HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

# Micro-Frontends

## Studienarbeit

**Abteilung Informatik**
**Fachhochschule Rapperswil**

**Autor(en):**
Matthias Baumann

**Datum:**
2019-01-14

Das Microfrontends Konzept übernimmt die Prinzipien der Microservices wie Isolation von Code, Trennung der Aufgabenbereiche und Skalierbarkeit in das Frontend.

Dieser Bericht stellt Architektur-Alternativen zum Bau von Microfrontends vor. Diese Alternativen werden im Anschluss anhand eines Kriterienkatalogs und einer Beispielapplikation verglichen.

Im Fazit wird die Machbarkeit, Chancen und Risiken des Konzeptes diskutiert. Der Bericht schliesst mit einem Ausblick auf nächste Forschungsschwerpunkte ab.

# Inhaltsverzeichnis

**1 Einführung** **3**
1.1 Einführung in die Microfrontend Architektur . . . . . . . 3
1.2 Anforderungen um die Architekturen zu vergleichen . . . 6
1.3 Kriterienkatalog . . . . . . . . . . . . . . . . . . . . . . . 6
1.3.1 Migrationsarbeit . . . . . . . . . . . . . . . . . . . 7
1.3.2 Aktualisierbar in Produktion . . . . . . . . . . . 8
1.3.3 Unabhängigkeit der Services . . . . . . . . . . . . 10
1.3.4 Effekt auf die Geschwindigkeit . . . . . . . . . . . 13
1.3.5 Theoretische Skalierbarkeit . . . . . . . . . . . . 13
1.3.6 Informationssicherheit . . . . . . . . . . . . . . . 15

**2 Implementation der Komposition von Microfrontends** **17**
2.1 Einführung in die Beispielapplikation . . . . . . . . . . . 17
2.1.1 Software Engineering Dokumente . . . . . . . . . 17
2.2 Wichtige Architekturentscheidungen . . . . . . . . . . . 18
2.2.1 Eigene Beispielanwendung . . . . . . . . . . . . . 18
2.2.2 Architektur der Applikation . . . . . . . . . . . . 18
2.2.3 Entscheidung über Server Framework . . . . . . . 19
2.3 Portal Komposition . . . . . . . . . . . . . . . . . . . . . 20
2.3.1 Einführung in die Architektur . . . . . . . . . . . 20
2.3.2 Implementation der Architektur . . . . . . . . . . 21
2.3.3 Resultate aus der Programmierung der Alternative 22
2.3.4 Diskussion der Alternative . . . . . . . . . . . . . 23
2.4 Backend Komposition . . . . . . . . . . . . . . . . . . . . 24
2.4.1 Einführung in die Architektur . . . . . . . . . . . 24
2.4.2 Implementation der Architektur . . . . . . . . . . 25
2.4.3 Resultate aus der Programmierung der Alternative 27
2.4.4 Diskussion der Alternative . . . . . . . . . . . . . 27
2.5 Frontend Komposition . . . . . . . . . . . . . . . . . . . . 28
2.5.1 Einführung in die Architektur . . . . . . . . . . . 28
2.5.2 Implementation der Architektur . . . . . . . . . . 28
2.5.3 Resultate aus der Programmierung der Alternative 31
2.5.4 Diskussion der Alternative . . . . . . . . . . . . . 31

**3 Resultate der Arbeit** 33
3.1 Eignung der Microservice Patterns von Richardson . . . 33
3.2 Beispielapplikation . . . 34
3.2.1 Installation der Beispielapplikation . . . 34
3.3 Kriterien-basierter Vergleich von Architektur Optionen . 36
3.3.1 Einführung . . . 36
3.3.2 Migrationsarbeit . . . 36
3.3.3 Aktualisierbar in Produktion . . . 38
3.3.4 Unabhängigkeit der Services . . . 39
3.3.5 Effekt auf die Geschwindigkeit . . . 40
3.3.6 Theoretische Skalierbarkeit . . . 42
3.3.7 Informationssicherheit . . . 43
3.4 Diskussion der Ergebnisse des Vergleichs der Alternativen 44
3.4.1 Ausarbeitungszeit der Prototypen . . . 45
3.4.2 Unterbrechungsfreie Aktualisierbarkeit: Kann das Microfrontend zusammen mit dem Microservice ausgetauscht werden? . . . 45
3.4.3 Die Microfrontend Architektur darf nicht die Unabhängigkeit der Microservices erweichen . . . 46
3.4.4 Erkenntnisse über mögliche Technologien . . . 47
3.5 Ausblick auf weiterführende Arbeiten . . . 47

# Management summary

## Ausgangslage

Microservices stellen eine moderne Art der service oriented architecture dar. Die Vorteile der Microservice-Architektur sind unter anderem die Trennung von Aufgabenbereiche in kleinere, verteilte, atomare Applikationen, die Definition der Datenhoheit und die Skalierbarkeit.

Diese Vorteile werden im Konzept der Microfrontends in die Präsentationsschicht übernommen. Um dies zu erreichen muss der *User Interface* (UI)-Code auf die Services aufgeteilt werden und vor der Anzeige zusammengefügt werden.

## Vorgehen

Diese Arbeit untersucht das Zusammenfügen mithilfe von drei Architekturalternativen. Diese werden mithilfe eines Kriterienkatalogs verglichen und bewertet. Es ist nicht das Ziel der Arbeit die beste Alternative zu finden, sondern die Chancen und Risiken der Auswahlmöglichkeiten zu beleuchten und bei der Entscheidungsfindung zu unterstützen.

Der Autor hat zur Überprüfung der Kriterien eine Prototypimplementierung der Möglichkeiten ausgearbeitet. In der Ergebnissektion wird auf die Erkenntnissen aus der Programmierung der Prototypen verwiesen.

Für die Implementierung der Beispielapplikation wurde Spring[1], Thymeleaf [2] und React[3] im Frontend eingesetzt.

## Ergebnisse

Die Ergebnisse zeigen die Erkenntnisse aus der Implementierung der Alternativen auf. Auch wurden literarische Quellen herbeigezogen bei Kriterien

Die Arbeit hat die Alternativen nicht gewertet. Es ist in jedem Einzelfall die Gewichtung der Kriterien zu bestimmen um die optimale Implementationsmöglichkeit zu finden.

[1] https://spring.io/
[2] https://www.thymeleaf.org/
[3] https://reactjs.org/

## Ausblick

Der Verfasser sieht in folgenden Themen noch Möglichkeiten zur Weiterführung der Arbeit.

Eine Untersuchung des *Backends for Frontends* Patterns von Newman. Es kann dann auch einen Vergleich zu den Alternativen dieser Arbeit gezogen werden.

Eine Weiterführung der Beispielapplikation von dem Prototyp, der in dieser Arbeit erstellt wurde, in eine umfangreichere Version, die die Alternativen genauer einführt.

# 1 Einführung

## 1.1 Einführung in die Microfrontend Architektur

Diese Arbeit vergleicht Architektur-Alternativen für die Verteilung des Frontends auf die Services. Der Text wird die Architektur beschreiben und die Chancen und Risiken anhand eines Kriterienkataloges untersuchen. Diese Kriterien werden verwendet um die ausprogrammierten Alternativen zu vergleichen. Das letzte Kapitel wird die gewonnen Fakten über die Architekturalternativen vergleichen und bewerten. Zum Schluss wir die Arbeit einen Ausblick auf weiterführende Arbeiten geben.

**Microservices**

Der Begriff „Microservices“ ist in einem Software-Architektur-Workshop in Mai 2011 aufgekommen [Jam18]. Microservices werden als „lose gekoppelte *Service Oriented Archtitecture* (SOA) mit einem begrenzten Kontext“[Coc16] beschrieben. Nicolai Josuttis beschreibt Microservices als der best-practice Ansatz um SOA zu realisieren. [Pau+17] Vielfach wird diese neuartige Architektur bei Webservices mit aktuellen Technologien wie *Continuous Integration* (CI) und *Representational State Transfer* (REST) [San01]ful HTTP verbunden. [Pau+17]

Die Applikationsschicht wurde in verschiedene Services aufgeteilt, aber die Präsentationsschicht ist zurzeit eine grosse Website, die alle Services überspannt. Das Frontend übernimmt die Koordinierung, damit die Services in der richtigen Reihenfolge aufgerufen werden. Beispielsweise ist im Bild 1.1 der Login-Service ersichtlich, der vor den anderen Services aufgerufen werden muss um den Nutzer zu authentisieren.

