clients machen. Daher ist auch eine Client-Authentifizierung nötig. Die Client- und Server-Rollen in diesem Beispiel können beide beteiligten Services annehmen, abhängig davon welcher Dienst die Anfrage initiiert hat.

Nutzung vorhandener Dienste Das DVZ entwickelt und betreibt Softwareprodukte, allerdings nimmt es meist keine Pflege von Daten in diesen Diensten vor. Diese Daten werden von anderen Stellen gepflegt. Um den Arbeitsaufwand zu minimieren und Datenkonsistenz sicherzustellen, gilt es vorhandene Dienste so weit wie möglich weiter zu nutzen z. B. die IDMV. Siehe Kapitel 2.2, externe Schnittstellen.

Technologiematrix Im DVZ existiert eine Technologiematrix, welche eine Reihe von untersuchten und beurteilten Technologien beinhaltet. Für eine Reihe von Kategorien gibt es zugelassene, beobachtete und geduldete Technologien. Dies vereinfacht das Patch-Management.

Da der in dieser Arbeit erarbeitete Entwurf produktiv einsatzbar sein soll, dürfen ausschließlich in dieser Matrix enthaltene Technologien genutzt werden. Sollte eine Technologie für gut befunden werden, aber nicht in der Technologiematrix enthalten sein, kann diese in die Technologiematrix aufgenommen werden.

OZG-konform Das OZG schreibt eine Referenzarchitektur vor, diese muss weiterhin implementierbar sein.

3.2 Anforderungen an Dienste

Durch die Verwendung von Self-Contained-Systems ergeben sich allgemeine Anforderungen an die Schnittstellen, die ein Dienst bereitstellen muss. Die eAST ist innerhalb des Deployments einzigartig, wohingegen eine Vielzahl von Online-Diensten implementierbar sein sollen. Daher ist es sinnvoll jedem Online-Dienst eine Minimal-API vorzuschreiben, welche implementiert werden muss. Somit kann die eAST mit allen Online-Diensten einheitlich interagieren.

Zusätzlich zu dieser Minimal-API muss ein Online-Dienst Routen zum Beziehen der HTML-Fragmente, Stylesheets und JavaScript-Dateien bereitstellen. Die eingebetteten Dienste sollten nicht auf einen bestimmten Kontext bestehen. Für eine einfachere Konfiguration sollte der Kontext dem Root-Kontext[1] entsprechen.

[1] Der Kontext ist ein Pfad-Präfix [8][9]. Beispielsweise handelt es sich bei Routen mit dem „/od/"-Präfix um Routen im „od"-Kontext. Wohingegen „/" dem Root-Kontext entspricht.

4 Grundlagen

Im diesem Kapitel werden Grundlagen erläutert, welche für das Verständnis der darauf aufbauenden Kapitel nötig sind. Es wird keine Abwägung vorgenommen, diese folgt später. Es wird mit einem kurzen Überblick verteilter und nicht verteilter Architekturen begonnen. Diese Übersicht kann auch als Evolutionsgeschichte von Self-Contained-Systems verstanden werden. Anschließend werden Aufbau und Charakteristika von RESTful APIs erklärt. Des Weiteren werden Docker und Kubernetes mit den zugrundeliegenden Konzepten erläutert. Abschließend werden relevante Sicherheitsaspekte vorgestellt.

4.1 Architekturen im Überblick

Im Folgenden Unterkapitel werden die in dieser Arbeit relevanten Architekturen vorgestellt. Zunächst werden die traditionellen Architekturkonzepte wie Monolithen und Service-Oriented-Architecture (SOA) und anschließend die neueren Ansätze Microservices und Self-Contained-Systems betrachtet. Die Vorteile werden entsprechend des Einsatzzweckes hervorgehoben.

Die Grafik 4.1 dient der Visualisierung der Struktur verschiedener Architekturen.

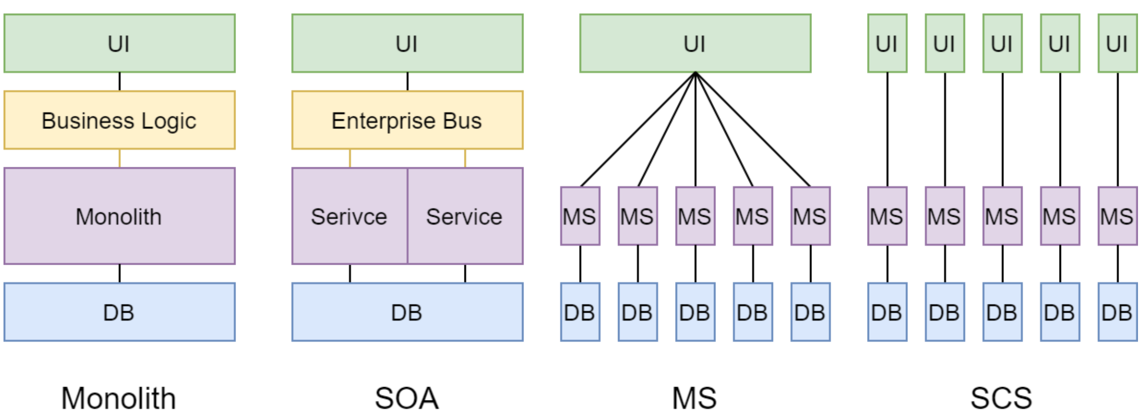

Bild 4.1: Gegenüberstellung verschiedener Architekturen in Anlehnung an [10]

4.1.1 Monolith

Die älteste Architektur ist der Monolith. Monolithen zeichnen sich durch eine hohe Unabhängigkeit zu anderen Diensten ab. Sie sind nicht immer in abgrenzbare Module unterteilt, oft sind sie stark an Hardware, bestimmte Produkte oder Betriebssysteme gebunden. Ihren Ursprung nahmen sie bei den Mainframes; leistungsstarke Systeme, die oft nur ein einzelnes Programm ausführen.

Monolithische Programme besitzen einige Vorteile: sie sind oft leicht zu deployen, unabhängig von anderen Diensten und bieten potentiellen Angreifern wenige Ziele, durch den Mangel an externen Schnittstellen.

Monolithen besitzen meist keinen wiederverwendbaren Code, benötigten leistungsstarke Hardware, sind nicht horizontal skalierbar und schlecht wartbar oder erweiterbar. Aufgrund der schwerwiegenden Nachteile von Monolithen haben sich weitere Architekturen herausgebildet. [11]

4.1.2 Service-Oriented Architecture

SOA basierte Programme präsentiert sich dem Nutzer als eine Einheit, sie können aber aber aus mehreren Diensten bestehen. SOA-Dienste bilden Geschäftsprozesse ab und legen dabei besonderen Fokus auf Wiederverwendbarkeit des Codes. Durch die Teilung des Programms in eindeutig abgegrenzte Komponenten ist die Software einfacher zu warten und zu erweitern. Zusätzlich können Kosten gesenkt werden, da Code wiederverwendet werden kann, solange alle Softwareprojekte auf demselben Technologie-Stack basieren.

Besonders in einer sehr wandelbaren Industrie wie der Informatik kann diese bedingte Wiederverwendbarkeit ein Nachteil sein. Aus der starken Abhängigkeit von einem Technologie-Stack können hohe Kosten durch Sicherheits-Updates und geringe Flexibilität folgen. [12]

4.1.3 Microservices

Microservises sind eine neue Entwicklung. Der Begriff Microservice wurde im Jahr 2011 zum ersten Mal genutzt[13], seitdem hat sich jedoch keine einheitliche Definition durchgesetzt. Zu den Charakteristika eines Microservice (MS) gehören die Folgenden:

- Dienste kommunizieren technologieunabhängig über ein Netzwerk (z. B. HTTP) [13].
- Dienste sind unabhängig voneinander deploybar [14].
- Dienste kommunizieren über Schnittstellen und sind daher Programmiersprachen unabhängig [15].
- Dienste sind leichtgewichtig und haben oft genau nur eine Funktion [14].

Aufgrund dieser Eigenschaften können einzelne Backend-Funktionalitäten von kleinen, unabhängigen Teams weiterentwickelt werden und das Risiko von Fehlern in anderen Komponenten minimiert werden. Mit dem Output der Dienste wird von einem weiteren Dienst das UI gerendert, welches dem User präsentiert wird.

Die Unterschiede zu SOA liegen vor allem in der loseren Kopplung, der feineren Unterteilung von Funktionen in eigene Dienste und der Technologienunabhängigkeit der einzelnen Dienste.

4.1.4 Self-Contained-Systems

Self-Contained-Systems denken den MS-Gedanken einen Schritt weiter. Es wird nicht nur das Backend in kleinere Komponenten gegliedert, sondern auch das Frontend. Man erhofft sich hieraus dieselben Vorteile wie auch bei der Aufteilung des Backends.

Jedes SCS ist ein unabhängiger Dienst, welcher von einem Team gepflegt wird. Jedes SCS enthält die gesamte benötigte Logik, Daten und UI. Ein SCS ist nicht mehr als ein MS mit UI. [3]

Wie in Bild 4.2 dargestellt können mehrere SCS einen größeren modularen Dienst bilden. Die Komponenten dieses Dienstes sind aufgrund ihrer APIs einfach gegen andere austauschbar und können einander nicht negativ beeinflussen.

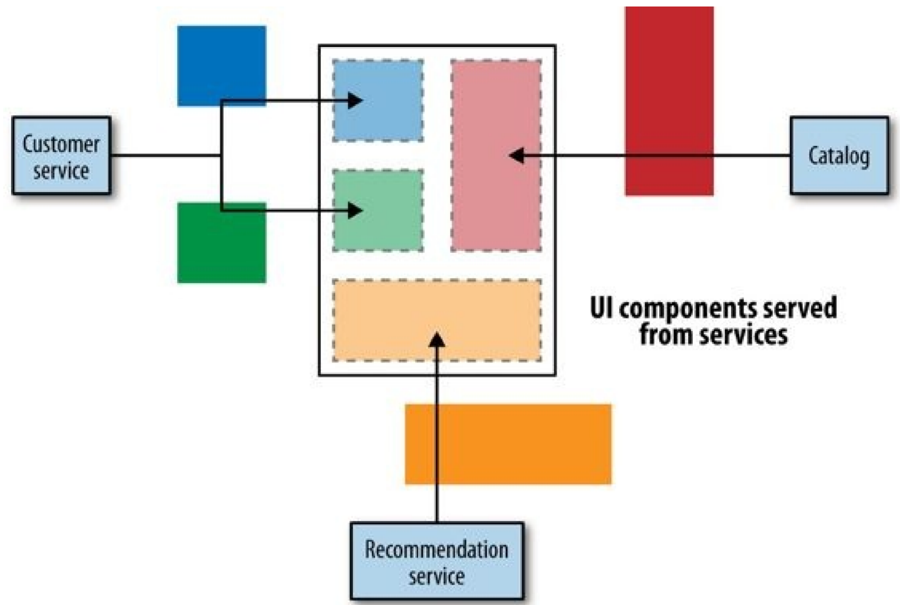

Bild 4.2: SCS basierte Online-Plattform [16]

4.2 RESTful API

SCS, wie sie in dieser Arbeit verwendet werden, benötigen APIs. Representational-State-Transfer (REST) ist ein Application-Programming-Interface (API) Design-Pattern. REST-Schnittstellen sind zustandslos. Anders als ein Webserver, welcher meist eine Session besitzt, speichert ein RESTful Webservice keine Daten zwischen Anfragen. Das bedeutet, dass für jede Anfrage Authentifizierung und Autorisierung geprüft werden müssen.

Eine API wird RESTful genannt, wenn sie mindestens folgende Anforderungen erfüllt: Objekte werden über Pfade adressiert, es werden HTTP-Methoden genutzt und "Hypermedia Controls" verwendet, soweit dies möglich und sinnvoll ist [17].

4.2.1 Pfade

REST-Schnittstellen adressieren Objekte über Pfade. Diese Pfade beginnen meist mit einem Vorsatz, welcher signalisiert, dass es sich um eine API und um welche Version es sich handelt, z. B. "/api/v1".

In Pfaden gibt es zwei Arten von Parametern: Pfad-Parameter und Query-Parameter. Ein Pfad-Parameter kann z. B. die ID eines Objektes oder auch die Art eines Objektes sein. Ein Beispiel hierfür wäre der folgende Pfad:

```
1 /api/v1/users/1
```

In diesem Beispiel sind zwei Pfad-Parameter enthalten, die Klasse des Objektes und die ID des Objektes. Die angefragte Ressource ist ein Nutzer und dieser Nutzer besitzt die ID 1.

Würde keine ID angegeben werden, sollten alle Nutzer ausgegeben werden. Soll diese Auswahl eingegrenzt werden können Query-Parameter genutzt werden. Dieser Beispielpfad könnte alle Nutzer ausgeben, welche im Januar 2019 angelegt wurden.

```
1 /api/v1/users?from=1.1.2019\&to=1.2.2019
```

4.2.2 HTTP-Methoden

HTTP-Methoden deuten auf gewünschte Aktionen hin, welche auf Ressourcen ausgeführt werden sollen. Eine GET-Request beispielsweise soll Daten beziehen, wohingegen ein POST üblicherweise Daten übermitteln soll.

Diese Methoden sind in den RFC Spezifikationen 7231 und 5789 wie folgt definiert:

GET Die GET-Methode greift auf ein spezifisches Objekt zu. GET sollte nur zum Beziehen von Daten genutzt werden [18].

HEAD Die HEAD-Methode erfragt dasselbe wie die GET-Methode, allerdings ohne den Response Body [18].

POST Die POST-Methode wird genutzt um ein neues Objekt in einer Ressource anzulegen. Dies kann Statuswechsel oder andere Seiteneffekte zur Folge haben [18].

PUT Die PUT-Methode ersetzt ein gesamtes Objekt mit dem Inhalt der Anfrage [18].

DELETE Die DELETE-Methode löscht das spezifizierte Objekt [18].

CONNECT Die CONNECT-Methode etabliert einen Tunnel, spezifiziert durch das Ziel der Anfrage [18].

OPTIONS Die OPTIONS-Methode wird verwendet, um eine Beschreibung der Kommunikation mit dem Ziel zu erfragen [18].

TRACE Die TRACE-Methode führt ein Loop-Back Test zum Pfad der Ressource durch [18].

PATCH Die PATCH-Methode ähnelt dem PUT, allerdings wird nicht das vollständige Objekt ersetzt, sondern nur die Änderungen übernommen[19].

4.3 Verteilte Architekturen

Jede Applikation mit großen Mengen gleichzeitiger Nutzer muss früher oder später eine verteilte Architektur annehmen. Populäre Beispiele hierfür sind Amazon, Zalando oder Netflix [20]. Um Ressourcen so effizient wie möglich einsetzen zu können haben die oben genannten Beispiele ihre Architekturen auf Microservices und / oder Self-Contained-Systems umgestellt.

Ein Self-Contained-System ist eine Komponente eines Dienstes. Diese Komponenten können verschiedene Makroarchitekturen bilden. In diesem Kapitel werden einige Ausprägungen vorgestellt.

Der gedankliche Schritt von SOA zu Microservices ist kurz, Ressourcen werden nun nicht mehr über Klassen und Funktionen bezogen, sondern über HTTP(S)-Requests. Aus Sicherheits-, Überwachungs- und Leistungsgründen kann dieses simple Konzept des direkten Aufrufes eines MS um weitere Komponenten und Konzepte ergänzt werden.

In diesem Unterkapitel werden die grundlegenden Konzepte von Microservices, Service-Meshes und API-Gateways vorgestellt.

4.3.1 Microservices

MS als Begriff tauchte zum ersten Mal bei einem Treffen von Softwarearchitekten im Jahr 2011 auf [13]. Es wird gelegentlich als Unterkategorie der SOA bezeichnet. Microservices stellen eine API bereit, über welche Daten bezogen werden können, meist ist diese RESTful. Die API nutzt HTTP, der Pfad repräsentiert den Weg zu einem Objekt; die Kommunikation erfolgt zustandslos.