Dies hat den Nachteil, dass in der Präsentationsschicht Code für verschiedene Services miteinander verhängt ist. Dadurch verlieren wir den grossen Vorteil von Microservices, ihre Unabhängigkeit voneinander. Es existieren keine logischen Barrikaden zwischen den Frontends. Aus diesem Grund kann der gesammte Code auf jede Service API zugreifen. Es existieren keine klar definierten Zuständigkeiten. Ausserdem muss jeder Entwickler im Frontend die gleichen Abhängigkeiten benutzen. Als

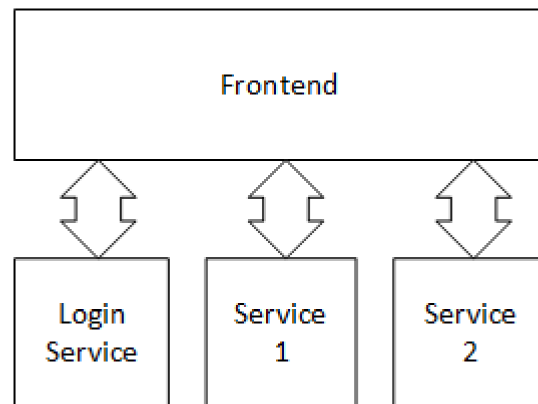


Abbildung 1.1: Eine traditionelle Service Architektur

Beispiel muss der gesammte Frontend-Code auf Angular[1] aufgesetzt werden.

**Microfrontends** Das *Microfrontend* Konzept nimmt die Vorteile, die in der *Applikationsschicht* mit dem *Microservices* Konzept gewonnen wurden, in die Präsentation-Schicht. Wir unterteilen das UI in selbst-verwaltende Subsysteme, welche per Definition nur zu ihrem Microservice kommunizieren. Die Kombination von Microservice und Microfrontend sind selbstverwaltend und ein geschlossenes System.

Unter 1.2 ist der Aufbau von Microfrontends ersichtlich. Der Service liefert sein eigenes UI, das als Teil der übergeordneten Seite einfach eingefügt wird. Somit ist jeder Service im Frontend unabhängig von der verwendeten Technologie bei den anderen Services. Die Microfrontends haben auch die Aufgabe mit dem Service zu kommunizieren und teilen die Daten über ein Message-System anderen Microfrontends im Browser mit. Die Funktionsweise eines solchen Messaging-Systems ist ausserhalb des Umfangs dieser Arbeit und wurde in anderen Arbeiten[2] untersucht.

**Vorteile** Microfrontends haben eine breite Palette von Vorteilen. Von einer technischen Perspektive, der Code ist modularisierter, Komponenten sind austauschbar in Produktion und sie unterteilen die Applikation in kleinere Sicherheitsbereiche. Die Microfrontends könnnen die Präsenz von der Skeleton Page erwarten. Beispielsweise können die Microfrontends Nachrichten über einen gemeinsamen Message-Bus[Cur05] austauschen.[Söd17]

[1] https://angular.io
[2] https://medium.com/@tomsoderlund/micro-frontends-a-microservice-approach-to-front-end-web-development-f325ebdadc16#3b29

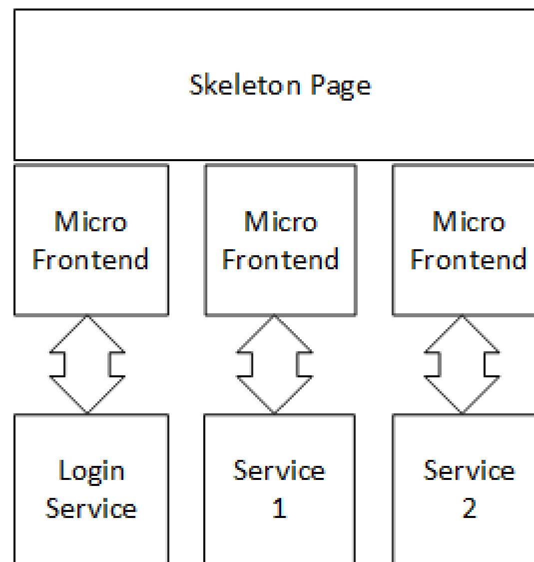


Abbildung 1.2: Eine Mikrofrontend Architektur

Das Pattern könnte den Entwicklungsprozess von Webservices grundlegend verändern. Jedes Microfrontend kann individuell programmiert werden, auf die Produktionsumgebung geladen werden und instand gehalten werden. Die Aufsicht kann durch ein kleines und unabhängiges Team gewahrt werden. Die Teams müssen sich nicht untereinander absprechen beim Aktualisieren der Software, denn der Frontend-Code ist unabhängig voneinander.

**Risiken**

Auf der anderen Seite, ist die Implementierung der Microservice- und Microfrontend-Patterns aufwendig und verzögert die Entwicklung von neuen Funktionen. Ausserdem kann das Einsetzen von verschiedenen UI-Hilfsbibliotheken viele Ressourcen im Browser brauchen, zum Beispiel haben viele aufwendige Webservices einen hohen Hauptspeicherverbrauch auf der Nutzermaschine.

Die Trennung der Code-Abhängigkeiten kann zur Verletzung des *dont repeat yourself* Prinzipes[HT08] führen. Die Javascript-Bibliotheken, auf denen die Microfrontends basieren, muss mehrmals zum Code compiliert werden.

## 1.2 Anforderungen um die Architekturen zu vergleichen

In dieser Sektion beschreibe ich die Vorgaben um die Beispielapplikationen zu vergleichen. Im Groben muss die Beispielapplikation einfach genug sein um mit wenigen Konsolenbefehlen auszuführen. Auf der anderen Seite sollte es komplex genug sein um einen Vergleich der Architekturen zu tätigen.

Um die Beispielapplikationen auszuwählen wurden in einer Vorstudie verschiedene Patterns untersucht. Als Orientierungspunkt wurden mir die Patterns von Chris Richardson [Ric18g] nahegelgt.

## 1.3 Kriterienkatalog

Der Kriterienkatalog zeigt die Aufstellung der Kriterien zur Bewertung der Architekturalternativen. Diese Kriterien erlauben einen qualitativen Vergleich der Architekturen und sollen einem SW-Architekten ermöglichen die Vor- und Nachteile der Architekturen zu vergleichen.

Der Kriterienkatalog basiert auf dem Katalog von ISO25010 [com18]. Ich habe die relevantesten Charakteristiken ausgesucht um die Architekturen zu vergleichen. Um die Anzahl der Kriterien weiter zu reduzieren, habe ich verschiedene Kategorien zusammengefasst.

Um die Fälle vergleichbar zu gestalten wird das Konzept der „Landing Zones“[Wir11] eingeführt.

**Beispielskriterium**

- Der erste Paragraph zeigt die Motivation für das Kriterium. Er enthält Kontextinformationen.

  *Kategorie*: Das Kriterium beschrieben als Frage, die eindeutig bestimmt werden kann.

  + Schranke ab welcher die Implementation das Kriterium erfüllt

  − Schranke ab welcher die Implementation das Kriterium nicht erfüllt

### 1.3.1 Migrationsarbeit

Das Kriterium „Migrationsarbeit“ untersucht Risiken zu der Überführung in eine Microfrontend-Umgebung. Jedes Entwicklungsteam muss Refactorings vornehmen um die Frontends voneinander zu trennen. Dazu kommt, dass alle Teammitglieder die neuen Programmierungstechniken der Microfrontend-Architektur verstehen müssen.