Microservices zeichnen sich durch ihre Leichtgewichtigkeit und ihren Fokus auf genau eine Aufgabe aus. Das Teilen von Prozessen in einzelne Aufgaben und somit Services, kann eine Herausforderung, beim Entwerfen von Microservices, darstellen.

4.3.2 Service-Mesh

Service-Meshes sind eine Spezialform der Microservices. Eine Komponente eines Service-Meshes folgt dem sogenannten Sidecar-Pattern. Jedem Service wird ein Sidecar bereitgestellt.

Ein Sidecar kann eine Vielzahl von Aufgaben erfüllen:

- Service-Discovery
- Routing
- Zertifikaterzeugung, -Verwaltung und -Validierung
- Health-Checks
- A/B-Tests
- TLS-Terminierung

Vieles an allgemeiner Funktionalität, welche jeder Dienst in einem Service-Mesh bereitstellen muss wie kryptografische Funktionen, Routing oder Service Discovery, kann in ein Sidecar ausgegliedert werden. Hier bieten sich Docker und Kubernetes an. Jedes Sidecar kann exakt gleich konfiguriert sein und daher dasselbe Docker-Image / Template verwenden.

4.3.3 API-Gateway

Microservice und Service-Meshes besitzen unter anderem Nachteile im Logging. Bei der Feststellung und Behebung von Fehlern sind detaillierte und vollständige Logs essentiell. MS sind grundsätzlich verteilt, genau wie ihre Logs. Unter Umständen ist ein Fehler nur in einer Replik eines Dienstes aufgetreten, welche bereits neu gestartet wurde, wodurch das lokale Log verloren gegangen sein könnte.

Ein API-Gateway ist eine zentrale Komponente, welche das gesamte Routing, Logging, sowie Authentifikation und Autorisation durchführt. Es ähnelt in seiner Funktion also einem Sidecar, welches zwischen allen Diensten geteilt wird.

Ein API-Gateway ist ein zentraler Dienst, welcher die Funktionalität von mehreren MS übernehmen kann. Das macht es auch zu einem „single point of failure”.

4.4 Docker

Docker ist eine moderne Container-Technologie. Container sind eine Form von Software-Virtualisierungstechnologie, andere bekannte Beispiele hiervon sind z. B. Virtuelle Maschinen oder LXC-Container. Container ermöglichen es Software inklusive aller Abhängigkeiten auszuliefern, hiermit eignen sie sich besonders gut für SCS.

Das Ziel aller Software-Virtualisierungstechnologien ist es, den Betrieb von Software effizienter zu gestalten. Dies kann sich auf die Auslastung von Ressourcen, einen reduzierten Ressourcenbedarf, höhere Verfügbarkeit, oder auf einen schnelleren Entwicklungs- und Bereitstellungsprozess beziehen. Docker erfüllt diese Ziele, und bietet weitere Vorteile.

Docker verwendet sogenannte Dockerfiles, um das gesamte Deployment einer Applikation zu beschreiben. Diese Dockerfiles können zu Images zusammengebaut werden. Images sind unveränderlich und können in Image Repositorys gespeichert werden. Aus Images werden zur Ausführung Container gebaut. Container besitzen per default keinen persistenten Speicher und stellen somit sicher, dass jedes erneute Deployment unverändert zu den vorherigen ist.

Docker hat in den letzten Jahren an Popularität gewonnen, da Docker-Container in allen Phasen der Softwareentwicklung eingesetzt werden können. Für den Betrieb liegen die Vorteile bei der einfachen Administration durch Orchestrierungslösungen wie Kubernetes oder OpenShift. Während der Entwicklung erleichtern Portweiterleitung, einfaches und konsistentes Konfigurationsmanagement, geringe Start-Up Zeit und ein erleichtertes Update-Management die Arbeit [21].

Docker ermöglicht es außerdem denselben Container aus der Entwicklungsphase in den produktiven Betrieb zu nehmen, was das Ausführen von Updates weniger riskant macht. Betriebsumgebung und Entwicklungsumgebung sind dadurch identisch, Konfigurationsfehler oder Fehler durch weitere Unterschiede der Umgebungen sind somit unwahrscheinlicher [21].

Docker allein ist eine nützliche Technologie, besonders deutlich werden die Vorteile aber erst in Kombination mit anderen Technologien. In den vergangenen Jahren hat sich ein Ökosystem um Docker gebildet. Innerhalb dieses Ökosystems kann Docker seine Stärken zeigen.

4.5 Kubernetes

Kubernetes ist eine Open-Source Container-Orchestrierungslösung für Docker- Container. Orchestrierungsplattformen ermöglichen das gleichzeitige Betreiben von vielen Diensten. Somit eignen sie sich für die Administration verteilter Systeme.

Kubernetes hat den aktuell höchsten Marktanteil [22], unter den Docker- Orchestrierungsplattformen. Zu den Alternativen zu Kubernetes gehören RedHat's OpenShift, Cloud Founrdy und PKS von VMWare. Diese Konkurrenten basierten ursprünglich auf unterschiedlichen Technologien, allerdings stellt Kubernetes mittlerweile die Basis für diese Plattformen. Die jeweiligen Herrsteller ergänzen ihre eignen Konzepte. Sie können als Kubernetes-Distributionen bezeichnet werden.

Kubernetes nutzt einige zusätzliche Konzepte, ein besonders wichtiges hiervon sind Pods. Ein Pod ist die kleinste verwaltbare Einheit in einem Kubernetes Cluster. Ein Pod kann einen oder mehrere Container enthalten. Pods haben eine eigene IP-Adresse und eine eigene Port-Range.

Kubernetes und Kubernetes-Distributionen übernehmen viele Funktionen von Virtualisierungsplattformen, wie die Überwachung von Prozessen und Ressourcenauslastung, Netzwerkadministration, Failover-Management und Ressourcenmanagement [23]. Des Weiteren ist es über das Rechtemanagement möglich, Entwicklern Einblicke in das produktive Verhalten einer Applikation zu verschaffen. Bei einem traditionellem Deployment auf einer Virtuelle Maschine (VM) oder einem Bare-Metal Server kann dies schwer oder nicht möglich sein, dies ist abhängig von den organisatorischen Strukturen.

4.6 Sicherheitsaspekte

Verteilte Systeme bieten große Angriffsflächen, da sie Schnittstellen benötigen. In diesem Kapitel werden verschiedene, für diese Arbeit relevante, Sicherheitsaspekte und Authentifikationskonzepte erläutert. Diese Konzepte sollen dabei helfen die Angriffsfläche zu minimieren.

Bezüglich der Authentifizierung werden in diesem Kapitel zwei Aspekte gesondert betrachtet:

- Steht ein Nutzer hinter dieser Anfrage?
- Wie kann Dienst A Dienst B vertrauen?

4.6.1 Anfrageauthentifizierung

Die Anfrageauthentifizierung beschäftigt sich mit der Frage, ob ein Nutzer oder ein potentieller Angreifer hinter einer Anfrage steht. Diese Überprüfung ist relativ simpel. Der Nutzer muss sich authentisieren damit Services auf Anfragen reagieren.

In monolithischen Systemen ist diese Überprüfung einfach. Es kann geprüft werden, ob eine Session für den anfragenden Nutzer existiert und ob dieser eingeloggt ist. In verteilten Systemen gestaltet sich dies komplexer, da es nicht immer einen für alle Komponenten zugänglichen Speicher gibt.

Sticky Session Das „sticky session" Prinzip basiert darauf, dass alle Anfragen eines Nutzers zu dem Server geschickt werden, welcher die erste Anfrage beantwortet hat. Somit besitzt der angefragte Server immer die richtige Session. Diese Art der Session-Verwaltung bietet sich bei horizontal skalierten Monolithen an. Jede Instanz eines Monolithen besitzt alle nötigen Funktionen und Berechtigungen, um jede Anfrage eines Nutzers zu verarbeiten. Bei verteilten Systemen ist dies nicht der Fall, da jeder Service nur einen Teilaspekt der Anfrage verarbeiten kann.

Session Replication Session-Replikation sorgt dafür das jedem Dienst jede Session vorliegt, indem jede Erstellung, Änderung und Löschung einer Session über das Netzwerk jedem anderen Dienst mitgeteilt wird. Dies sorgt für eine hohe Netzwerklast, welche mit jedem beteiligten Dienst ansteigt. Die Anzahl der Anfragen pro Änderung einer Session entspricht n^2, wobei n der Anzahl

der Dienste entspricht. Zusätzlich kann die Konsistenz der Session nur schwer sichergestellt werden.

Centralized Session Storage Ein zentraler Session-Speicher enthält alle aktiven Sessions und ist von allen Diensten nutzbar. Der zentrale Session-Speicher kann transaktionsbasiert sein, wodurch die Konsistenz bewahrt wird. Ein zentraler Session-Speicher ist auch ein „single point of failure“, besonders in lose gekoppelten verteilten Systemen ist dies nicht wünschenswert. Der Session-Speicher sollte redundant sein. Aufgrund der potentiell sensiblen Daten sollte in diesem Fall auch eine Service-to-Service Authentifizierung stattfinden.

4.6.2 Service-Authentifizierung

Zu den typischen Irrtümern in Bezug auf verteilte Systeme gehört der folgende:

> „Das Netzwerk ist sicher“. [24]

Besonders in sensiblen Domänen wie E-Government hat die Datensicherheit eine hohe Priorität. Daher sollte der Ausgangszustand der des „zero trust“ sein. Das bedeutet, dass die Verbindung verschlüsselt sein muss, dass jeder Dienst sich jedem anderen Dienst gegenüber authentisieren muss und, dass die Authentifizierung symmetrisch erfolgen muss. Der Server muss sich dem Client gegenüber authentisieren und der Client dem Server gegenüber. Erst dann kann davon ausgegangen werden, dass die Kommunikation sicher erfolgt.

Es existieren offene Standards wie OpenID Connect oder JSON Web Token (JWT), diese werden im Folgenden nicht betrachtet, da sie nur der Nutzerauthentifizierung und Autorisierung dienen und daher für den oben beschriebenen Einsatz, der Service-to-Service Authentifizierung nicht geeignet sind.

Da REST-Schnittstellen HTTP basiert sein können ist die Verwendung von Zertifikaten naheliegend. Durch die Verwendung von HTTPS können zwei von drei Anforderungen erfüllt werden. Die Verbindung ist automatisch verschlüsselt und das verwendete Serverzertifikat kann validiert werden.

Zusätzlich können Client-Zertifikate verwendet werden, um den Client gegenüber dem Server zu authentisieren. Server und Client können nun die Zertifikate der jeweiligen Gegenseite gegen ein vorliegendes Certificate Authority (CA) Zertifikat validieren.

Die Validierung der Zertifikate besteht aus 4 Schritten: Die Validierung des Stammzertifikats, die Validierung der Zwischenzertifikate, dem Revocation-Check und der Gültigkeitsdauer-Check [25]. Die ersten beiden Schritte sind rein Kryptografische Prozesse, welche sicherstellen, dass die Zertifikate nicht durch einen Angreifer ausgetauscht wurden. Mit dem Revocation-Check wird überprüft, ob die Zertifikate von der CA abgestoßen wurden. Dies kann passieren, wenn Zertifikate als nicht länger sicher gelten z. B. weil die privaten Schlüssel bekannt geworden sind.

Der Validierungsprozess wird zwei Mal durchlaufen, einmal pro Zertifikat. Zuerst wird eine HTTPS-Verbindung zwischen Server und Client hergestellt und das Server Zertifikat validiert, anschließend wird das Client-Zertifikat übertragen. Besonders aufgrund des Revocation-Checks kann dieser Prozess die Latenz eines Aufrufes signifikant erhöhen.

4.6.3 Weitere Sicherheitsüberlegungen

Je früher ein Angriff scheitert, desto besser. Die bisherigen Authentifizierungskonzepte lassen Anfragen erst am Dienst selbst scheitern. Das bedeutet, dass auch abgewendete Angriffe Kosten erzeugen. Eine Möglichkeit dies zu verbessern ist die Verwendung eines Reverse-Proxys. Ein Reverse-Proxy verbirgt die Topologie des jeweiligen Netzwerkes und ermöglicht das Abweisen von offensichtlich fehlgeleiteten Anfragen. Sollte ein Nutzer einen API-Pfad für interne Dienste aufrufen kann dies bereits unterbunden werden bevor der Dienst die Anfrage selbst abweisen muss.

Zudem kann ein Reverse-Proxy als Load Balancer eingesetzt werden, um die Last auf mehrere Instanzen des Dienstes zu verteilen. Ein Dienst kann eine Ressource bereitstellen, welche von Nutzern gelesen werden darf; Nutzer sollen aber nicht unbedingt neue Objekte erstellen dürfen. Hier können ebenfalls mit einem Reverse-Proxy bestimmte HTTP-Methoden abgewiesen werden.

Mit diesen Konzepten sollte es weitgehend möglich sein ein verteiltes System sicher zu betreiben.

5 Entwurf

In diesem Kapitel wird der Entwurf vorgestellt, der prototypisch implementiert wird.

Es wird mit der Architektur und einer Abwägung verschiedener Makroarchitekturen begonnen. Anschließend folgt der API-Entwurf, mit diesem können eine Reihe von Abläufe umgesetzt werden, diese werden im Anschluss modelliert.

5.1 Architektur

In Kapitel 4.3 wurden drei Architekturen vorgestellt: Microservices, Service-Meshes und API-Gateways. Aufgrund der 3.1 fallen einfache Microservices aus der Betrachtung. Es sind Authentifizierungskonzepte und verschlüsselte Verbindungen nötig, daher sind Service-Meshes und API-Gateways geeigneter.

Die Abwägung zwischen API-Gateway und Service-Mesh fällt ebenfalls einfach, da eine lose Kopplung gefordert wird. Das API-Gateway ist eine zentrale Komponente und erhöht somit die Kopplung. Fällt diese zentrale Komponente aus, ist kein Dienst mehr erreichbar. Das ist explizit nicht gewünscht.

Die Service-Mesh-Architektur wurde aufgrund der Möglichkeiten in Hinsicht auf TLS-Terminierung, Authentifizierung und Unabhängigkeit der Dienste untereinander gewählt.

In Bild 5.1 ist eine vereinfachte Übersicht der angestrebten Architektur dargestellt. Externe Komponenten sind blau, vom OZG vorgeschriebene Komponenten sind grau, veränderte oder erweiterte Komponenten sind orange und neue Komponenten sind grün markiert.

Diese Übersicht zeigt nicht auf welchen Servern welche Dienste zu deployen sind. Dies ist eine bewusste Entscheidung. Es steht noch nicht fest, ob und wann Docker-Container produktiv eingesetzt werden können. Aufgrund der in Kapitel 4.4 erläuterten Vorteile besteht großes Interesse daran, dies so bald wie möglich zu tun. Der

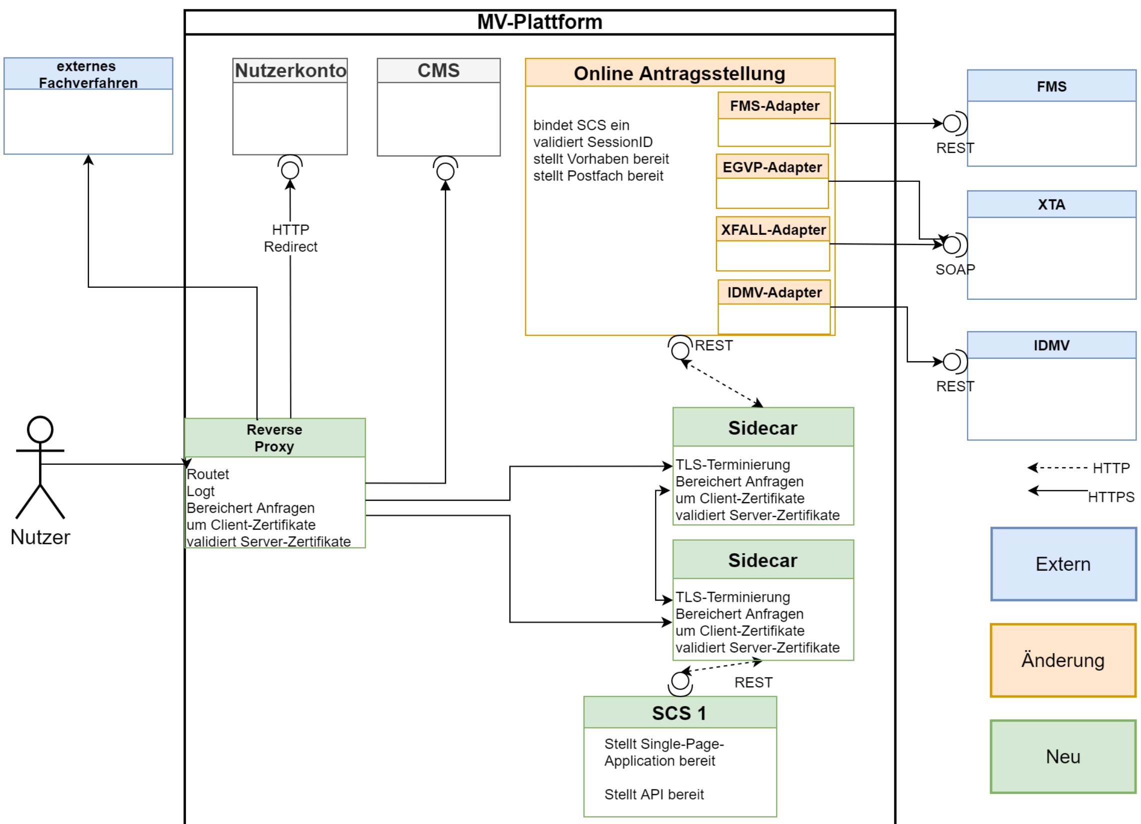

Bild 5.1: Angestrebte Architektur

Entwurf wurde daher generisch genug gehalten, um sowohl auf traditionellen Servern als auch mit Kubernetes und in Docker-Containern deployt werden zu können.

Sämtliche externen Anfragen erfolgen über den Reverse-Proxy . Nur so sind die internen Dienste erreichbar. Dieser Proxy blockt unerwünschte HTTP-Methoden, abhängig von der Route. Ebenfalls werden Anfragen Client-Zertifikate angehängt damit die Sidecars sie akzeptieren.

Die gesamte Netzwerkkommunikation innerhalb des MVSP erfolgt verschlüsselt. Die Sidecars sorgen für die TLS-Terminierung und das Validieren der Client-Zertifikate. Somit muss die Zertifikatsvalidierung nicht in jeder Methode implementiert werden. Entwickler können sich somit auf Kernfunktionalität konzentrieren und menschliches Versagen wird vermieden.

Die Dienste kennen nicht die Adressen der anderen Dienste. Die Kommunikation erfolgt nur mit relativen Pfaden. Die Auflösung dieser relativen Pfade zu absoluten obliegt den Sidecars. Fragt der Dienst `SCS1` das MVSP an, so ruft der Dienst die folgende Adresse auf: `http://sidecarSCS1/MVSP`, `sidecarSCS1` entspricht hierbei dem Namen des Sidecars. Das Sidecar kann den aufgerufenen Pfad auflösen, da die

Sidecars pro Deployment konfiguriert werden.

In den IDMV muss der zugehörige Online-Dienst für ein Fachverfahren hinterlegt werden. Dies geschieht über einen Link. Im IDMV soll aber keine deploymentspezifische Konfiguration hinterlegt werden, daher muss der Link relativ sein. Dieser relative Link wird über den Sidecar des jeweiligen Dienstes aufgelöst. Dies vereinfacht die Entwicklung und Konfiguration - besonders wenn Docker verwendet wird. Dasselbe Image kann für jeden Sidecar verwendet werden. Somit müssen pro Deployment nur der Reverse-Proxy und ein Sidecar-Image angepasst werden.

Der Online-Dienst ist zustandslos und benötigt daher keine Session. Allerdings nutzt der Online-Dienst die serverseitige SessionID des MVSP, um eine Identifikation des Nutzers vorzunehmen. Die eAST dient somit als zentraler Session-Speicher, was bei einer horizontalen Skalierung zu beachten wäre.

5.2 API

Verteilte Applikationen bestehen aus mehreren Diensten, diese müssen untereinander kommunizieren können. Hierfür sind Schnittstellen nötig. Diese können verschiedene Formen annehmen. Aufgrund des hohen Anteils an REST-Schnittstellen im direkten Umfeld des MVSP wurde für REST-Schnittstellen in der eAST und OD entschieden.

In diesem Kapitel werden die APIs des MVSP und die Minimal-API der Online-Dienste erläutert. Eine OpenAPI-konforme[26] Dokumentation kann im Anhang gefunden werden.

Der API-Kontext beginnt grundsätzlich mit:

/api/v1/

5.2.1 MVSP API

Da wie im Kapi 4.2 erläutert REST-APIs auf Pfaden zu Objekten basieren, gilt es als erstes diese Objekte zu finden.

Das MVSP wird 3 Objekte bereitstellen: das Postfach, den Session-Speicher und Teilvorhaben.

Teilvorhaben

Teilvorhaben oder Partial-Applications sind Teile von Applications. Eine Application ist ein Antrag auf eine Leistungen. Diese Namensgebung stammt aus dem XFall-Umfeld [27] und bezeichnet eine Datenstruktur. Einige Verwaltungsleistungen benötigen mehrere Anträge, daher ist diese Unterteilung nötig. Eine Partial-Application kann über ihre Partial-Application-ID identifiziert werden. In der Partial-Application sind Daten, wie der Absendezeitpunkt, Status, Sicherheitsstufe usw. gespeichert.

Eine Partial-Application kann nur vom MVSP selbst erstellt oder gelöscht werden. Partial-Applications können von weiteren Diensten abgerufen und teilweise geändert werden.

Zu unterstützende HTTP-Methoden:

- GET
- PATCH / PUT

Route:

/api/v1/vorhaben

Session-Speicher

Der Session-Speicher ist nötig, da Dienste Anfragen validieren können müssen und um weitere Nutzerdaten abzufragen. Aufgrund der Überlegungen im Unterkapitel 4.6.1, wurde für einen zentralen Session-Speicher entschieden. Sessions werden vom MVSP verwaltet und können daher von externen Diensten nur validiert werden.

Zu unterstützende HTTP-Methoden:

- GET

Route:

/api/v1/sessions

Postfach

Um einen Nutzer über den Stand seiner Anträge zu informieren, wird das Postfach genutzt. Das Löschen und Lesen der Nachrichten obliegt dem Nutzer, die Schnittstelle muss daher nur POSTs unterstützen.

Zu unterstützende HTTP-Methoden:

- POST

Route:

/api/v1/inbox

5.2.2 Dienst-API

Um verschiedenste Online-Dienste einbetten zu können, muss jeder Dienst mindestens eine Minimal-API enthalten.

Jeder Online-Dienst muss eine lokale Repräsentation der vorliegenden Partial-Applications besitzen. Auf diesem Objekt müssen CRUD-Funktionen möglich sein.

Die daraus resultierenden zu unterstützenden HTTP-Methoden sind daher:

- POST
- GET
- PATCH / PUT
- DELETE

Zusätzlich muss im Root-Kontext einbettbares HTML zurückgegeben werden, ebenso müssen Dateien mit den Namen „main.js” und „main.css” im Static-Kontext abrufbar sein. Diese Dateien enthalten den nötigen JavaScript (JS) Code und das zugehörige CSS, sie werden standardmäßig eingebunden.

5.3 Abläufe

In diesem Kapitel werden eine Reihe von Abläufen modelliert, welche übliche Nutzungsszenarien abbilden. Diese werden in Kapitel 7 implementiert. Sie sind darauf ausgelegt alle definierten Schnittstellen zu nutzen.

5.3.1 Erstellen

Der „Erstellen"-Prozess wird vom Nutzer, nach dem Finden des zugehörigen Online-Dienstes mithilfe der Daten aus dem IDMV, ausgelöst. Wie in Bild 5.2 abgebildet, geschieht dies durch das Auslösen der „createApplication"-Funktion. In dieser Funktion wird eine Application in der Datenbank angelegt. Anschließend werden die Partial-Applications mit dem zugehörigen Verweis auf die Application in der Datenbank angelegt. Für das Anlegen werden Daten, aus dem IDMV benötigt, wie z. B. der Zustellkanal oder die Ablaufzeit.

Nach dem erfolgreichen Anlegen der Objekte in der Datenbank der eAST wird eine POST-Request an den Online-Dienst getätigt. Das Ziel des Aufrufes wurde vor dem Anlegen der Objekte ebenfalls dem IDMV entnommen. Der, aus dem IDMV stammende, relative Link wird vom Sidecar aufgelöst. Dieser wurde zur Verbesserung der Übersichtlichkeit nicht in das Diagramm mit aufgenommen. Die Request enthält die Partial-Application-Id, mit welcher der Online-Dienst weitere Daten zur Partial-Application abrufen kann. Wenn der Online-Dienst ein lokales Objekt erfolg-

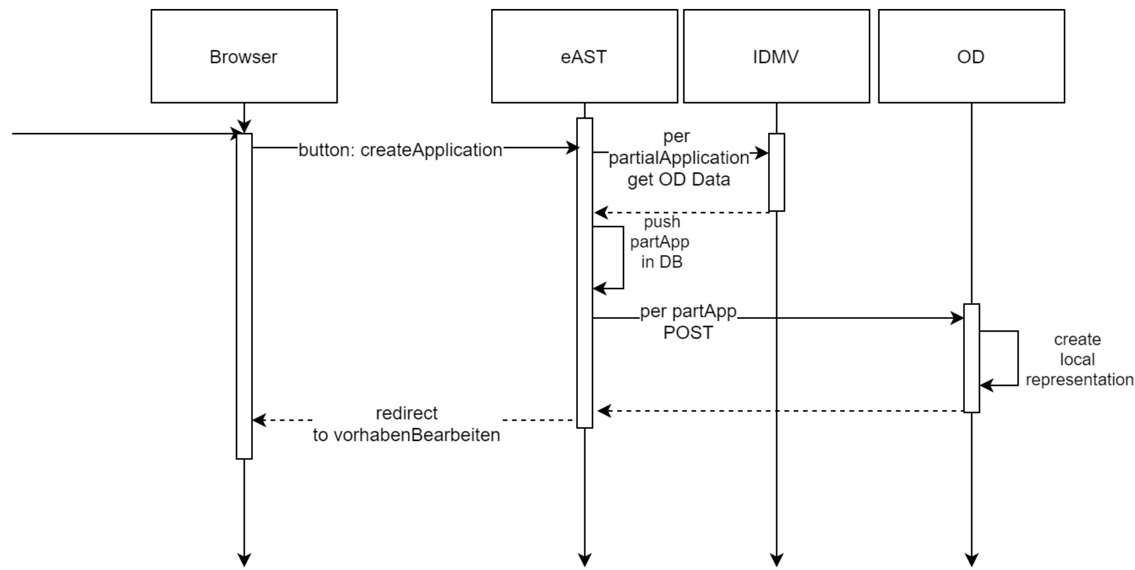

Bild 5.2: Sequenzdiagramm: Erstellen einer Application

reich anlegen konnte, gibt der Online-Dienst den HTTP Status Code 201 zurück, um zu signalisieren, dass ein neues Objekt angelegt wurde.

Anschließend wird der Nutzer auf eine Seite, zum Einsehen der Details und Ändern des Vorhabens, umgeleitet. Dieser Prozess ist im nächsten Kapitel erläutert.

5.3.2 Lesen und Ändern

Der „Lesen / Ändern"-Prozess wird mit dem Aufrufen der Seite „VorhabenBearbeiten" ausgelöst und ist in Bild 5.3 dargestellt. Auf dieser Seite kann ein Nutzer die Details seines Antrages einsehen und bearbeiten, bevor der Antrag abgeschickt wird. Im Standardfall wird diese Seite vom Formular-Management-System bereitgestellt. Benötigt die Partial-Application aber einen domänenspezifischen Online-Dienst, so wird dieser eingebunden.

Beim Aufrufen der Seite wird daher mit dem IDMV geprüft, ob der benötigte Online-Dienst domänspezifisch ist und wenn ja, unter welcher relativen URL der Dienst erreichbar ist. Die eAST erwartet, dass im Static-Kontext der eingebundenen Applikation eine JavaScript-Datei existiert. Diese Datei enthält die Funktionen für das Laden der Daten und Animationen. Mit dem Laden der Website beim Client wird eine JavaScript Funktion auslöst, welche den Root-Kontext des Online-Dienstes aufruft und den dort vorliegenden HTML-Code in einen Platzhalter des Templates einfügt.

Damit ist das statische Gerüst der Seite vollständig geladen. Anschließend daran werden die Daten geladen. Hierfür wird vom Client eine GET-Request auf das Vorhaben im Online-Dienst ausgeführt. Bevor der Online-Dienst diese Anfrage beantwortet, wird die Existenz der übergebenen Session überprüft. Die Anfrage wird nur bearbeitet, wenn die Session zur übergebenen SessionID existiert. Der Online-Dienst speichert lokal nur verfahrensspezifische Daten, werden weitere Daten wie z. B. der Status der Partial-Application benötigt, so muss der Online-Dienst eine Anfrage gegen die eAST durchführen. Hierfür muss ebenfalls die SessionID übermittelt werden, damit die eAST überprüfen kann, ob die angefragte Partial-Application zum anfragenden Nutzer gehört. Ist dies der Fall, gibt die eAST ein JSON-Objekt zurück, welches vom Online-Dienst mit den lokalen Daten zusammengeführt wird. Das dadurch entstehende JSON-Objekt wird anschließend an den Client zurückgegeben.