Das Team muss die folgenden Kriterien beachten bei dem Übergang zur neuen Architektur:

- Die *Verständnisarbeit* von jedem Entwickler um das Verständnis aufzubauen. Nach der Entscheidung die Architektur auf eine Microfrontends-Architektur zu überführen, braucht jeder Entwickler eine Einführungslektion und Einarbeitungszeit um die neue Architektur zu verstehen.

  Gute Dokumentation und ein einfaches Konzept helfen bei dem Verständnis neuer Architektur-Alternativen.

  *Ausarbeitungszeit des Konzeptes*: Wie lange brauchte ich das Konzept auszuarbeiten?

  + 2 Stunden
  − 6 Stunden

- Die *Umstellungszeit* wird auch dadurch bestimmt, wie lange die Ausarbeitung des Prototyps der neuen Architektur dauert.

  *Ausarbeitungszeit des Prototyps*: Wie lange dauert die Ausarbeitung des ersten Prototyps der Architektur-Alternative?

  + 2 Stunden
  − 6 Stunden

  Wie Zeitaufwendig ist die Umstellung bis zum ersten hybriden Prototyp?

- In der *Anlaufszeit* des Entwicklungsteams ist es wahrscheinlich, dass die Entwickler mehr fehlerhaften Code schreiben werden.

  *Fehler im Code*: Wie viele Fehler hatte ich im Code?

  + Weniger als 2 Fehler
  − Mehr als 2 Fehler

- In *grösseren Projekten* muss der Übergang schrittweise geschehen. Damit werden grosse Umbrüche in der Produktion verhindert und die daraus resultierenden Probleme.

  Mit der Anzeige von Teilen der alten Applikation in der neuen Anwendung kann die alte Website Funktion für Funktion abgelöst werden und durch das neue Frontend ersetzt werden.

  *Schrittweiser Übergang*: Kann eine externe Seite im Micro-Frontend angezeigt werden?

  + Ja
  − Nein

- Aus der Forderung für einen schrittweisen Übergang folgt, dass das monolithische Frontend und Microfrontends nebeneinander bestehen müssen. Dies erlaubt einen guten, schrittweisen Migrationspfad.

  *Schrittweiser Übergang*: Kann die alte Seite übernommen werden und im Aussehen auf die neue Seite angepasst werden?

  + Forderungen
    * Integration von monolithischer Website mit funktionierenden Links und Bildern
    * Nur die notwendigen Teile eines HTML Dokumentes werden übernommen
    * Der Style der neuen Seite kann übernommen werden auf das alte Frontend, damit die Nutzererlebnis nicht gestört wird.
  − Nicht alle Forderungen sind erfüllt.

### 1.3.2 Aktualisierbar in Produktion

Entwicklungsteams sind für einen Service zuständig. In einem Team arbeiten Datenbankadministratoren, Backend- und Frontendentwickler eng zusammen um einen Service weiter zu entwickeln. Diese können Lösungen schneller und in kleineren Intervallen bereitstellen. Die Aktualisierung des Services muss auch in der Produktion innerhalb von Minuten bis Stunden ablaufen. [Auc18]

| Beschreibung | − | + |
| --- | --- | --- |
| Wie lange brauchte ich das Konzept auszuarbeiten? | 6 Stunden | 2 Stunden |
| Wie lange dauert die Ausarbeitung des ersten Prototyps der Architektur-Alternative? | 6 Stunden | 2 Stunden |
| Wie viele Fehler hatte ich im Code? | 2 Fehler | 1 Fehler und weniger |
| Kann eine externe Seite im Micro-Frontend angezeigt werden? | Nein | Ja |
| Kann die alte Seite übernommen werden und im Aussehen auf die neue Seite angepasst werden? | Nicht alle Forderungen sind erfüllt. | Forderungen sind erfüllt. |

Tabelle 1.1: Zusammenfassung Kriterien für Migrationsarbeit

Wie im IEEE Artikel „Microservices: The journey so far and challenges ahead“ beschrieben wird, gibt es viele Werkzeuge zur Automatisierung der Aktualisierung von Microservices auf dem Markt. [Jam18] Beispielsweise Spinnaker[3] Kubernetes[4] und Docker [5]. Diese Arbeit überprüft stattdessen die Kompatibilität von den Alternativen zu einer solchen Anforderung.

Das Pattern muss die folgenden Kriterien beachten:

- Das Microfrontend muss zusammen mit dem Microservice austauschbar sein. Damit kann garantiert werden, dass bei einer Änderung der Business-Logik, das Frontend bei der nächsten Anfrage aktualisiert wird und es zu keinem Unterbruch des Services kommt weil das Frontend den neuen API-Vertrag nicht kennt.

  *Unterbrechungsfreie Aktualisierbarkeit*: Kann das Microfrontend zusammen mit dem Microservice ausgetauscht werden?

  + Ja

  – Nein

- Die Aktualisierung sollte so wenige Services wie möglich betreffen. Jeder zusätzlich ausgetauschte Service macht die Aktualisierung komplexer.

  *Unterbrechungsfreie Aktualisierbarkeit*: Die Aktualisierung des Frontends darf nur *einen* Service betreffen.

  + Ein Service ist betroffen

  – Zwei Services sind betroffen

### 1.3.3 Unabhängigkeit der Services

Die tägliche Entwicklung von grossen, monolithischen Systemen ist langsam. Aus der lange dauernden Bau- und Startzeit wird der work-build-run-test-rhythm sehr langsam. Folglich sinkt die Produktivität der Entwickler. [Ric18e]

Durch die Unabhängigkeit der Services kann der Aktualisierungsprozess einfach gehalten werden. Von einer Datenbank-Aktualisierung darf nur

[3] https://www.spinnaker.io
[4] https://kubernetes.io
[5] https://www.docker.com

| Beschreibung | − | + |
|---|---|---|
| Kann das Microfrontend zusammen mit dem Microservice ausgetauscht werden? | Nein | Ja |
| Die Aktualisierung des Frontends darf nur *einen* Service betreffen. | Zwei Services | Ein Service |

Tabelle 1.2: Kriterien für Aktualisierbar in Produktion

der Service betroffen sein, dem die Datenbank gehört. Bei einer Trennung der Services innerhalb einer monolithischen Anwendung, muss der gesamte Prozess ausgetauscht werden.

Wenn Services in einer Microservice Architektur miteinander kommunizieren möchten, dann müssen sie das mit Messages über die öffentliche *Application Programming Interface* (API) tun. [Ric18e]

Die Unabhängigkeit der Microservices im Applikationschicht darf nicht im Frontend erweicht werden.

Die Unabhängigkeit der Services muss die folgenden Kriterien beachten:

- Die Komponenten sollten unabhängig voneinander sein, damit sie ausgetauscht werden können in der Produktion. Die einzige Abhängigkeit der Webserver darf auf ihr Applikationserver sein.

  *Unabhängigkeit*: Ausser der Abhängigkeit auf den Applikationserver dürfen die Webserver keine weitere Verbindungen benötigen.

  + Einzige Abhängigkeit ist auf Applikationserver des Microfrontends.
  − Das Frontend besitzt mehr als eine Abhängigkeit.

- Um den Akualisierungsprozess zu ermöglichen muss der Service unabhängig von anderen Services in der Build-Chain verarbeitbar sein.

  *Unabhängigkeit*: Der Service darf keine Abhängigkeiten auf andere Services besitzen in der build-phase.

  + Service ist unabhängig von anderen Services kompilierbar

| Beschreibung | − | + |
|---|---|---|
| Ausser der Abhängigkeit auf den Applicationserver dürfen die Webserver keine weitere Verbindungen benötigen. | Einzige Abhängigkeit ist auf Application-Server des Micro-Frontends. | Einzige Abhängigkeit ist auf Application-Server des Micro-Frontends. |
| Der Service darf keine Abhängigkeiten auf andere Services besitzen in der „Build-Phase“. | Service hat zur „Build-Zeit“ Abhängigkeiten auf andere Services in der Infrastruktur. | Service ist unabhängig von anderen Services kompilierbar |
| Die Microfrontend Architektur darf nicht die Unabhängigkeit der Microservices erweichen. | Durch Kopplungen im Frontend sind die Microfrontends nicht individuell austauschbar. | Die Frontends sind unabhängig. |
| Services müssen einzeln aktualisierbar sein. | Nein | Ja |

Tabelle 1.3: Kriterien für Unabhängigkeit der Services

- − Service hat zur „Build-Zeit“ Abhängigkeiten auf andere Services in der Infrastruktur.