Damit ist die Seite vollständig geladen und bereit um vom Nutzer verändert zu werden. Hat der Nutzer eine Änderung vorgenommen und möchte diese speichern, so

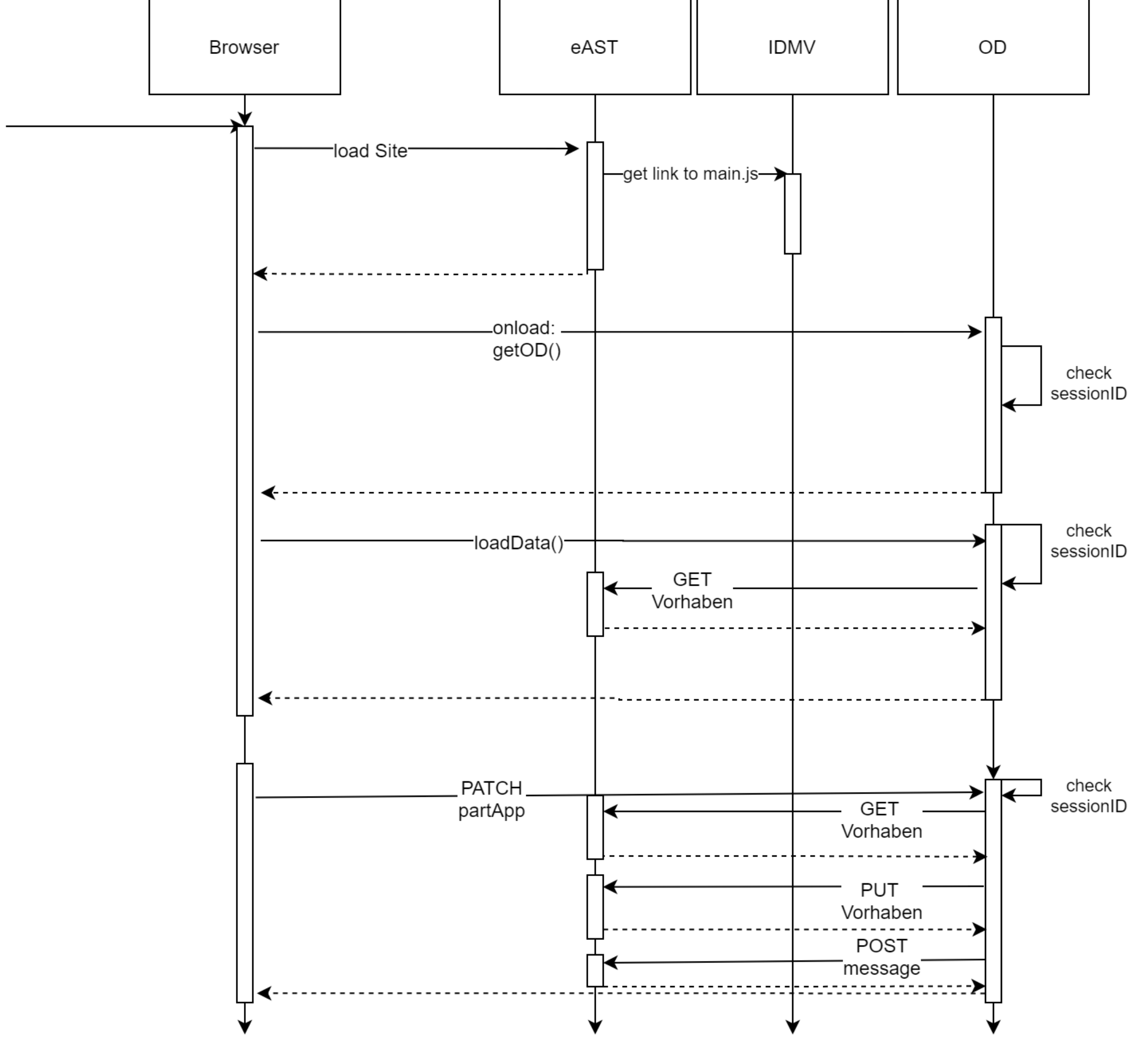

Bild 5.3: Sequenzdiagramm: Lesen und Ändern einer Application

werden die änderbaren Daten zurück an den Online-Dienst geschickt und dort validiert und gespeichert. Der Nutzer bekommt anschließend eine visuelle Rückmeldung über den Erfolg der Aktion und eine Nachricht in seinem Postfach.

5.3.3 Löschen

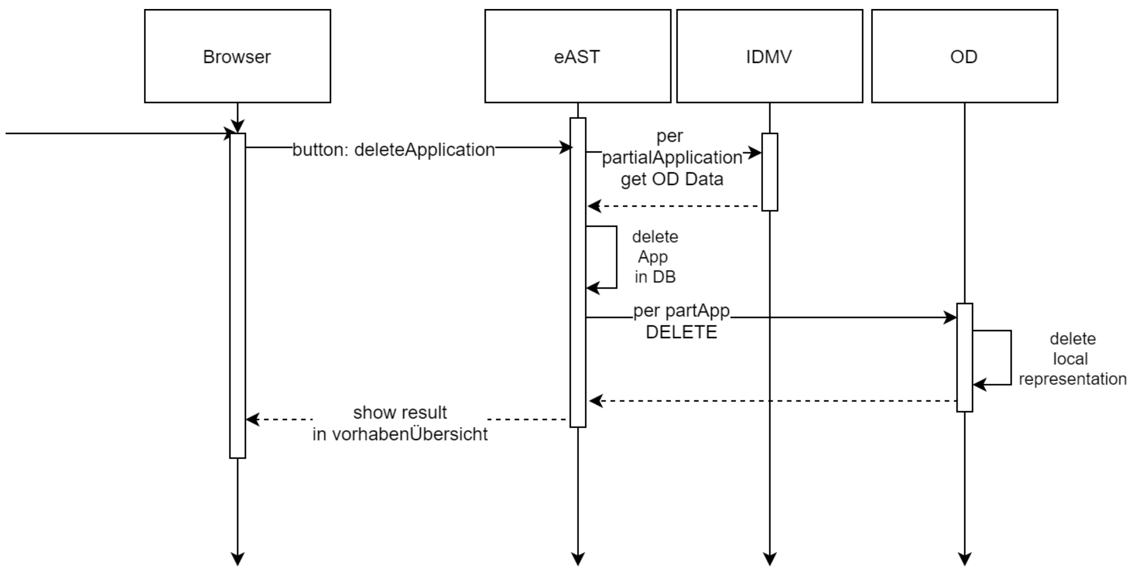

Bild 5.4: Sequenzdiagramm: Löschen einer Application

Das Löschen von Vorhaben ist in der Vorhabenübersicht möglich. Wie in Bild 5.4 zusehen, wird zuerst überprüft, ob die Application löschbar ist, oder ob sie bereits abgeschickt ist. Ist sie löschbar, so wird die Application mit ihren Partial-Applications gelöscht. Ist dies erfolgreich, so wird eine DELETE-Request pro Partial-Application an die Partial-Application im zugehörigen Online-Dienst geschickt. Die Adresse des Online-Dienstes stammt, wie auch beim Erstellen, aus dem IDMV.

Ist das Löschen nicht möglich, weil das Vorhaben bereits abgeschickt wurde, so wird dies dem Nutzer mitgeteilt. Ist das Vorhaben, aufgrund eines technischen Fehlers, nicht löschbar, wird eine Fehlermeldung ausgelöst.

6 Voruntersuchung

In diesem Kapitel werden für die Umsetzung der entworfenen Architektur zentrale Technologien evaluiert und verglichen. Es wird untersucht, ob die Komposition der UI Server- oder Clientseitig implementiert werden sollte, wie der Client die für den jeweiligen Dienst zuständigen Server erreicht und wie sich die einzelnen Dienste im System untereinander authentifizieren und sicher miteinander kommunizieren können.

6.1 Docker

Docker ist ein hervorragendes Tool für die Entwicklung von verteilten Systemen. Docker bietet virtuelle Netze und eine Namensauflösung innerhalb dieser Netze. Somit lassen sich komplexe Verteilte Systeme auf einer VM oder lokal deployen.

Zusätzlich wurde Docker-Compose verwendet. Ein in Python geschriebenes Tool, welches eine YAML-Datei verwendet, um ein Deployment von mehreren Docker-Containern gleichzeitig zu ermöglichen. Alle Versuche wurden mit offiziellen Docker-Images durchgeführt.

6.2 Test verschiedener UI-Fragment-Composition Konzepte

UI-Fragment-Composition ist ein Begriff, der aus der Entwicklung von Microservices stammt. Es ist vergleichbar mit den verschiedenen Renderstrategien wie Client Side Rendering (CSR) oder Server Side Rendering (SSR). Gelegentlich werden diese Strategien auch als Client bzw. Server Side Include (CSI / SSI) bezeichnet. In Tabelle 6.1 werden sie miteinander Verglichen [28].

Tabelle 6.1: Vergleich der Vor- und Nachteile von SSR und CSR

	Server Side Rendering	**Client Side Rendering**
Vorteile	Vermeidung von CORS[1] JavaScript ist nicht notwendig bessere Browserkompatibilität bessere Barrierefreiheit besseres SEO[2]	geringe Serverlast geringe gefühlte Latenz stark bei Multiplattform
Nachteile	hohe Serverlast hohe Latenz Schwächen bei Multiplattform	Gefahr von CORS JavaScript notwendig Browserinkompatibilität Barrierefreiheit schlechtes SEO

Server Side Rendering

SSR ist so alt wie dynamische Websites. Bei SSR wird ein Template vom Server mit allen nötigen Daten befüllt und als eine Datei an den Client geliefert. Da die HTML-Datei auf dem Server generiert wird, bietet SSR einige Vorteile.

Client Side Rendering

CSR invertiert die Vor- und Nachteile des SSR. Wo SSR stark ist, ist CSR schwach. Client Side Rendering hat sich vor allem aufgrund der Nachteile des SSR entwickelt.

In Abbildung 6.1 ist zu sehen, dass sich die Zeiten bis zum vollständigen Laden der Seite kaum unterscheiden. Dem Nutzer erscheint dies aber anders. Der Nutzer bekommt beim CSR schneller eine Rückmeldung vom Server. Dies trägt dazu bei, dass sich eine App oder Website flüssiger im Umgang anfühlt. Die genauen Zahlen hierzu können im Kapitel 8 gefunden werden.

Wie in Abbildung 6.2 dargestellt, ist den beiden Containern ein Reverse-Proxy vorgeschaltet. Dieser ist aus Sicherheitsgründen nötig. Durch diesen Reverse-Proxy nimmt der Browser an, dass es sich bei den zwei Servern um einen handelt. Würde kein Reverse-Proxy verwendet werden, müsste CORS zugelassen werden, was ein potentielles Sicherheitsrisiko ist. Die Verwendung eines Reverse-Proxys ist nötig, da das Frontend des eingebetteten Online-Dienstes mit dem Backend kommunizieren muss.

[1] Cross Origin Ressource Sharing (CORS)
[2] Search Engine Optimization (SEO)

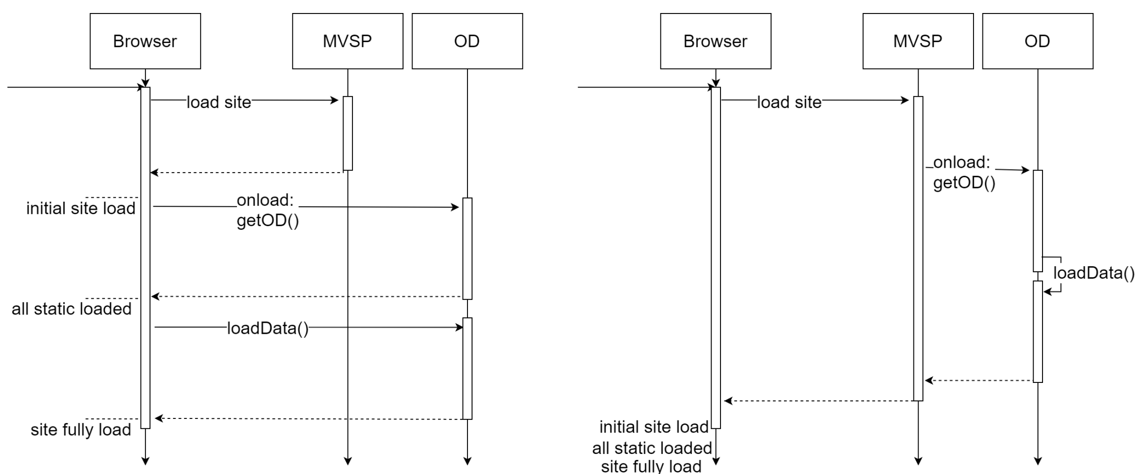

Bild 6.1: Vergleich der Ladezeiten CSI vs SSI

Dies erfolgt über HTTPS-Requests, welche nur gegen den Server laufen dürfen, von dem der Browser die Seite ursprünglich geladen hat.

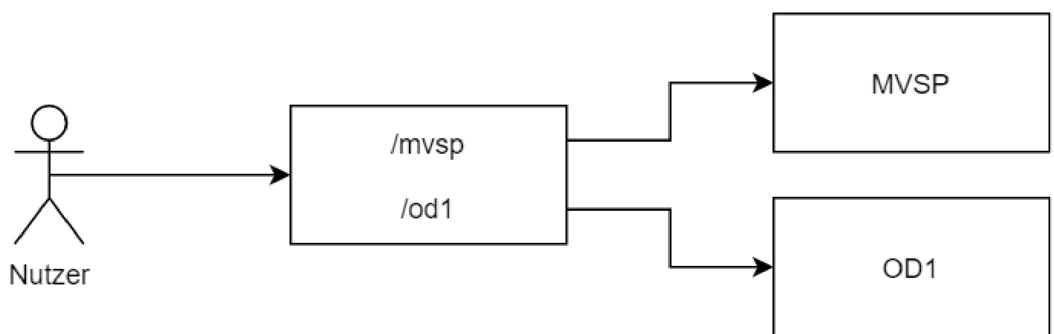

Bild 6.2: Versuchsaufbau UI-Fragment-Composition

Das MVSP nutzt Java Server Faces (JSF) zum Rendern der Seiten. JSF ermöglicht das Einbinden von HTML-Fragmenten aus Dateien oder anderen Pfaden. Ein Pfad kann auch eine URL sein. Um ein HTML-Fragment des Online-Dienstes einzubinden, ist der Code aus Listing 6.1 und 6.2 nötig: In Listing 6.1 wird ein Include-Statement definiert, welches HTML-Fragmente aus einem Pfad einbindet. In Listing 6.2 wird der definierte Ausdruck eingefügt.

```
<ui:define name="OD">
2    <ui:include src="/OD1" />
</ui:define>
```

Listing 6.1: Voruntersuchung: SSR Define Statement vorhabenBearbeiten.xhtml

```
<ui:insert name="OD" ></ui:insert>
```

Listing 6.2: Voruntersuchung: SSR Insert Statement layout.xhtml

Sollte der Online-Dienst (OD) nicht erreichbar sein, wird der Nutzer auf eine allgemeine Error-Seite weitergeleitet. Von dieser aus kann er wieder zu Vorhabenübersicht gelangen, die Nutzbarkeit ist aber eingeschränkt. Um dies zu verhindern, müsste eine Bean erstellt werden, welche die Verbindung zum Server prüft, bevor der Pfad eingebunden wird. Dies würde die Latenz weiter erhöhen.

Beim CSR tritt dieses Problem nicht auf. Sollte der OD nicht erreichbar sein, wird anstelle des OD eine Meldung eingeblendet, dass der Dienst zurzeit nicht erreichbar ist. Der Code hierfür ist komplexer, dies liegt aber hauptsächlich daran, dass in dieser Implementierung Fehler abgefangen werden. Für CSR ist der Code aus Listing 6.3, 6.4 und 6.5 nötig:

```
<ui:insert name="odLink" ></ui:insert>
[...]
<ui:insert name="OD" ></ui:insert>
```

Listing 6.3: Voruntersuchung: CSR Insert Statement layout.xhtml

```
<ui:define name="odLink">
   <script src="/OD1/static/main.js" type="text/javascript"></script>
</ui:define>

<ui:define name="OD">
   <div id='od'>
      <script>loadOD();</script>
   </div>
</ui:define>
```

Listing 6.4: Voruntersuchung: CSR Define Statement vorhabenBearbeiten.xhtml

```
function loadOD(){
   try{
      var xmlHttp = new XMLHttpRequest();
      xmlHttp.open( "GET", "/OD1", false );
      xmlHttp.send();
      document.getElementById("od").innerHTML = xmlHttp.responseText;
   }
   catch(e){
      document.getElementById("od").innerHTML = "Ein Fehler ist beim Laden des
          Dienstes aufgetreten, bitte versuchen Sie es spaeter erneut.";
   }
}
```

Listing 6.5: Voruntersuchung: CSR Define Statement /od1/static/main.js

Analog zum SSR wird mit Define- und Insert-Statements gearbeitet. In diesem Fall wird aber kein `<include>` verwendet, stattdessen wird ein JS-Funktionsaufruf in einem Platzhalter-Div eingefügt. Mit dieser Funktion wird der Online-Dienst in das Platzhalter-Div eingefügt. Zusätzlich wird ein Link zur „main.js" des Online-Dienstes eingebunden, diese kann z. B. Animation enthalten. Die entstehende Ansicht ist somit dieselbe, allerdings wird die Seite damit clientseitig gerendert.

6.3 Interaktion zwischen Client und Servern

Die Kommunikation zwischen Client und Server ist bei monolithischen Webapps leicht nachvollziehbar. Die Website liefert die HTML-Seite und verarbeitet eingehende Anfragen. Sobald diese Aufrufe aber nicht mehr den Ursprungsserver als Ziel haben, greifen Sicherheitsmechanismen, welche die Anfragen verhindern.