- Die Microservice-Architektur empfiehlt die komplette Trennung der Services. Der Microfrontend-Vorschlag darf diese Trennung nicht erweichen, damit die Vorteile in der Applikationsschicht auch in die Präsentationsschicht übernommen werden können.

  *Unabhängigkeit*: Die Microfrontend Architektur darf nicht die Unabhängigkeit der Microservices aufweichen.

  + \+ Die Frontends sind unabhängig.
  + − Durch Kopplungen im Frontend sind die Microfrontends nicht individuell austauschbar.

- *Unabhängigkeit*: Services müssen einzeln aktualisierbar sein.

  + \+ Ja
  + − Nein

### 1.3.4 Effekt auf die Geschwindigkeit

Der Nutzer misst die Geschwindigkeit einer Seite vor allem an der Ladegeschwindigkeit. Für Webunternehmen ist es sehr wichtig, dass die Nutzer kontinurierlich über die Webseite bewegen können und nicht durch Ladebälken gestört werden. Die Ladegechwindigkeit hat verschiedene Messgrössen um diese feiner zu unterteilen:

**Time to first Byte** Zeit die vergeht bis eine Antwort des Servers eintrifft

**Content Download** Zeit in welcher Netzwerk-Pakete für das File ankommen

**Rendering** Zeit um die Daten zu interpretieren und die Ansicht zu zeichnen

Die „Time to first Byte“ ist sehr wichtig und normalerweise langsam für Server-Side-Rendering-Lösungen wie Thymeleaf. Bei Frontend-Rendering, wie zum Beispiel Angular, React und anderen Javascript Bibliotheken, ist die „Time to first Byte“ sehr schnell. Diese Lösungen brauchen viel Zeit beim Rendering. Heutzutage wird die zweite Option bevorzugt, denn immerhin bleibt da der Bildschirm nicht leer in der Ladezeit, sondern sieht schnell nach der Webseite aus, auch wenn noch nicht alle Elemente eingetroffen sind.

Aus diesem Grund, müssen folgende Punkte bei der Geschwindigkeit beachtet werden:

- *Geschwindigkeit*: Die definierten Zeiten müssen so kurz wie möglich sein
- *Geschwindigkeit*: Der Nutzer muss das Gefühl bekommen, dass die Website schnell lädt
- *Geschwindigkeit*: Für Backend Lösungen muss die Zeit bis zum Ausliefern der Seite besonders beachtet werden
- *Geschwindigkeit*: Für Frontend Rendering muss der Ressourcenverbrauch durch den Seitenrenderer beachtet werden

### 1.3.5 Theoretische Skalierbarkeit

Der Autor wird die Skalierbarkeit ableiten von den Abläufen und den Daten, die zwischen den Kommunikationspartnern gesendet werden.

Die Skalierbarkeit einer Microservice-Infrastruktur mit Microfrontends muss folgende Punkte beachten:

| Beschreibung | − | + |
|---|---|---|
| Die Ladegeschwindigkeit muss so schnell wie möglich sein. | Die Ladegeschwindigkeit ist bedeutsam langsamer als die Alternativen. | Die Geschwindigkeit ist bedeutsam schneller als die Alternativen. |
| Der Nutzer hat das Gefühl einer flüssigen Ladegeschwindigkeit | Das Laden stockt oder der Nutzer muss lange einen Browserladebalken ansehen. | Die Website ist schnell beim Nutzer und braucht kurz zum Seitenaufbau. |
| Die Zeit zur Auslieferung der Seite (TTFB) muss so schnell wie möglich sein. | Die Alternativen sind bedeutend schneller in TTFB. | Die Alternativen sind bedeutend langsamer inTTFB. |
| Die Zeit zwischen der Auslieferung und dem Fertigstellen des Renderings (Rendering Time) muss so schnell wie möglich sein. | Die Alternativen sind bedeutend schneller in Rendering Time. | Die Alternativen sind bedeutend langsamer in Rendering Time. |

Tabelle 1.4: Kriterien für Effekt auf die Geschwindigkeit

| Beschreibung | − | + |
|---|---|---|
| Die Kommunikation über das HTTP-Protokoll mit dem Applikationsserver muss stateless[Hoh15] sein. | Die Kommunikation mit dem Frontend ist ein stateful Protokoll | Die Applikationsserver haben eine stateless Kommunikation mit dem Frontend |
| Neue Instanzen können im Webserver Tier hochgefahren werden um die Anfragelast zu bewältigen. | Nein | Ja |
| Der Webserver darf keinen State, inklusive keine Session, des Nutzers sepeichern. | Er speichert Sessiondaten | Er ist stateless |

Tabelle 1.5: Kriterien für theoretische Skalierbarkeit

- Die Frontend-Server müssen stateless sein um skalierbar auf die Anfragelast zu sein.
- Das System muss dafür ausgelegt sein, dass neue Instanzen zu jeder Zeit hochgefahren werden können

### 1.3.6 Informationssicherheit

Bei der Sicherheit von einer Web Applikation müssen folgende Punkte beachtet werden:

- Die *Confidentiality, Integrity and Availability* (CIA) von Benutzerdaten muss zu jeder Zeit gewährleistet sein
- Server darf keine Nutzerdaten rendern, wenn der Nutzer nicht authentisiert ist
- Jeder Microservice muss *Authentication, Authorization, Accounting* (AAA) betreiben um Informationsdiebstahl zu verhindern
- Server-Sessions müssen über HTTPS gesichert werden

| Beschreibung | − | + |
|---|---|---|
| Die Vertraulichkeit (confidentiality) der Daten muss im Transport sichergestellt werden | Die Daten sind unverschlüsselt im Transport | Die Daten und die Frontends sind durch Verschlüsselung im Transport abgesichert. |
| Die Integrität der Daten muss sichergestellt werden. | Die Daten können von Unbekannten verändert werden | Die Backends können mit dem Microfrontend AAA betreiben. |
| Die Verfügbarkeit (availability) muss zu jeder Zeit sichergestellt sein. | Software ist nicht sicher oder lässt sich einfach zum Absturz bringen. | Software ist sicher gebaut, sodass sie nicht versehentlich abstürzt. |
| Die Services dürfen keine Daten und keine Frontend-Fragmente ausliefern an unberechtigte, dritte Personen. | Der Service liefert Daten oder Fragmente mit Daten aus | Der Service kann rechtmässige und unrechtmässige Anfragen unterscheiden. |

Tabelle 1.6: Kriterien für Informationssicherheit

# 2 Implementation der Komposition von Microfrontends

## 2.1 Einführung in die Beispielapplikation

Die Beispielapplikation baut auf den Problemstellungen beim heimischen Kochen auf. Folglich heisst die Anwendung „Cook Book“.

**Cook Book** Eine Kochbuch Applikation

**Purchase List** Eine Einkaufslisten Verwaltung

**Kitchen Device** Eine Küchengeräte Verwaltung

**Cook Book Service**

Der *Cook Book* Service integriert die Funktionen der *Purchase List* und *Kitchen Device* Services. Damit soll eine reichere Kundenerfahrung bereitgestellt werden und eine Microservice Architektur aufgebaut werden. Ich habe die Services mithilfe des *Decompose by business capability*[Ric18d] Pattern aufgeteilt.

**Hintergrundservices**

Die Hintergrundservices haben eine unabhängige Benutzeroberfläche, die dem Benutzer *Create, Read, Update and Delete* (CRUD) [Fan12] ihres Domain-Bereiches erlaubt. Ihre Daten werden vom *Cook Book* Service in eine gemeinsame Oberfläche integriert. Ab Sektion 2.3 werden die verschiedenen Integrationsformen verglichen.