CORS erlaubt es dem Browser, Daten von einer anderen Quelle als dem Ursprungsserver zu beziehen. Dies kann, in einem Fall wie diesem, durchaus sinnvoll sein. CORS stellt aber auch ein Sicherheitsrisiko dar, da die bezogenen Daten oder Skripte Schadcode enthalten können. Daher ist die Unterstützung von CORS standardmäßig von Browsern und Webservern ausgeschaltet.

Um das Frontend ohne CORS mit mehreren Servern im Backend kommunizieren zu lassen, wird ein Reverse-Proxy benötigt. Der Reverse-Proxy vermittelt dem Browser den Eindruck, dass nur ein Server auf unterschiedlichen Pfaden aufgerufen wird, indem er die Anfragen zentral entgegennimmt und transparent an den zuständigen Server weiterleitet.

Der Testaufbau hierfür ist derselbe wie für den Test zu CSR in „Test verschiedener UI-Fragment-Composition-Konzepte", siehe Abbildung 6.1.

Der Code in Listing 6.6 erstellt einen Flask-Server[3], welcher unter `http://localhost:5000` erreichbar ist. Im Root-Kontext wird ein HTML-Fragment bereitgestellt. Unter der Route `/ranInt` wird eine Zufallszahl zurückgegeben. Diese Route wird vom Script unter `/script` aufgerufen. Dieser Aufruf ist nur dann erfolgreich, wenn die CORS-Richtlinie eingehalten wird.

Der einbettende Server ist unter `http://localhost:8000` erreichbar. Der Reverse-Proxy macht beide Server unter `http://localhost` erreichbar. Da der Proxy auf

[3]Flask ist ein Python WSGI-Mikro-Framework, mit welchem es möglich ist Web-Applikationen in Python zu programmieren.

Port 80 alle Server zusammenführt, können Fragmente des Dienstes unter `http://localhost:80` eingebunden werden, unter Port 8000 aber nicht, da unterschiedliche Ports als unterschiedliche Server gewertet werden und so gegen die CORS-Sperre verstoßen.

```
@app.route('/ranInt')
def ran_int():
    return str(random.randint(1, 1000000))
@app.route('/script')
    def script():'''
        function newInt(){
            var xmlHttp = new XMLHttpRequest();
            xmlHttp.open( "GET", "/ranInt", false );
            xmlHttp.send();
            document.getElementById("id1").innerHTML =  xmlHttp.responseText
        }
        '''
        return Response(js)
@app.route('/')
    def home():
        html = '''
            <div class="jumbotron">

                <a href="http://localhost:5000/redirect">redirect test</a>
                <div id='id1'> '''+ ran_int() +''' </div>
                    <button class="ui-button ui-widget ui-state-default ui-corner-all
                         ui-button-text-only none-jsf-style button-login margin-top-
                        login margin-right-lg"
                        type='button'
                        onclick='newInt()'>
                        new Int
                    </button>

                    <button class="ui-button ui-widget ui-state-default ui-corner-all
                         ui-button-text-only none-jsf-style margin-top-login margin-
                        right-lg test-button"
                        type='button'
                        onclick="changeColor()">
                        change color
                    </button>
                <script src="/script"></script>
            </div>
            '''
        return html
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, threaded=True, debug=1)
```

Listing 6.6: Voruntersuchung: Flask Server Code

6.4 Authentifizierung

Ein wichtiger Aspekt dieser Arbeit ist die Sicherheit. Im Folgenden wird ein zentrales Konzept erprobt, die Verwendung von Sidecars und eines zentralen Reverse-Proxys für die Service-zu-Service-Authentifizierung.

Wie bereits erläutert, sollen Client-Zertifikate für die Service-zu-Service-Authentifizierung genutzt werden. Um diese nicht bei jeder eingehenden Anfrage im Quellcode validieren zu müssen, wird diese Funktion in das Sidecar ausgelagert. Ebenso kann ein Sidecar zum Anreichern von Anfragen, von einem Dienst an einen anderen Dienst, um Zertifikate genutzt werden. Das Sidecar agiert also entweder als Reverse und Forward Proxy oder als zwei Reverse-Proxys. Der allgemeine Versuchsaufbau kann in Bild 6.3 gefunden werden. Für diesen Test wurden zwei Proxies untersucht: NGINX[4] und Apache[5].

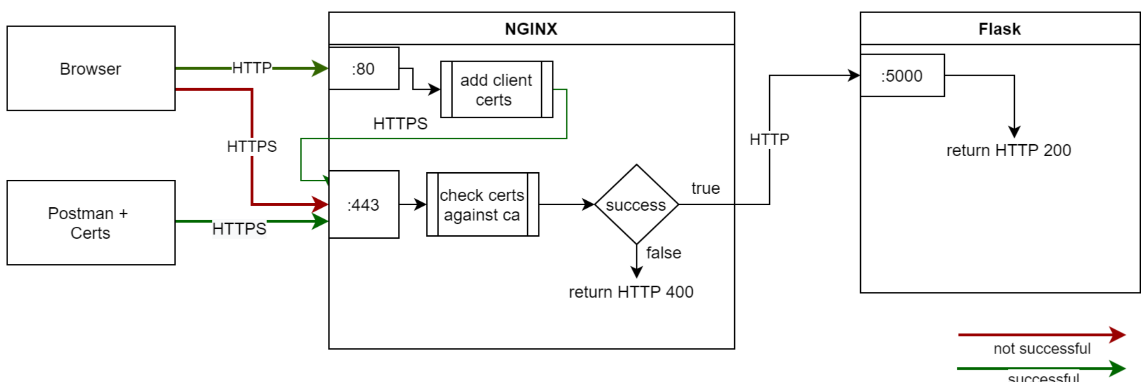

Bild 6.3: Versuchsaufbau Authentifizierung mit Client-Zertifikaten

Die verwendete NGINX Konfiguration ist in Listing 6.7 dargestellt.

NGINX wurde in Listing 6.7 nur als Reverse-Proxy , nicht als Forward Proxy verwendet. Soll NGINX als Forward Proxy genutzt werden, so muss Zeile 7 gegen Zeile 6 des Listing 6.7 getauscht werden. Durch diese Änderung agiert NGINX als Forward-Proxy, Forward und Reverse-Proxys funktionieren grundsätzlich unterschiedlich.

[4]NGINX ist ein weit verbreiteter Load-Balancer, Proxy und HTTP Server.
[5]Apache HTTPD, als Apache abgekürzt, ist ein HTTP Server, welcher auch als Proxy eingesetzt werden kann.

```
1   http {
     server {
3        listen      80;
         server_name client;
5        location /test {
             proxy_pass                       https://localhost:443/;
7            #proxy_pass                       https://$http_host$request_uri;
             proxy_ssl_certificate            ./certs/client.crt;
9            proxy_ssl_certificate_key        ./certs/client.key;
             proxy_ssl_trusted_certificate    ./certs/ca.crt;
11           proxy_ssl_server_name            off;
             proxy_ssl_verify                 off;
13       }
     }
15   server {
         listen      443 ssl;
17       server_name server;
         ssl_certificate        ./certs/server.crt;
19       ssl_certificate_key    ./certs/server.key;
         ssl_client_certificate ./certs/ca.crt;
21       ssl_verify_client      on;
         location / {
23           proxy_pass http://localhost:5000/;
         }
25   }
}
```

Listing 6.7: Voruntersuchung: NGINX Test mit Client-Zertifikaten

Bei der Verwendung von NGINX als Forward Proxy gibt es zwei Probleme:

- NGINX unterstützt nativ nicht den CONNECT Befehl
- Forward Proxies nehmen keine TLS-Terminierung / TLS-Re-Encryption vor

Die fehlende Unterstützung des CONNECT-Befehls ist problematisch, da es bedeutet, dass keine HTTPS-Ziele angesprochen werden können. Dieses Problem lässt sich mit einem NGINX Plug-In umgehen. Dies schließt die produktive Verwendung von NGINX aus, da es ein manuelles Kompilieren erfordert, was aufgrund des erschwerten Patch-Management nicht gestattet ist.

Das zweite Problem ist weniger einfach zu lösen. Eine Möglichkeit wäre die Verwendung eines Proxys, welcher eine TLS-Terminierung vornimmt, ein solcher befindet sich aber nicht in der Technologiematrix. Aufgrund des hohen Zeitaufwandes wurde vorerst dafür entschieden keinen Forward Proxy in die Technologiematrix aufzunehmen.

Aufgrund dieser Probleme kann NGINX nicht für die Implementierung genutzt werden.

Apache beherrscht den CONNECT-Befehl, nimmt aber als Forward-Proxy ebenfalls keine TLS-Terminierung vor. Daher werden stattdessen Reverse-Proxys für interne und externe Anfragen genutzt. Die Konfiguration der Sidecars kann in Listing A.2 gefunden werden und ist analog zur NGINX-Konfiguration. Der Versuchsaufbau hierfür ist in Bild 6.4 dargestellt.

Dieser Aufbau wurde getestet, indem mit einem Browser ohne Client-Zertifikate Port 80 des Servers aufgerufen wurde. Erwartungsgemäß war dies erfolgreich. Ein Aufruf von Port 443 schlug mit dem HTTP Error 400 fehl. Ein Aufruf desselben Ports mit dem Tool Postman und den nötigen Client-Zertifikaten war erfolgreich.

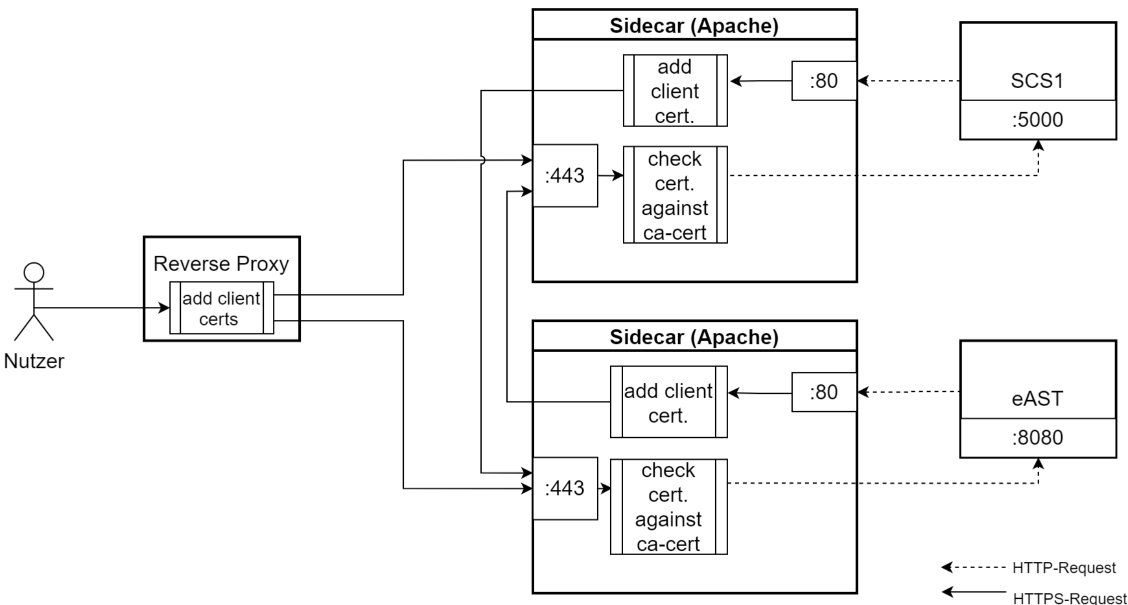

Bild 6.4: Versuchsaufbau Authentifizierung mit Client-Zertifikaten mit Apache

Als Reverse-Proxy wird ein Apache-Server genutzt. Ein NGINX Server wäre ebenso geeignet und performanter. Die Konfiguration für Sidecars und Reverse-Proxy sind sich ähnlich, daher ist es im Interesse des Betriebs, Apache als Reverse-Proxy zu nutzen.

Für diesen Versuch wurden angepasste Apache Images, unangepasste JDK-1.8 und Flask Images und die Docker-Compose File im Listing A.1 genutzt

Der Unterschied zum vorher durchgeführten Test liegt darin, dass alle Anfragen über einen zentralen Proxy laufen wodurch HTTP-Methoden blockiert werden können. Für das MVSP müssen POSTs und GETs erlaubt werden, alle anderen Methoden können verboten werden. Für das SCS1 müssen GETs und PUTs, entsprechend des Entwurfes, erlaubt werden.

```
<VirtualHost *:80>
    ProxyRequests off
    ProxyPreserveHost On
    SSLProxyEngine On
    SSLProxyCACertificateFile /certs/ca.crt
    SSLProxyMachineCertificateFile /certs/client.pem
    SSLProxyVerify none
    SSLProxyCheckPeerCN off
    SSLProxyCheckPeerName off
    SSLProxyCheckPeerExpire off
    <Location /eAST>
        ProxyPass https://sidecar_east/eAST
        ProxyPassReverse https://sidecar_east/eAST
      Require all granted
      AllowMethods GET POST
    </Location>
    <Location /od1>
        ProxyPass https://sidecar_od
        ProxyPassReverse https://sidecar_od
      Require all granted
      AllowMethods GET PUT
   </Location>
</VirtualHost>
```

Listing 6.8: Voruntersuchung: Apache Test mit Client-Zertifikaten

Für diesen Test wurde die Konfiguration, in Listing 6.8, für den Reverse-Proxy erstellt. Diese Konfiguration wurde mit Postman erfolgreich getestet. GET- und PUT- / POST-Requests wurden zugelassen, während alle anderen HTTP-Methoden mit HTTP Error 405 abgewiesen wurden.

6.5 Auswertung

In diesem Kapitel wurden, dieser Arbeit zentrale, Konzepte evaluiert. Hierbei wurde nicht nur gezeigt, dass diese Konzepte anwendbar sind, sondern auch wie genau sie anzuwenden sind. Begonnen wurde mit dem Vergleich verschiedener UI-Fragment-Composition-Konzepte, wobei Client-Side-Rendering überzeugte. Des Weiteren wurde erprobt, wie eine sichere Kommunikation des Clients mit den Servern möglich ist. Zuletzt wurde die symmetrische Authentifizierung der Dienste mittels Sidecars evaluiert. Hierbei zeigte sich, dass Apache als Proxy-Lösung besser geeignet ist, als NGINX.

Die in diesem Kapitel gesammelten Erkenntnisse können in der Implementierung des Entwurfes zur Anwendung gebracht werden.

7 Implementierung

Nach der Erprobung der Konzepte im vorherigen Kapitel wird in diesem Kapitel die Erweiterung des MV-Serviceportals beschrieben. Die vorigen Kapitel wurden am Beispiel des MVSP erklärt, die Erkenntnisse aus ihnen können aber auf beliebige Projekte angewandt werden. Das folgende Kapitel dient der Erläuterung der Implementierung speziell für das MVSP.

In Kapitel 5.1 wurde bereits die angestrebte Architektur beschrieben. In Bild 7.1 sind nur die in diesem Kapitel beschriebenen Komponenten und ihre Schnittstellen aufgeführt. Der Aufbau aus Bild 7.1 wurde mit der Docker-Compose Konfiguration in Listing A.1 umgesetzt.

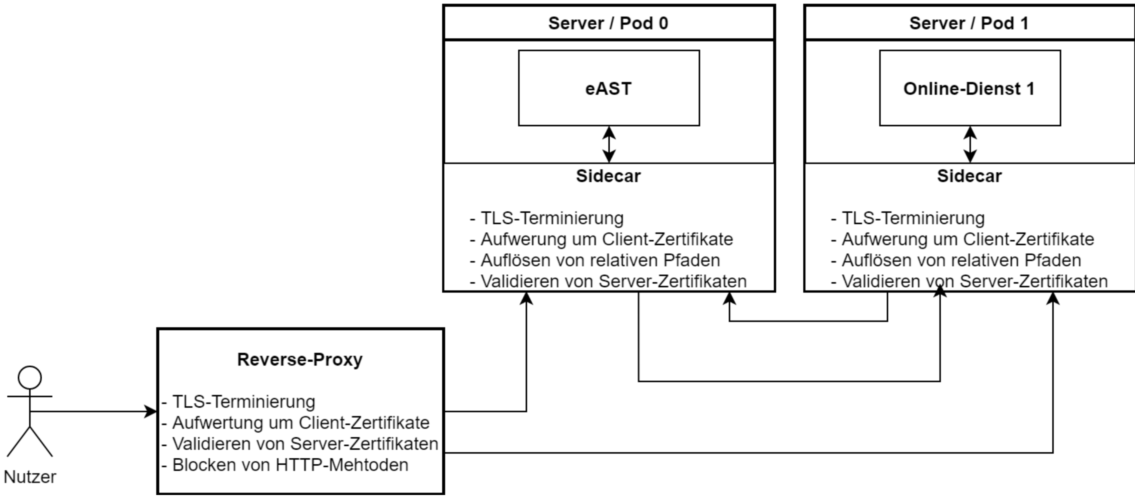

Bild 7.1: Vereinfachte Architektur

7.1 Online-Dienst-Prototyp

In diesem Unterkapitel wird die spezifische Implementierung eines Online-Dienstes entsprechend des generischen Entwurfes erläutert. Der Prototyp zeigt die Umsetzbarkeit des Entwurfes und der Erfüllbarkeit der definiert Anforderungen.

Die eAST ist Java-basiert. Durch die Verwendung von Self-Contained-Systems ist es nicht nötig, alle Dienste in derselben Programmiersprache oder mit denselben Frameworks zu implementieren. Daher ist der Prototyp in Python geschrieben und verwendet Flask und Flask-RESTful als Frameworks für die Bereitstellung des Webservers und der API. Python ist eine interpretierte Sprache mit Fokus auf Menschenlesbarkeit, welche sich für Rapid-Prototyping eignet, daher ist diese Sprache für die Verwendung in dieser Arbeit geeignet [29].

Der Prototyp zeigt alle Daten an, die in der eAST und im Dienst selbst vorliegen. Der Dienst speichert lokal nur verfahrensspezifische Daten. Das Ausgeben und Ändern der Daten erfolgt über eine REST-API des Online-Dienstes. Diese API wird von JavaScript-Funktionen im Frontend zur Darstellung und Änderung der Daten und von der eAST im Backend zur Erstellung und Löschung von Partial-Applications aufgerufen.

Damit das Backend des Online-Dienstes die Anfragen bearbeitet, muss eine Session-ID übergeben werden. Diese wird mit jedem Aufruf einem Cookie entnommen, den die eAST angelegt hat.

Somit wurden alle geschaffenen Schnittstellen des OD genutzt und die Funktionalität belegt.

7.1.1 OD-API

Die API wurde entsprechend des Entwurfes implementiert. Eine ausführliche Dokumentation der API im OpenAPI3 Format kann im digitalen Anhang gefunden werden.

Der folgende Code Ausschnitt erstellt einen Webserver und API:

```
1   app = Flask(__name__)
    api = Api(app)
3
    api.add_resource(Vorhaben,'/api/v1/vorhaben/<string:id>')
5
    @app.route("/")
7   def index():
        """serve the ui"""
9       return render_template("index.html")
```

Listing 7.1: Implementierung: Flask Server mit API

Im Root-Kontext wird das statische HTML bereitgestellt. Die API erwartet ein Objekt pro Endpoint. Dieses Objekt besitzt Funktionen für die unterstützten HTTP-Methoden. In diesem Fall ist das Objekt vom Typ Vorhaben und besitzt Funktionen für POST-, GET-, PUT- und DELETE-Requests, wobei eine leere Anfrage auf die Ressource nicht möglich ist. Es muss immer eine ID vom Typ String übergeben werden. Die ID ist hierbei die Partial-Application-ID.

POST Anhand der Partial-Application-ID im Pfad wird lokal ein Eintrag mit dieser ID als Key angelegt.

GET Die GET-Funktion erwartet eine Session-ID als Query-Parameter. Ist diese nicht vorhanden, oder kann diese nicht gegen die eAST validiert werden, wird die Anfrage mit dem HTTP Status Code 401 zurückgewiesen. Kann sie validiert werden, wird eine Anfrage an die eAST mit der Partial-Application-ID und der Session-ID durchgeführt. Ist diese erfolgreich, gibt dieser Aufruf ein JSON-Objekt zurück, welches zu einem Python Dictionary konvertiert wird. Dieses wird mit den abgerufenen lokalen Daten zusammengeführt und als JSON-Objekt im „data”-Teil der Response zurückgegeben.

PUT Die PUT-Funktion erwartet ein JSON-Objekt im Body der Request mit den Schlüsseln: „reason” und „note” , wobei „reason” und „note” die verfahrensspezifischen Daten darstellen. Zusätzlich wird eine Session-ID als Query-Parameter erwartet. Ist die Validierung der Session-ID erfolgreich, werden die geänderten Daten lokal gespeichert.

DELETE Die DELETE-Funktion benötigt nur die ID im Pfad. Kann ein Objekt mit dieser ID gefunden und gelöscht werden wird HTTP Status Code 204 zurückgegeben, andernfalls der Code 404.

7.2 eAST

Um ein SCS einbinden zu können, muss die eAST nicht umgebaut werden. Die Erweiterung um einen Spezialfall ist ausreichend. Dies ist wichtig, um bekannte Abläufe beizubehalten.

Die eAST muss unterscheiden, ob die Partial-Application einen domänenspezifischen Online-Dienst benötigt, oder Formulare des FMS nutzt. Dieser Ablauf ist in Abbildung 7.2 dargestellt. Diese Unterscheidung wird anhand einer gesetzten booleschen

Variable im IDMV getroffen. Ist „`is_specific`” `True`, wird das gerenderte HTML-Template um weitere Einschübe zum Einbinden des Online-Dienstes ergänzt. In diesem Fall wird eine JS-Datei anstelle des Standardformularassistenten eingebunden. Der genaue Ablauf wird im Folgenden erläutert.

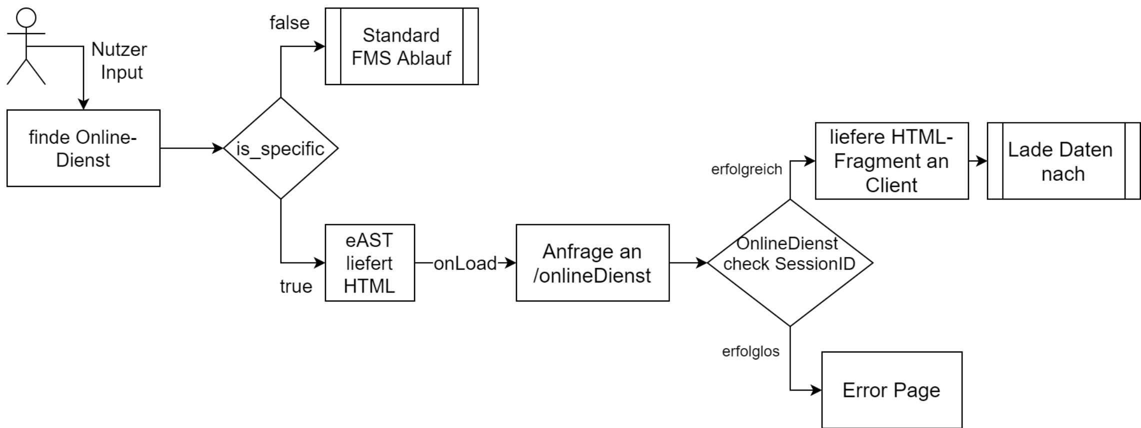

Bild 7.2: Flowchart Einbinden eines Online-Dienstes

7.2.1 UI-Fragment Composition

Zum Ladezeitpunkt der Seite wird die Funktion `loadOD1()` ausgelöst, welche eine JS-Funktion des Online-Dienstes auslöst. Die JS-Funktion `loadOD()` wird vom Online-Dienst bereitgestellt, siehe Listing 7.3. Sie enthält die applikationsspezifische Logik zum Laden der Daten.

```
function loadOD1(){
    try{
        loadOD();
    }
    catch(e){
        document.getElementById("od1").innerHTML = "Ein Fehler ist beim Laden des
            Dienstes aufgetreten, bitte versuchen Sie es spaeter erneut."
    }
}
```

Listing 7.2: Implementierung: eAST-JS-Code zum einbetten des OD

Die eAST erwartet, dass eine Datei `main.js` im Static-Kontext des Online-Dienstes existiert, welche eine Funktion `loadOD()` enthalten muss. Der Online-Dienst hingegen erwartet, dass ein Platzhalter-Div mit der ID `od1` an der korrekten Stelle im HTML-Dokument vorliegt. Nur wenn beide Erwartungen erfüllt sind, wird der Online-Dienst korrekt geladen.

Ist dies der Fall, ist der Online-Dienst nach dem Einbinden relativ autonom. Funktionen und Animationen können spezifisch für jeden Online-Dienst sein. Die Online-Dienste können sich aber auch am Corporate-Design der einbindenden Applikation orientieren, indem sie z. B. vordefinierte CSS-Klassen nutzen.

```
function loadOD(){
    try{
        var xmlHttp = new XMLHttpRequest();
        xmlHttp.open( "GET", "/od1", false );
        xmlHttp.send();
        document.getElementById("od1").innerHTML = xmlHttp.responseText;
        loadData();
    }
    catch(e){
        document.getElementById("od1").innerHTML = "Ein Fehler ist beim Laden
            des Dienstes aufgetreten, bitte versuchen Sie es spaeter erneut.";
    }
}
```

Listing 7.3: Implementierung: OD-JS-Code zum einbetten des OD

7.2.2 eAST-API

Die API wurde entsprechend des Entwurfes implementiert. Für die Implementierung wurde „Java API for RESTful Web Services” (jaxrs)[30] genutzt. Eine ausführliche Dokumentation der API im OpenAPI2 Format kann im Anhang gefunden werden.

Im Folgenden wird darg-gestellt, wie genau der Entwurf implementiert wurde. Um die Nutzung der API zu vereinfachen, wird der Request Body nicht in ein statisches Objekt geparst. Beim Parsen in ein statisches Objekt kann es schnell zu Fehlern kommen. Zu diesem Fehlerfällen gehören: die Datentypen sind nicht korrekt, es wurde nicht das vollständige Objekt übergeben oder es wurde ein Objekt mit zu vielen Attributen übergeben. Diese Fehlerfälle können mit Serialisierungsklassen abgefangen werden, die Verwendung von Maps ist jedoch eine flexiblere Lösung für das Problem. Daher werden Maps mit Strings als Schlüsseln und Objekten als Werten verwendet.

Vorhaben Vorhaben werden über ihre Universally Unique Identifier (UUID) referenziert. Der eingebettete Online-Dienst hat aber keine Möglichkeit diese auszulesen, da der einzige Kontaktpunkt des Online-Dienstes mit der eAST im Frontend ist. Dort ist die ID aus Sicherheitsgründen nur verschlüsselt in der URL verfügbar. Wird ein Vorhaben angefragt, muss daher zuerst geprüft werden, ob die angegebene ID eine UUID oder verschlüsselt ist. Diese Überprüfung wird mit einem regulären Ausdruck in Listing 7.4 durchgeführt.

```
^[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}$
```

Listing 7.4: Implementierung: Regulärer Ausdruck zum Validieren einer ID

Ist die übergebene ID keine UUID, wird davon ausgegangen, dass ein verschlüsselte ID vorliegt. Diese wird entschlüsselt und wieder überprüft. Liegt immer noch keine UUID vor, wird die Anfrage abgewiesen. Liegt eine UUID vor wird diese validiert. Ist dies erfolgreich, wird die übergebene SessionID überprüft. Damit die Anfrage weiter bearbeitet wird, muss der Nutzer mit der Session auch die angefragte Partial-Application besitzen. Ist dies der Fall, wird die methodenspezifische Logik ausgeführt.

GET Sind die Vorbedingungen erfüllt, wird das Vorhabenobjekt in einen JavaScript Object Notation (JSON) String umgewandelt und als Value zum Data Attribut im Response Body zurückgegeben.

PUT Sind die Vorbedingungen erfüllt, wird ein Partial-Application-Objekt aus der Datenbank erzeugt. Die übergebenen Attribute werden anschließend im Objekt ersetzt und das neue Objekt wird in der Datenbank hinterlegt.

Postfach Das Postfach ist die einzige Ressource, welche den Request Body in ein statisches Objekt parst, da die Nachrichten nur erstellt und nie geändert werden. Im Fall des Postfaches müssen immer vier Attribute gegeben sein: die Partial-Application-Id, Public-Partner-Key (die ID der zuständigen Stelle), der Titel und der Nachrichteninhalt.

POST Nachdem der Request Body in das `inBoxRequest`-Objekt geparst wurde, werden die übergebenen IDs validiert. Ist dies erfolgreich, wird eine neue Nachricht in der Datenbank erstellt.

Session Diese Ressource dient nur dem grundsätzlichen Validieren, ob eine Session existiert und ob der Nutzer mit dieser Session eingeloggt ist. Es werden keine Daten ausgegeben, da alle globalen Daten ohnehin nur von der eAST verwaltet werden. Es muss bei jeder Anfrage des Online-Dienstes eine Session-ID angeben werden. Die Überprüfung, ob der Nutzer berechtigt ist, diese Ressource anzufragen, läuft intern ab. Hiermit dient die eAST als zentraler Session-Speicher.

GET Existiert eine Session mit der übergebenen ID und ist der Nutzer mit dieser ID eingeloggt, wird ein HTTP Status Code 200 und ein Menschenlesebarer Response Body zurückgegeben.

7.3 Deployment

Das Deployment entspricht Bild 7.1. Applikation und Sidecar sollten, abhängig von der Infrastruktur, immer auf demselben Server, im selben Container oder Pod deployt werden, da die Kommunikation zwischen Sidecars und Applikation unverschlüsselt abläuft. Aus demselben Grund darf zwischen Sidecar und Applikation nur Port 80 freigegeben werden, insofern ein Docker basiertes Deployment genutzt wird. Andernfalls müssen Sidecar und Applikation auf demselben Server deployt werden. Extern darf nur Port 443 für HTTPS-Traffic offen sein.

In der Konfiguration des Reverse-Proxys können die Erkenntnisse des Proof of Concept genutzt werden und unerwünschte HTTP-Methoden blockiert werden. Daraus ergibt sich die Konfiguration aus Listing A.2 für die Sidecars und die Konfiguration aus Listing 6.8 für den Reverse-Proxy.

Für dieses Projekt wurde mit Docker gearbeitet, daher wurde das Deployment aus Grafik 7.1 mit Docker-Compose umgesetzt, die Konfiguration hierfür befindet sich in Listing A.1.

Da das MVSP eine vollständige Docker-basierte Continous-Integration und Delivery Pipeline besitzt, welche OpenShift als Umgebung für das Test-Deployment nutzt, wurde zusätzlich ein OpenShift Template angefertigt. Dieses befindet sich im digitalen Anhang. Hiermit ist es möglich, den angefertigten Prototypen die bestehende Test-Pipeline durchlaufen zu lassen.

8 Tests

In diesem Kapitel werden die Ergebnisse zu Tests der Erfüllung der Anforderungen, Entkopplung der Komponenten und zur generellen Benutzbarkeit und Funktionalität vorgestellt.

8.1 Erfüllung der Anforderungen

In diesem Kapitel wird die Umsetzung der anfangs definierten Anforderungen untersucht.

Lose Kopplung Fällt ein Online-Dienst aus, wird der Nutzer zwar darüber informiert kann den Rest des MVSP aber ungehindert weiter nutzen. Siehe Kapitel 6.2.