### 2.1.1 Software Engineering Dokumente

Die Software-Engineering Dokumente können im Source-Code Repository unter dem Ordner docs/ betrachtet werden.

Als Teil der Arbeit wurde ein URL Schema erstellt, das im Repository-Ordner docs/url/ dokumentiert ist. [1]

Das Repository ist unter https://github.com/mattbaumann/Micro-Frontends zugänglich.

## 2.2 Wichtige Architekturentscheidungen

Ich habe die architekturelle Signifikanz der Entscheidungen, die ich während der SA getroffen habe verglichen. Die folgenden drei Entscheidungen haben die höchste Signifikanz gezeigt in dem Vergleich. Weil diese Arbeit in einer SA geschrieben wurde habe ich als Signifikanzkriterium das technische Risiko genommen.

Weitere Architekturentscheidungen können im MADR Format [2] im Code-Repository gefunden werden.

### 2.2.1 Eigene Beispielanwendung

**Auswahl Beispielanwendung**

Dem Buch „*Microservices Patterns*“[Ric18e] liegt eine Beispielanwendung bei[3]. Der Autor hat dieses Beispiel für ein Expertenmanual zugedacht. Basierend auf der Komplexität der Beispielanwendung von Richardson, entschied ich, dass es zu umfangreich ist für eine Klassenraum-Übung. Die Beispielanwendung in dieser Arbeit sollte stattdessen drei Services umfassen. Im Vergleich dazu hat die Richardson-Anwendung acht verschiedene Services und zwei produktive *Database Management System* (DBMS).

Ein Ziel dieser Arbeit ist eine einfache Klassenraum-Anwendung, die auf drei Services aufgetrennt wird. Jeder Service muss in eine getrennten Datei-basierte Datenbank schreiben.

### 2.2.2 Architektur der Applikation

Die Kochbuch Applikation soll aus drei unabhängigen Services bestehen:

- Cook Book Applikation

[1] Zum Zeitpunkt der Implementierung war das Projekt „*Springfox: Automated JSON API documentation for API's built with Spring*“ noch nicht Spring 5 fähig. [Lis18] Der Autor erstellte aus diesem Grund eine textuelle Dokumentation.

[2] https://github.com/adr/madr

[3] https://github.com/microservices-patterns/ftgo-application

- Purchase List Applikation
- Kitchen Device Applikation

Wobei der *Cook Book* Service die zwei Hintergrundservices (*Purchase List* und *Kitchen Device*) integriert in eine gemeinsame Applikation.

Die Wahl der Architektur ergab auch die Möglichkeit die folgenden Fragestellungen genauer zu untersuchen.

1. Wie verhaltet sich die Integration der zwei Services auf einem Integrationsserver?
2. Welche Schwierigkeiten treten bei einem Integrationsserver auf, der die Integration von externen Ressourcen und eigener URL-Endpunkte beherrschen muss?

### 2.2.3 Entscheidung über Server Framework

**ExpressJS**

Für den Application Layer sind zwei Optionen als Basis der Applikation zur Verfügung gestanden. Einerseits konnte Javascript mit ExpressJS[4] eingesetzt werden. Dieses Framework kann als Basis für die zukünftige Applikation dienen. Die Erweiterung Loopback[5] baut auf dem ExpressJS Framework auf und stellt die REST API zur Verfügung. Das Framework wird an der Hochschule im Modul WED2 unterrichtet.

**Spring Framework**

Die Alternative dazu ist das Spring Framework[6]. Dieses basiert auf der Programmiersprache Java und bietet viele Module an, die wiederkehrenden Code abstrahieren. So ist das Object-Relational Mapping, JSON-Objective Mapping schon durch das Framework gelöst. Der Websiten-Renderer des Frameworks „Thymeleaf“[7] ist sehr mächtig und der Websiten-Fragment-Zusammensetzung der Bibliothek erschien mächtig genug für die Portal Komposition (2.3). Das Framework wird im Modul AppArch unterrichtet, wo die Beispielanwendung in den Übungsbetrieb eingebaut werden kann.

**Entscheidung**

Ich habe mich für das Spring Framework entschieden. Das Spring Framework wird im Modul AppArch unterrichtet, die Beispielanwendung kann in den Übungsbetrieb eingebaut werden. Ausserdem ist die Sprache Java die Hauptsprache an der Hochschule.

[4] https://expressjs.com/
[5] http://loopback.io/
[6] https://spring.io/
[7] https://www.thymeleaf.org/

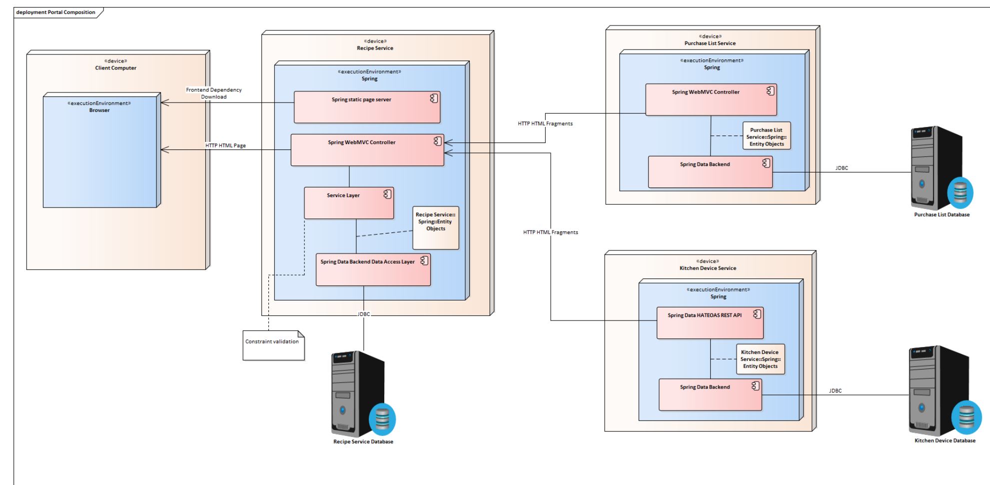


Abbildung 2.1: Portal Komposition

**Implikationen** Die Komposition der Services wurde durch das Framework sehr gut unterstützt. Im Speziellen haben folgende Spring Technologien die Beispielapplikation vereinfacht.

**WebClient** Eine Bibliothek für HTTP RESTful Service-Calls auf JSON Objekten.[8]

**Thymeleaf** Einen HTML Renderer, der HTML Dokumente von externen Services einfügen kann. [9]

**Spring Security** Eine Implementierung von *Cross-Origin Resource Sharing* (CORS) [Moz18a]

## 2.3 Portal Komposition

### 2.3.1 Einführung in die Architektur

**Architektur** Diese Alternative untersucht das Zusammenführen von bestehenden Webservices in eine Portal-Architektur. Der Portal-Server führt Teilseiten von anderen Services zusammen und baut aus diesen eine gemeinsames *Hypertext Markup Language* (HTML) [W3C17]-Dokument. Alle Links auf

[8] https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html#webflux-client
[9] https://www.thymeleaf.org/doc/articles/layouts.html#including-with-markup-selectors

den Fragmenten der Hintergrundservices müssen über das Portal funktionieren. Zwischen den Services werden HTTP GET- und POST-Anfragen zu den Services versendet. Die Antworten an den Browser enthalten gültige HTML Dokumente.

Die Alternative basiert auf dem Pattern *Server-side fragment composition*[Ric18f] von Richardson.

**Implikationen** Alle Services müssen HTML rendern können. Währenddem die Hintergrundservices die Fragmente mit den Daten rendern, muss der Portalserver die HTML-Dokumente der Seite zusammenführen in ein gemeinsames HTML-Dokument. Daraus folgt, dass alle Services einen HTML-Rendering unterstützen müssen. Der Portal-Server muss ausserdem die Integration von Drittseiten unterstützen.

**Vorteile** Der Autor erwartet die folgenden Vorteile durch die verwendete Architektur.

+ Die Integration bedarf keine Änderungen an den bestehenden Services.
+ Die Integration kann aufgebaut werden, währendem die Anwender die bestehenden Seiten nutzen können.