TLS Durch die Sidecars erfolgt auch die interne Netzwerkkommunikation über HTTPS-Verbindungen. Bestätigt wurde dies durch die Tests im Kapitel 6.4.

Service-Authentifizierung Durch die Sidecars werden netzwerkinternen Anfragen immer Client-Zertifikate angehängt und alle eingehenden Anfragen auf Client-Zertifikate überprüft, wie in Kapitel 6.4 getestet.

Nutzung vorhandener Dienste Die Daten zum Online-Dienst werden dem IDMV entnommen.

Technologiematrix Der Entwurf kann problemlos ausschließlich mit Technologien der Technologiematrix umgesetzt werden. Der Prototyp wurde mit Python und Flask implementiert, daher kann dieser nicht produktiv deployt werden. Das ist aber auch nicht gewünscht.

OZG-konform Die OZG-Referenzarchitektur [4] macht keine Aussage zur genauen Umsetzung der Antragstellung, daher kann es auch nicht zu Konflikten kommen.

Somit sind alle definierten Anforderungen an die Architektur erfüllt.

8.2 Abgrenzungstest

Ein Schwerpunkt dieser Arbeit bestand darin, den hohen Sicherheitsanforderungen gerecht zu werden. Dieser Test soll sicherstellen, dass die Dienste nur über die geplanten Methoden erreichbar sind. Dies wurde mit dem Code-Ausschnitt in Listing 8.1 getestet. Die IP-Adresse des Servers auf dem die Applikation deployt wurde ist 'xx.xx.xx.xx'.

```
responseObj = []
routes = ["http://xx.xx.xx.xx/", "http://xx.xx.xx.xx/od1", "http://xx.xx.xx.xx/
    api/v1/vorhaben/1", "http://xx.xx.xx.xx/od1/api/v1/vorhaben/1"]
for route in routes:
    results = {}
    try:
        results["get"] = requests.get(route, verify=False).status_code
        results["post"] = requests.post(route, verify=False).status_code
        results["put"] = requests.put(route, verify=False).status_code
        results["patch"] = requests.patch(route, verify=False).status_code
        results["delete"] = requests.delete(route, verify=False).status_code
        results["head"] = requests.head(route, verify=False).status_code
        results["options"] = requests.options(route, verify=False).status_code

    except Exception as e:
        print(e)
    jsonObj = {"route": route, "results": results}
    responseObj.append(jsonObj)
with open('data.json', 'w') as f:
    json.dump(responseObj, f)
```

Listing 8.1: Tests: Test der Methodenbehandlung

Diese Funktion speichert die HTTP-Codes aller HTTP-Methoden gegen die definierten Routen. Es ist zu erwarten, dass gegen die Route der eAST nur GET und POST erlaubt ist. Der Online-Dienst sollte nur GET und PUT unterstützen, da nur diese Funktionen extern erreichbar sein sollten. Die Anfragen an die API-Pfade sollten alle mit 405, 404, 403 oder 401 Codes Abgewiesen werden.

Die Ergebnisse in Listing A.5 bestätigen dies. Der Test wird somit als Erfolg gewertet.

8.3 Ladezeiten und Latenz

Die Modularität und Funktionalität der Implementierung mag für Entwickler vorteilhaft sein, schlussendlich darf sie aber die Nutzererfahrung nicht negativ beeinflussen. Dieser Test dient dem Vergleich der Ladezeiten, eines neuen Online-Dienstes

und des Standardantragsassistenten. Vorweg ist zu sagen, dass die genauen Ergebnisse je nach Implementierung und Umfang des Online-Dienstes variieren werden. Der implementierte Prototyp nutzt alle Schnittstellen der eAST, ruft aber keine externen Schnittstellen auf.

Für diesen Test wurde das Google Chrome Plug-In „Page load time” verwendet. Die Tests wurden auf einem Lenovo T470s mit einem Intel I5 CPU und 8 Gb RAM durchgeführt, wobei eine Latenz zum Server von circa 2ms existierte. Jede Messung wurde fünf Mal durchgeführt, anschließend wurde der Durchschnitt berechnet. In den Messungen wurde unterschieden zwischen der Zeit, die zum Beantworten der Anfrage benötigt wurde und der Zeit, die insgesamt benötigt wurde bis die Seite nutzbar wurde. Die Testergebnisse können im Anhang in Listing A.3 gefunden werden.

Der Standardantragsassistent benötigt durchschnittlich 1,06 Sekunden zum Laden der Seite, wobei der Server 0,409 Sekunden benötigt, um die Anfrage zu beantworten. Der Online-Dienst hingegen benötigte durchschnittlich 1,03 Sekunden zum Laden der Seite und nur 0,384 Sekunden zum Beantworten der Anfrage.

Zum Beantworten der Anfrage benötigt der Online-Dienst wenig Zeit, da die Ansicht teilweise clientseitig gerendert wird. Der Standardantragsassistent hingegen wird komplett serverseitig unter Verwendung von JSF gerendert.

Dennoch könnte erwartet werden, dass der Online-Dienst länger zum Laden braucht, da dieser mehrere Anfragen an die eAST stellen muss. Daher wurde ebenfalls die Zeit zum Laden des Online-Dienstes gemessen. Hierzu wurde der JS-Code aus Listing 8.2 genutzt.

```
1     function timeIt(){
          for(let i = 0; i < 50; i++){
3             console.time("load");
              loadOD1();
5             console.timeEnd("load");
          }
7     }
```

Listing 8.2: Tests: Benchmark der Funktion zum Laden des Online-Dienstes

Tatsächlich wird zum Laden des statischen HTML und dem Nachladen der Daten durchschnittlich nur 35 ms benötigt. Dies ist hauptsächlich damit zu erklären, dass sowohl der Online-Dienst als auch die eAST keiner nennenswerten Last ausgesetzt waren und auf demselben Server ausgeführt wurden und daher fast ohne Latenz kommunizieren konnten.

Die Ergebnisse dieses Tests belegen die Nutzbarkeit des Prototypen und zeigen, dass das umgesetzte Konzept auch aus Sicht der Nutzbarkeit grundsätzlich geeignet ist.

8.4 Latenz durch Zertifikate

Die Verwendung von Sidecars und Client-Zertifikaten ist mit Kosten verbunden. Jede beteiligte Komponente muss einen Teil der Anfrage entpacken und weitersenden. Je mehr Komponenten an einer Anfrage beteiligt sind, desto länger ist die Antwortzeit. Dieser Test soll der Bestimmung des Overheads durch die Verwendung von Client-Zertifikaten dienen. Die Testergebnisse können im Anhang in Listing A.4 gefunden werden.

Es wurden zwei Testdurchläufe mit je 50 Messungen durchgeführt. Die durchschnittliche Antwortzeit ohne Sidecars beträgt 1,4065 ms, werden Sidecars mit Client-Zertifikaten und HTTPS verwendet steigt die Antwortzeit auf 3,2572 ms. Die Messungen wurden in einem Docker-Container unter CentOS durchgeführt. Es wurde `time.time()` verwendet, diese Funktion besitzt eine Präzision von ca. 239 ns [31].

Die benötigte Zeit unter Verwendung der Reverse-Proxys ist durchschnittlich höher als die Zeit ohne. Der Overhead beträgt circa 132% nach der folgenden Rechnung:

$$(3.2572 - 1.4065) * 100/1.4065 = 131.58\%$$

Die Verwendung eines Reverse-Proxys mit Client-Zertifikaten scheint einen hohen prozentualen Einfluss auf die Latenz zu haben. Allerding ist hier anzubringen, dass der Overhead immer noch bei nur circa 2 ms liegt. Der Anteil ist prozentual nur deshalb so hoch, da keine Netzwerk Latenz vorliegt.

Wird eine Latenz von 25ms vom Client zum Server angenommen, liegt der prozentuale Overhead bei ca 4%.

$$(53.2572 - 51.4065) * 100/51.4065 = 3.6\%$$

Diese zusätzliche Latenz ist für einen Menschen nicht wahrnehmbar [32] und daher akzeptabel.

8.5 Auswertung

In diesem Kapitel wurde überprüft, ob die anfangs definierten Anforderungen an die Architektur erfüllt wurden. Dies ist der Fall.

Anschließend wurden alle HTTP-Methoden gegen diverse Routen getestet um sicherzustellen, dass die korrekten Fehler zurückgegeben werden und dass die Konfiguration der Proxys korrekt ist. Dies ist der Fall.

Abschließend wurden weitere Tests zur Responsivität der neuen Architektur durchgeführt und die Ergebnisse mit der Ausgangsarchitektur verglichen. Hierbei war kein Nachteil der neuen Architektur wahrnehmbar.

Somit können die Tests als erfolgreich gewertet werden, die Ziel- und Aufgabenstellung ist hiermit erfüllt.

9 Fazit und Ausblick

In dieser Arbeit wurde die Erweiterung der elektronischen Antragstellung des MV-Serviceportals untersucht. Ziel dieser Untersuchung war es, Möglichkeiten und Konzepte für die Verwendung von Self-Contained-Systems zu erarbeiten. Der verfolgte Service-Mesh-Ansatz erwies sich als sehr gut geeignet um dieses Ziel unter Einhaltung der Anforderungen zu erreichen.

In dieser Arbeit wurde nicht nur gezeigt, dass die Verwendung von Self-Contained-Systems möglich ist, sondern auch wie dies möglich ist. Der angefertigte Entwurf für die Schnittstellen eines Online-Dienstes ist darauf ausgelegt generisch genug sein, um von beliebigen Applikationen implementiert zu werden.

Sollte das DVZ in der Zukunft Kubernetes produktiv einsetzen, ist der implementierte Ansatz erweiterbar. Die Sidecars des Service-Meshes könnten durch „Istio”-Container ersetzt werden. Istio[33] ist eine Cloud-Nativ-Lösung für Service-Meshes, somit bietet es sich für die Verwendung in einem Kubernetes-Cluster an. Mit Istio ist eine bessere Überwachung und Konfiguration aller Dienste möglich.

In der Zukunft könnten Dienste wie das Postfach ausgelagert werden, sollte dies nötig werden. Ebenso könnte die Ausgliederung der Datenhaltung untersucht werden. Sollte es nötig werden das MVSP horizontal zu skalieren, muss der Session-Speicher ebenfalls ausgegliedert werden, um ein effizientes Skalieren zu ermöglichen. Bevor dieser Entwurf produktiv eingesetzt werden kann, müssen Prozesse für die sichere Erstellung, Verwaltung und Übertragung der verwendeten Zertifikate geschaffen werden, da in dieser Arbeit nur selbst generierte Zertifikate verwendet wurden.

In der Zukunft könnten die „einfache Melderegisterauskunft” und eine neue Zahlungsverkehrsplattform nach diesem Entwurf implementiert werden.

Bei der Weiterentwicklung des MV-Serviceportals können Self-Contained-Systems eine tragende Rolle spielen, um die zukünftigen Anforderungen, wie Skalierbarkeit und Erweiterbarkeit zu erfüllen.

Literaturverzeichnis

[1] *Anteil der Internetnutzer in Deutschland in den Jahren 2001 bis 2018.* https://de.statista.com/statistik/daten/studie/13070/umfrage/entwicklung-der-internetnutzung-in-deutschland-seit-2001/, Abruf: 25. Januar 2020

[2] *Gesetz zur Verbesserung des Onlinezugangs zu Verwaltungsleistungen (Onlinezugangsgesetz - OZG).* https://www.gesetze-im-internet.de/ozg/BJNR313800017.html, Abruf: 10. Dezember 2019

[3] Stranghörner, Roman: *Self Contained Systems.* https://scs-architecture.org, Abruf: 6. August 2019

[4] *Die Architektur des Portalverbunds.* https://www.it-planungsrat.de/DE/ITPlanungsrat/OZG-Umsetzung/Portalverbund/01_Architektur/Architektur_node.html, Abruf: 15. Januar 2020

[5] *The electronic Functions.* https://www.personalausweisportal.de/EN/Citizens/German_ID_Card/Functions/Functions_node.html, Abruf: 19. Februar 2020

[6] *Welcome to OpenID Connect.* https://openid.net/connect/, Abruf: 5. Februar 2020

[7] *Aktivität 11 – Basiskomponenten und Dienste.* https://www.regierung-mv.de/serviceassistent/download?id=1593251, Abruf: 14. Dezember 2019

[8] *Chapter 5. Configuring Contexts.* https://www.eclipse.org/jetty/documentation/current/configuring-contexts.html, Abruf: 5. Februar 2020

[9] *Spring Boot Change Context.* https://www.baeldung.com/spring-boot-context-path, Abruf: 5. Februar 2020

[10] *Best Architecture for an MVP.* https://rubygarage.org/blog/monolith-soa-microservices-serverless, Abruf: 6. Februar 2020

[11] *Pattern: Monolithic Architecture.* https://microservices.io/patterns/monolithic.html, Abruf: 9. Januar 2020

[12] *What Is SOA?* https://web.archive.org/web/20160819141303/http://opengroup.org/soa/source-book/soa/soa.htm, Abruf: 9. Januar 2020

[13] *Microservices.* https://web.archive.org/web/20180214171522/https://martinfowler.com/articles/microservices.html, Abruf: 9. Januar 2020

[14] Nadareishvili I. Mitra, M. Amundsen M. R. McLarty M. R. McLarty: *Microservice Architecture: Aligning Principles, Practices, and Culture.* 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA: O'Reilly Media, Inc., 2016

[15] *Microservices: Architecting for Continuous Delivery and DevOps.* https://www.researchgate.net/publication/323944215_Microservices_Architecting_for_Continuous_Delivery_and_DevOps, Abruf: 9. Januar 2020

[16] Newman, Sam: *Building Microservices.* First Edition. 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA: O'Reilly Media, Inc., 2015

[17] *Richardson Maturity Model.* https://martinfowler.com/articles/richardsonMaturityModel.html, Abruf: 9. Januar 2020

[18] *Request Methods.* https://tools.ietf.org/html/rfc7231#section-4, Abruf: 31. Dezember 2019

[19] *The PATCH Method.* https://tools.ietf.org/html/rfc5789#section-2, Abruf: 31. Dezember 2019

[20] *From Monolith to micro-services and back : The Self Contained Systems.* https://www.slideshare.net/rdebusscher/from-monolith-to-microservices-and-back-the-self-contained-systems, Abruf: 31. Januar 2020

[21] Mouat, Adrian: *Docker, Software entwickeln und deployen mit Containern.* Erste Auflage. dpunkt.verlag GmbH, Wieblinger Weg 17, 69123 Heidelberg: dpunkt Verlag, 2016

[22] *Kubernetes and Container Security and Adoption Trends.* https://www.stackrox.com/kubernetes-adoption-and-security-trends-and-market-share-for-containers/, Abruf: 17. Januar 2019

[23] *Kubernetes Documentation.* https://kubernetes.io/docs/home/?path=users&persona=app-developer&level=foundational, Abruf: 9. Januar 2020

[24] *Fallacies of distributed computing.* https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing, Abruf: 24. Dezember 2019

[25] *Welche Schritte sind notwendig um ein Zertifikat zu validieren / zu prüfen?* http://www.globaltrust.eu/php/cms_monitor.php?q=PUB&s=20446cuj, Abruf: 9. Januar 2020

[26] *The OpenAPI Specification.* https://www.openapis.org/, Abruf: 5. Februar 2020

[27] *XFall.* http://www.xfall.eu/, Abruf: 26. Januar 2020

[28] *Server-Side Rendering (SSR) vs Client-Side Rendering (CSR).* https://pagepro.co/blog/2019/11/13/ssrvscsr/, Abruf: 22. Februar 2020

[29] *Python (programming language).* https://en.wikipedia.org/wiki/Python_(programming_language), Abruf: 22. Februar 2020

[30] *JSR 311: JAX-RS: The JavaTM API for RESTful Web Services.* https://jcp.org/en/jsr/detail?id=311, Abruf: 22. Februar 2020

[31] *PEP 564.* https://www.python.org/dev/peps/pep-0564/#linux, Abruf: 15. Januar 2020