**Risiken** Es wird erwartet, dass die nachfolgenden Risiken bei der Implementierung der Alternative auftreten können.

– Das URL-Schema des Portal-Servers muss das Schema der Hintergrundservices übernehmen.
– Aus dem folgt, dass es Namenskollisionen der URL-Endpunkte auf dem Portal-Server entstehen können, die schwierig zu entfernen sind.

### 2.3.2 Implementation der Architektur

**Ablauf** Die Abbildung 2.2 zeigt das Sequenz-Diagramm der Implementierung von der Portal Komposition in der Beispielanwendung. Der Browser sendet einen HTTP GET Request für die Portalseite an den Server. Dann wird der Thymeleaf Page-Renderer mit dem Page-Template aufgerufen. Durch speziellen Template-Markup im Template, das sind Befehle in der Eingabedatei vom Renderer, wird Thymeleaf im Template aufgefordert die externen Seiten nachzuladen. Beim Ladeprozess sendet der Kompositionsserver ein HTTP-Request an den Hintergrundservice. Der Service antwortet mit einem HTML-Dokument. Wenn der Kompositionsserver die Antwort enthät, extrahiert Thymeleaf einen Teil des Dokumentes

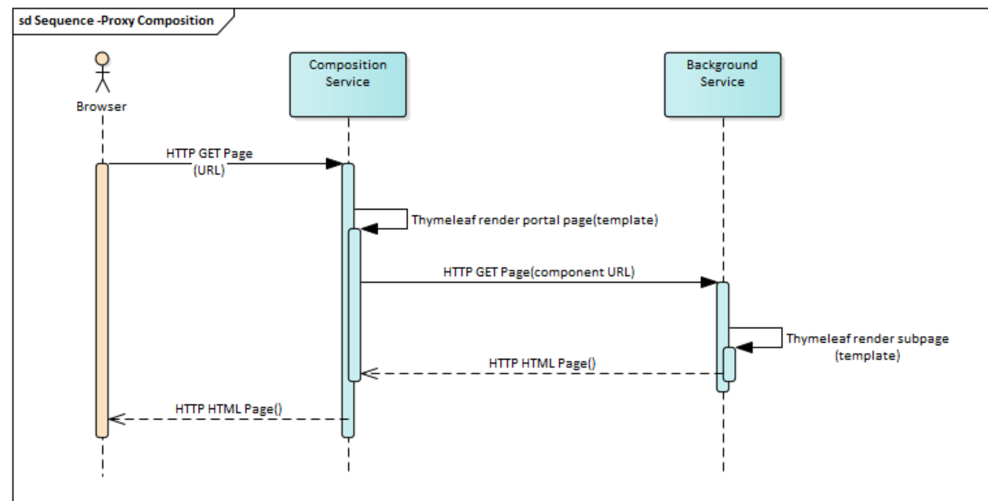


Abbildung 2.2: Interaktion bei Portal Komposition

und fügt ihn in die Portalseite ein. Abschliessend wird die Portalseite an den Client ausgeliefert.[10]

**Implementierung**

Der Source-Code-Block 2.10 zeigt den Code für die Portal-Komposition. Die Funktion wird für alle HTTP GET Requests auf den Kompositionspfad `/controller/kitchenDevice/` aufgerufen. Sie gibt Thymeleaf über den `model` Parameter `TARGET_SITE_KEY` die URL zur Ressource weiter und weisst Thymeleaf an die Seite zu rendern.

Die Thymeleaf Instruktion in Abbildung 2.4 weisst Thymeleaf an ein Fragment an dieser Stelle einzusetzen. `${targetSite}` ist die URL des Dokumentes aus dem Controller und `${targetElement}` ist der Thymeleaf Matcher [11] um das zu übernehmende HTML-Element aus der Antwort des Hintergrundservices zu finden.

### 2.3.3 Resultate aus der Programmierung der Alternative

**Chancen**

Die Implementation der Portal-Komposition hat die Einfachheit der Zusammenführung von HTML-Seiten gezeigt. Das Konzept kann auch sehr generell geschrieben werden und erlaubt somit mit wenig Code die Komposition von beliebig vielen Services.

[10] https://www.thymeleaf.org/doc/articles/layouts.html#including-with-markup-selectors

[11] https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#appendix-c-markup-selector-syntax

```java
@GetMapping("/kitchenDevice/**")
public Mono<String> kitchenDeviceService(@NotNull Model model
    , @NotNull HttpServletRequest request) {
    return matchPath(request).map(path -> {
        LOGGER.info("Path {} was requested", path);
        model.addAttribute(TARGET_SITE_KEY, "http://localhost
    :9603/controller/kitchenDevice/" + path);
        model.addAttribute(TARGET_ELEMENT_KEY, path.endsWith(
    "edit") ? "form" : "table");
        return "portal/frameHolder";
    });
}
```

Abbildung 2.3: PortalCompositionController.java, die Funktion zum Komponieren von Seiten

```html
<div th:insert="${targetSite} :: ${targetElement}">...</div>
```

Abbildung 2.4: Portal.html, das Page Template in welches andere Services integriert werden

**Risiken**

Die Verarbeitung von POST HTTP-Requests hat sich als sehr unhandlich herausgestellt. POST-Requests müssen empfangen werden und der Body der HTTP-Message muss gelesen werden, validiert, dann neu generiert und mit WebClient neu versendet werden.

Diese Operation muss für jeden POST-Request neu programmiert werden, da sich die Body-Inhalte zwischen POST-Messages untescheiden.

Diese Implementierung von der POST HTTP Method ist sehr nahe an der Lösung der Backend-Komposition. Bei POSTs verliert die Portal-Komposition ihre wichtigste Eigenschaft, sie ist weder einfach noch generell. Sondern muss jede Message einzeln behandelt und validiert werden um Parserfehler zu verhindern.

### 2.3.4 Diskussion der Alternative

Die Implementierung zeigt die Machbarkeit von einer Portal Integration auf. Es wurde im Zuge dieser Arbeit einen Prototyp erstellt und getestet.

Die Integration von HTTP POST Requests muss zurzeit individuell für jeden Request programmiert werden. Im Vergleich dazu können GET Re-

quests mithilfe von Thymeleaf auf eine Funktion pro Service generalisiert werden.

**Erkenntnisse** Die Erwartungen des Architekten (siehe 2.3) haben sich in der Beispielapplikation gezeigt. Die Komposition konnte ohne Änderungen an den dahinterliegenden Services ausgeführt werden. Es hat sich auch gezeigt, dass bei der Integration der Applikationen Namenskollisionen auftreten können.

In dieser Arbeit habe ich ferner gezeigt, dass eine enge Kopplung im Namensschema zwischen dem Integrationsserver und den Services entsteht.

**Ausblick** Der Architekt sieht folgende weiterführende Arbeiten:

**Edge Side Includes** Statt, dass die Integration in Spring Thymeleaf ausgeführt wird, kann diese auch in einem dafür optimierten Proxy ausgeführt werden. Als Beispiel seien hier Varnish[12] und Kong[13] zu nennen.

**POST Requests** Die einheitliche Behandlung von POST Requests durch den `PortalCompositionController`.

**Bidirektionale Integration** Diese Beispielapplikation kann keine integrationsspezifische Parameter an die Backendservices mitgeben. Mit der Weitergabe von Identifikationsmerkmalen können Kontext spezifische Dokumente generiert werden durch die Hintergrundservices.

## 2.4 Backend Komposition

### 2.4.1 Einführung in die Architektur

**Architektur** Ein Kompositionsserver führt die Daten der bestehenden Services in ein gemeinsames Frontend zusammen. Der Kompositionsserver kann Daten in verschiedenen Datenformaten erhalten und diese in ein HTML-Dokument rendern.

Die Alternative basiert auf dem Pattern *Server-side fragment composition* von Richardson.