[32] Sacher, Hartl Fritz in F. Fritz in Fucik: *Handbuch des Verkehrsunfalls, Teil 2*. Erste Auflage. : Manz-Verlag Wien, 2008

[33] *Istio.* https://istio.io/, Abruf: 18. Februar 2020

[34] Brenda Jin, Amir S. Saurabh Sahni S. Saurabh Sahni: *Designing Web APIs: BUILDING APIS THAT DEVELOPERS LOVE*. First Edition. 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA: O'Reilly Media, Inc., 2018

[35] *Elektronischen Gerichts- und Verwaltungspostfach EGVP.* https://egvp.justiz.de/, Abruf: 14. Dezember 2019

[36] *Patterns for Microservices — Sync vs. Async.* https://dzone.com/articles/patterns-for-microservices-sync-vs-async, Abruf: 26. Januar 2020

[37] Prinz, Hanna: *Brauchen asynchrone Microservices und Self-Contained Systems ein Service-Mesh?* https://www.heise.de/developer/artikel/Brauchen-asynchrone-Microservices-und-Self-Contained-Systems-ein/-Service-Mesh-4478057.html?seite=all, Abruf: 6. August 2019

Abbildungsverzeichnis

2.1 Ist-Architektur 8

4.1 Gegenüberstellung verschiedener Architekturen in Anlehnung an [10] 12
4.2 SCS basierte Online-Plattform [16] 15

5.1 Angestrebte Architektur 25
5.2 Sequenzdiagramm: Erstellen einer Application 29
5.3 Sequenzdiagramm: Lesen und Ändern einer Application 31
5.4 Sequenzdiagramm: Löschen einer Application 32

6.1 Vergleich der Ladezeiten CSI vs SSI 35
6.2 Versuchsaufbau UI-Fragment-Composition 35
6.3 Versuchsaufbau Authentifizierung mit Client-Zertifikaten 39
6.4 Versuchsaufbau Authentifizierung mit Client-Zertifikaten mit Apache 41

7.1 Vereinfachte Architektur 43
7.2 Flowchart Einbinden eines Online-Dienstes 46

Listings

6.1 Voruntersuchung: SSR Define Statement vorhabenBearbeiten.xhtml . 35
6.2 Voruntersuchung: SSR Insert Statement layout.xhtml 36
6.3 Voruntersuchung: CSR Insert Statement layout.xhtml 36
6.4 Voruntersuchung: CSR Define Statement vorhabenBearbeiten.xhtml . 36
6.5 Voruntersuchung: CSR Define Statement /od1/static/main.js 36
6.6 Voruntersuchung: Flask Server Code 38
6.7 Voruntersuchung: NGINX Test mit Client-Zertifikaten 40
6.8 Voruntersuchung: Apache Test mit Client-Zertifikaten 42

7.1 Implementierung: Flask Server mit API 44
7.2 Implementierung: eAST-JS-Code zum einbetten des OD 46
7.3 Implementierung: OD-JS-Code zum einbetten des OD 47
7.4 Implementierung: Regulärer Ausdruck zum Validieren einer ID 48

8.1 Tests: Test der Methodenbehandlung 51
8.2 Tests: Benchmark der Funktion zum Laden des Online-Dienstes . . . 52

A.1 Implementierung: Docker-Compose.yaml 65
A.2 Sidecar apache httpd.conf 66
A.3 Testergebnisse: Ladezeiten im Frontend 67
A.4 Testergebnisse: Zertifikat und Sidecar Overhead 69
A.5 Testergebnisse: Erreichbarkeit nach HTTP-Methode 70

Tabellenverzeichnis

6.1 Vergleich der Vor- und Nachteile von SSR und CSR 34

Abkürzungsverzeichnis

API Application-Programming-Interface. 15, 17, 23, 26, 28, 44, 45, 47

CA Certificate Authority. 22, 23

CMS Content-Management-System. 7, 8

CORS Cross Origin Ressource Sharing. 34, 37

CSR Client Side Rendering. 33, 34, 36, 37, 62

DVZ Datenverarbeitungszentrum Mecklenburg-Vorpommern. 7, 10, 11, 55

eAST elektronischen Antragstellung. 7

eAST Elektronischer Antragstellung. 8, 11, 26, 29, 30, 44–48, 51, 52

FMS Formular-Management-System. 8, 45

IDMV Infodienste M-V. 7, 9, 11, 26, 29, 30, 32, 46, 50

jaxrs „Java API for RESTful Web Services". 47

JS JavaScript. 28, 37, 46, 52

JSF Java Server Faces. 35, 52

JSON JavaScript Object Notation. 48

JWT JSON Web Token. 22

MS Microservice. 14, 17, 18

MVSP MV-Serviceportal. 2, 5–7, 10, 25–27, 35, 41, 43, 49, 50, 55

OD Online-Dienst. 36, 44

OZG Online Zugangsgesetz. 2, 5–7, 11, 24

REST Representational-State-Transfer. 15, 26, 44

SCS Self-Contained-System. 2, 5, 6, 10, 14, 15, 19, 45

SEO Search Engine Optimization. 34

SOA Service-Oriented-Architecture. 12–14, 17

SSR Server Side Rendering. 33, 34, 37, 62

TLS Transport Layer Security. 10, 24, 25, 40, 50

UI User Interface. 5, 14

UUID Universally Unique Identifier. 47, 48

VM Virtuelle Maschine. 20

VPS Virtuelle Poststelle. 8

ZVP Zahlungsverkehrsplattform. 8

A Weitere Listings

Listing A.1: Implementierung: Docker-Compose.yaml

```
version: '3'

services:
####
# setup for east
####
  db:
   build:
     context: ./db
     dockerfile: Dockerfile
   image: db
   environment:
     MYSQL_ROOT_PASSWORD: "XXXXXXXXXXXXX"
     MYSQL_DATABASE: "XXXXXXXXXXXXX"
     MYSQL_ALLOW_EMPTY_PASSWORD: "no"
     MYSQL_USER: "XXXXXXXXXXXXX"
     MYSQL_PASSWORD: "XXXXXXXXXXXXX"
   expose:
     - "3306"
   container_name: db
  east:
   build:
     context: ./eAST
     dockerfile: Dockerfile
   image:  east:latest
   environment:
     - DB_URI=db
     - SERVICE_REDIRECT=http://XXXXXXXXXXXXX/LoginServlet
     - IDMV_URL=https://XXXXXXXXXXXXX
     - SERVER_IP=http://sidecar_east
   expose:
     - "8080"
   container_name: east

  sidecar_east:
   build:
     context: ./sidecar
     dockerfile: Dockerfile
   image: sidecar:latest
   depends_on:
     - east
   environment:
     - HOST=east
     - PORT=8080
```

```
        - LISTEN=443
46    expose:
        - "80"
48      - "443"
      container_name: sidecar_east
50
####
52 # setup for online dienst
####
54  od:
      build:
56      context: ./OD
        dockerfile: od.Dockerfile
58    image: od:latest
      expose:
60      - "5000"
      container_name: od
62
    sidecarod:
64    build:
        context: ./sidecar
66      dockerfile: Dockerfile
      image: sidecar:latest
68    depends_on:
        - od
70      - sidecar_east
      environment:
72      - HOST=od
        - PORT=5000
74      - LISTEN=443
      expose:
76      - "80"
        - "443"
78    container_name: sidecarod

80 ####
   # reverse Proxy for entire Service
82 ####
    reverse_proxy:
84    build:
        context: ./reverseProxy
86      dockerfile: Dockerfile
      image: reverseproxy:latest
88    ports:
        - "80:80"
90    depends_on:
        - sidecar_east
92      - sidecarod
      container_name: reverse_proxy
```

```
  <VirtualHost *:80>
2 # do not act as a forward proxy
  ProxyRequests off
4 ProxyPreserveHost On

6 # use these client certifictes
```

```
    SSLProxyEngine On
    SSLProxyCACertificateFile /certs/ca.crt
    SSLProxyMachineCertificateFile /certs/client.pem

    # do not check if the certificate is correct for the server name
    # this is needed since we are using self-signed certs
    SSLProxyVerify none
    SSLProxyCheckPeerCN off
    SSLProxyCheckPeerName off
    SSLProxyCheckPeerExpire off

    <Location />
        ProxyPass https://sidecar_east/
        ProxyPassReverse https://sidecar_east/
        Require all granted
    </Location>

    <Location /od1/>
        ProxyPass https://sidecarod/
        ProxyPassReverse https://sidecarod/
        Require all granted
    </Location>

</VirtualHost>
<VirtualHost *:LISTEN>
    # do not act as a forward proxy
    ProxyRequests off
    ProxyPreserveHost On
    RequestHeader set X-Forwarded-Proto "https"

    # use these server certs for SSL
    SSLEngine on
    SSLCertificateFile "/certs/server.crt"
    SSLCertificateKeyFile "/certs/server.key"

    # check client certs against this CA
    SSLVerifyDepth       1
    SSLVerifyClient      require
    SSLCACertificateFile "/certs/ca.crt"

    # apache does not support envar
    # the replacement is done via sed in the start up script
    <Location />
        ProxyPass http://HOST:PORT/ retry=1 acquire=3000 timeout=600 Keepalive=On
        ProxyPassReverse http://HOST:PORT/
        Require all granted
    </Location>
</VirtualHost>
```

Listing A.2: Sidecar apache httpd.conf

Listing A.3: Testergebnisse: Ladezeiten im Frontend

```
[
    {
        "Type": "OD",
        "measured_with": "Page load time",
```

```
        "avg": 1.028,
        "avg_response": 0.384,
        "times": [
            {
                "total": 1.02,
                "request": 0.363
            },
            {
                "total": 0.97,
                "request": 0.417
            },
            {
                "total": 1.08,
                "request": 0.367
            },
            {
                "total": 0.95,
                "request": 0.362
            },
            {
                "total": 1.12,
                "request": 0.411
            }
        ]
    },
    {
        "Type": "OZG",
        "measured_with": "Page load time",
        "avg": 1.06,
        "avg_response": 0.409,
        "times": [
            {
                "total": 1.01,
                "request": 0.418
            },
            {
                "total": 1.06,
                "request": 0.382
            },
            {
                "total": 1.03,
                "request": 0.432
            },
            {
                "total": 1.03,
                "request": 0.404
            },
            {
                "total": 1.17,
                "request": 0.408
            }
        ]
    },
    {
        "Type": "loadOD function",
        "measured_with": "console.Time()",
```

```
        "avg": 35,
        "times": [
            75.7470703125,87.537841796875,43.8759765625,32.325927734375,
            141.171142578125,28.537841796875,30.473876953125,29.9130859375,
            30.756103515625,31.27197265625,27.36181640625,28.356689453125,
            26.906005859375,27.9208984375,25.024169921875,25.64501953125,
            26.47900390625,25.375244140625,24.899169921875,28.380859375,
            26.639892578125,24.0439453125,34.8759765625,28.109130859375,
            28.541015625,28.0439453125,23.632080078125,30.636962890625,
            47.662109375,32.162109375,47.678955078125,23.679931640625,
            25.756103515625,24.102783203125,30.828125,28.330078125,
            30.4951171875,29.723388671875,26.85009765625,28.1982421875,
            29.345947265625,28.827880859375,28.711181640625,25.036865234375,
            33.264892578125,68.718017578125,37.2099609375,38.99609375,
            36.888916015625,29.402099609375
        ]
    }
]
```


Listing A.4: Testergebnisse: Zertifikat und Sidecar Overhead


```
[
    {
        "avg": 0.003753623962402344,
        "route": "https://sidecar0/",
        "times": [
            0.011909008026123047,0.004461526870727539,0.0037856101989746094,
            0.0034439563751220703,0.0034728050231933594,0.004515647888183594,
            0.0036492347717285156,0.003619670867919922,0.003494739532470703,
            0.003459453582763672,0.0034737586975097656,0.003751754760742 1875,
            0.0033960342407226562,0.003484964370727539,0.0034127235412597656,
            0.003541707992553711,0.0032999515533447266,0.0034165382385253906,
            0.003474712371826172,0.003431558609008789,0.0035927295684814453,
            0.004484415054321289,0.003457784652709961,0.0033376216888427734,
            0.0034945011138916016,0.003400087356567383,0.00384521484375,
            0.0036101341247558594,0.0036399364471435547,0.003497600555419922,
            0.003361225128173828,0.003957509994506836,0.003496885299682617,
            0.003355741500854492,0.003507375717163086,0.0035808086395263672,
            0.00360488891601562 5,0.003548383712768554 7,0.00363922119140625,
            0.0035400390625,0.0035848617553710938,0.0035436153411865234,
            0.003600597381591797,0.0034456253051757812,0.0034890174865722656,
            0.003494739532470703,0.003542661666870117,0.0035588741302490234,
            0.0035207271575927734,0.003453016281127929 7
        ]
    },
    {
        "avg": 0.0014065170288085937,
        "route": "http://server0/",
        "times": [
            0.0015368461608886719,0.0014355182647705078,0.0014355182647705078,
            0.0014760494232177734,0.001363992691040039,0.0014002323150634766,
            0.0014119148254394531,0.0013628005981445312,0.0014057159423828125,
            0.0015788078308105469,0.0013267993927001953,0.0013799667358398438,
            0.0014052391052246094,0.0013766288757324219,0.0012874603271484375,
            0.0012962818145751953,0.0015621185302734375,0.0012922286987304688,
            0.0013058185577392578,0.001371145248413086,0.0013470649719238281,
```

```
36                0.00131988525390625,0.0013568401336669922,0.0015556812286376953,
                  0.0013849735260009766,0.0013241767883300781,0.0013017654418945312,
38                0.0012576580047607422,0.001352548599243164,0.0013115406036376953,
                  0.0015110969543457031,0.0013267993927001953,0.0013232231140136719,
40                0.001556396484375,0.0012917518615722656,0.0013239383697509766,
                  0.0012969970703125,0.002304553985595703,0.0012667179107666016,
42                0.0013208389282226562,0.0013308525085449219,0.0012922286987304688,
                  0.001308441162109375,0.0013573169708251953,0.0014801025390625,
44                0.0013051033020019531,0.0021622180938720703,0.0013127326965332031,
                  0.0013570785522460938,0.0013742446899414062
46            ]
        }
48  ]
```

Listing A.5: Testergebnisse: Erreichbarkeit nach HTTP-Methode

```
    [
2       {
            "route": "http://XX.XX.XX.XX/",
4           "results": {
                "get": 200,
6               "post": 200,
                "put": 405,
8               "patch": 405,
                "delete": 405,
10              "head": 200,
                "options": 405
12          }
       },
14      {
            "route": "http://XX.XX.XX.XX/od1",
16          "results": {
                "get": 200,
18              "post": 405,
                "put": 405,
20              "patch": 405,
                "delete": 405,
22              "head": 200,
                "options": 405
24          }
        },
26      {
            "route": "http://XX.XX.XX.XX/api/v1/vorhaben/1",
28          "results": {
                "get": 403,
30              "post": 405,
                "put": 405,
32              "patch": 405,
                "delete": 405,
34              "head": 403,
                "options": 405
36          }
        },
38      {
            "route": "http://XX.XX.XX.XX/od1/api/v1/vorhaben/1",
40          "results": {
```

```
            "get": 404,
            "post": 405,
            "put": 401,
            "patch": 405,
            "delete": 405,
            "head": 404,
            "options": 405
        }
    }
]
```

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die hier vorliegende Arbeit selbstständig verfasst und nur unter Verwendung der aufgeführten Hilfsmittel angefertigt habe. Die Stellen der Arbeit, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet.

Ich erkläre ferner, dass ich die vorliegende Arbeit in keinem anderen Prüfungsverfahren als Prüfungsarbeit eingereicht habe oder einreichen werde.

Die eingereichte schriftliche Fassung entspricht der auf dem Medium gespeicherten Fassung.

Ort, Datum　　　　　　　　　　Unterschrift