**Implikationen** Die Services müssen eine Schnittstelle zur Verfügung stellen über welche

[12] https://varnish-cache.org/
[13] https://github.com/Kong/kong

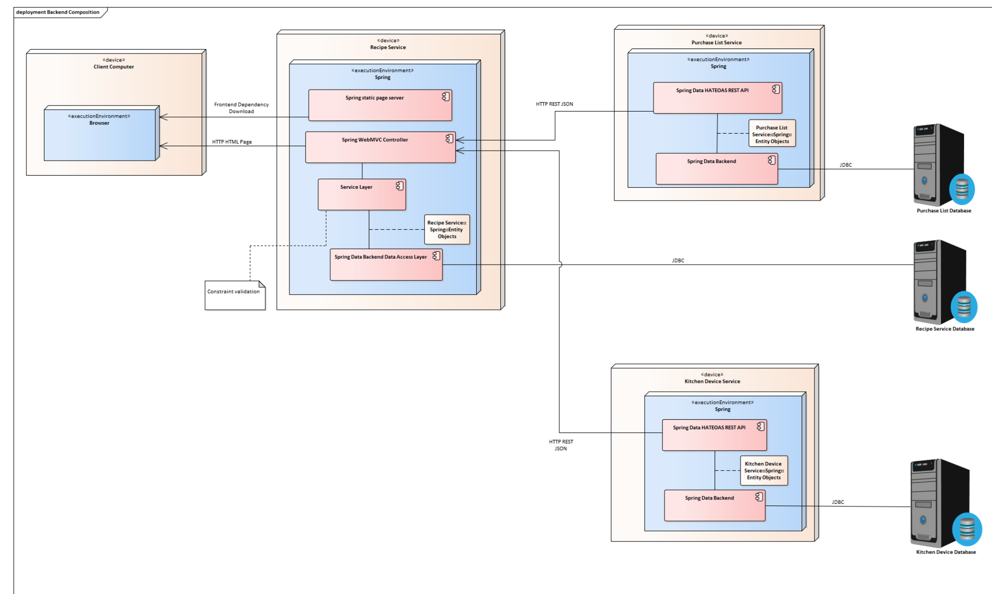


Abbildung 2.5: Backend Komposition

der Kompositionsserver auf die Daten des Services zugreifen kann. Diese Schnittstellen können unterschiedlichen Übertragungsprotokollen und Datenformaten besitzen.

Der Autor erwartet folgende Vorteile durch die verwendete Alternative.

+ Die Integration bedarf keine Änderungen an den Backendservices.

+ Die Integration kann mit verschiedenen Protokollen und Datenformaten umgehen.

Es wird erwartet, dass diese Risiken auftreten

– Das Frontend ist in einem getrennten Projekt vom Backend Service.

– Die *Data Transfer Objects* (DTO) [Fow03] müssen auf dem Kompositionsserver und auf dem Backendservice gleichzeitig aktualisiert werden.

– Frontend Code und Backend Code werden auf zwei Projekte gespalten.

### 2.4.2 Implementation der Architektur

**Sequenz**

Abbildung 2.6 zeigt das Sequenz-Diagramm der Implementierung von der

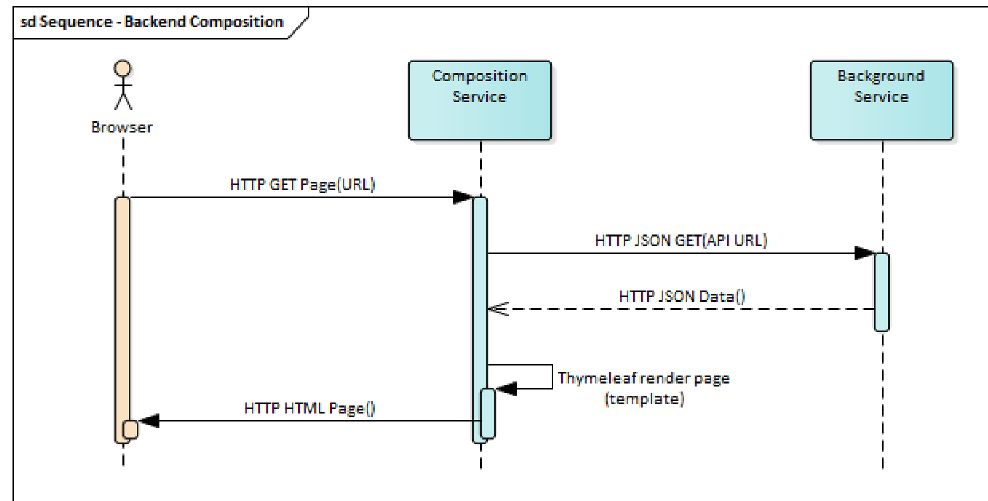


Abbildung 2.6: Interaktion bei Backend Komposition

```java
@GetMapping("/kitchenDevice/list")
public @NotNull Mono<ModelAndView> listKitchenDevice() {
    return kitchenDeviceService.get()
            .uri(KITCHEN_DEVICE_ROOT_PATH)
            .accept(MediaType.APPLICATION_JSON)
            .retrieve().bodyToMono(KitchenDeviceRoot.class)
            .map(kitchenDeviceRoot ->
                    new ModelAndView("kitchenDevice/list").
    addObject(PLURAL_MODEL_KEY, kitchenDeviceRoot)
            );
}
```

Abbildung 2.7: BackendCompositionController.java

Backend Komposition in der Beispielanwendung. Der Browser sendet einen HTTP GET Request an den Server. Der Server sendet einen RESTful HTTP Request für die Daten an den Backgroundservice. Nachdem der Backgroundservice die Daten geliefert hat, werden diese an Thymeleaf weitergegeben. Thymeleaf rendert ein Template mithilfe der empfangenen Daten und sendet das generierte Frontend an den Browser.

**Implementation**

Der Source-Code-Block 2.7 zeigt den Code für die Backend Komposition. Jede Frontend-Seite hat eine eigene Controller Methode im Kompositionsservice, die die Daten von den Backend Services lädt und das HTML-Dokument rendert. Diese muss für jede Seite auf dem Kompositionsserver spezifisch geschrieben werden.

### 2.4.3 Resultate aus der Programmierung der Alternative

**Chancen** Die Backend Komposition stellte sich in der Implementation als sehr flexibel heraus.

**Risiken** Die Risiken liegen in der Duplizierung von Code zwischen dem Application und dem Kompositionsserver. Zum Beispiel müssen die DTO auf beiden Servern auf gleichem Stand gehalten werden.

Der Autor sieht ein Risiko bei der Aktualisierung des Übertragungsprotokolls, dass die Aktualisierung der DTO des zweiten Services vergessen wird.

### 2.4.4 Diskussion der Alternative

Die Beispielimplementierung zeigt die Machbarkeit der Backend Composition auf. Ein Prototyp ist in dieser Arbeit erstellt worden. Anhand des Prototyps wurde dann der Kriterienvergleich ausgeführt.

Da jede Page eine eigene Controller Methode benötigt, ist es die Meinung des Architekten, dass die Backend Komposition Alternative eine Zentralisierung des Webservers darstellt. Das Frontend eines jeden Services wird herausgenommen und von dem Kompositionsserver ausgelieftert. Dies führt zu Risiken, die während der Implementierung der Alternative aufgetreten sind.

**Erkenntnisse** Die Erwartungen des Architekten (siehe 2.4.1) haben sich in der Beispielapplikation gezeigt. Das Frontend liegt in einem getrennten Build-Projekt. Dadurch ist es vermehrt zu Versionskonflikten gekommen. Beispielsweise sind die DTO nicht mehr aktuell gewesen auf dem Kompositionsserver. Auch der Umgekehrte Fall ist denkbar, wenn das Frontend verändert wird und der Applikationsserver nicht gleichzeitig aktualisiert wird.

## 2.5 Frontend Komposition

### 2.5.1 Einführung in die Architektur

**Architektur** Die Frontend Komposition beschreibt das Zusammenführen von bestehenden Webservices in einem Javascript Frontend, das Frontend führt die Daten der Services zusammen im UI-Layer.

Die Alternative basiert auf dem Pattern *Client-side ui composition* von Richardson.

**Implikationen** Die Services brauchen keinen Frontend-Layer mehr. Die Services stellen ein Web-API bereit auf welches das Javascript-Frontend zugreifen kann.

Der Autor erwartet die folgenden Vorteile durch die verwendete Architektur.

- Die TTFB der Applikation schneller ist als bei den Alternativen
- Die UI kann auch bei Nichtverfügbarkeit der Services geladen werden
- Die UI kann den Ladefehler anzeigen

Es wird erwartet, dass die nachfolgenden Risiken bei der Implementierung der Alternative auftreten können.

- Alle Services müssen offen ans Internet gelegt werden

### 2.5.2 Implementation der Architektur

**Ablauf** Die Abbildung 2.9 zeigt das Sequenz-Diagramm der Implementierung von der Frontend Komposition in der Beispielanwendung. Der Browser ladet zuerst das statische Webdokument und rendert dieses. Sobald der Javascript geladen ist, wird ein asynchroner RESTful HTTP Request an den Service gesendet um die Daten der Ansicht zu laden. Sobald die Daten angekommen sind, werden diese ins Frontend weitergereicht und durch das Frontend angezeigt.

**Implementation** Währenddem in den vorherigen Kapitel von Frontend als Webserver vorgeschaltet vor den Applikationsserver die Rede war, wird als Frontent in diesem Kapitel eine selbständige Browserapplikation genannt, die wiederum in ein Frontend und ein Backend aufgeteilt ist. Diese UI-Applikation im Browser sollte auch ein sauberes Layering einhalten.

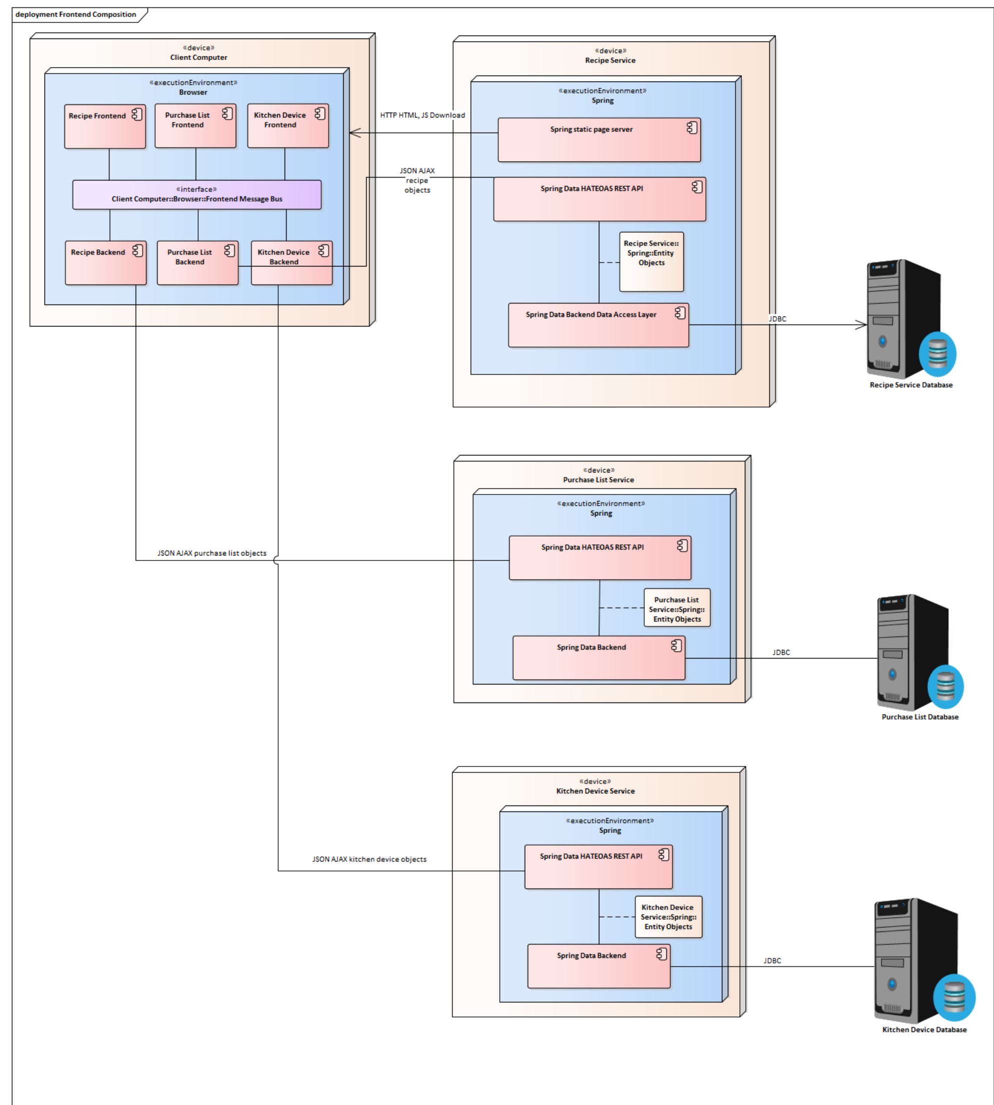


Abbildung 2.8: Frontend Komposition

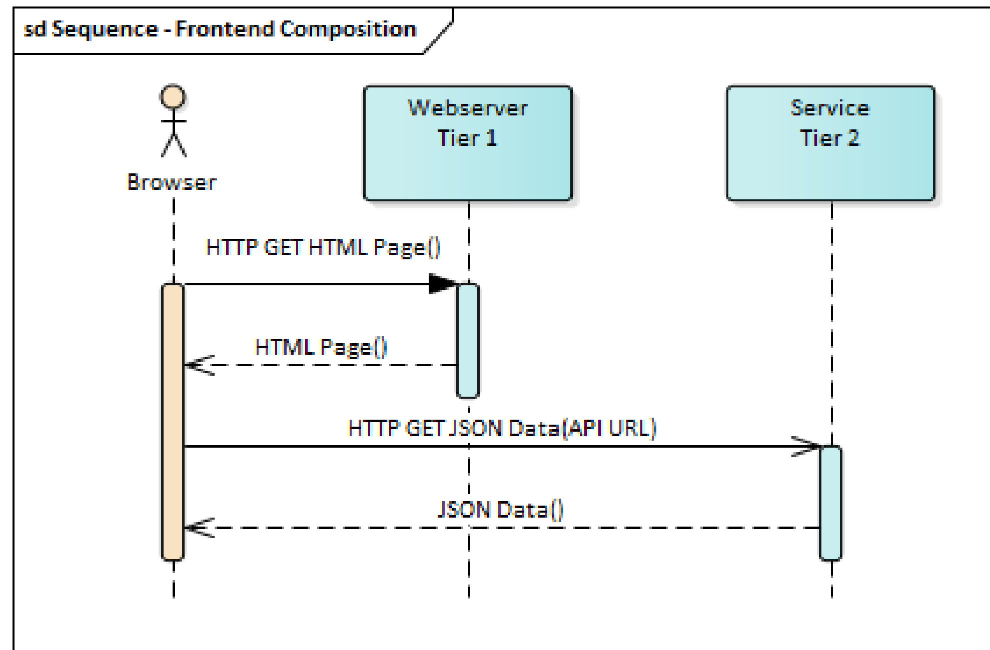


Abbildung 2.9: Interaktion bei Frontend Komposition

```
return matchPath(request).map(path -> {
    LOGGER.info("Path {} was requested", path);
    model.addAttribute(TARGET_SITE_KEY, "http://localhost
    :9603/controller/kitchenDevice/" + path);
    model.addAttribute(TARGET_ELEMENT_KEY, path.endsWith("
    edit") ? "form" : "table");
    return "portal/frameHolder";
});
```

Abbildung 2.10: PortalCompositionController.java, die Funktion zum Komponieren von Seiten

Zuunterst liegt der Backend-Layer, der die Datenbereitstellung für das UI übernimmt. In der Mitte kann ein Event-Bus eingerichtet werden, damit die Frontends mit dem Backend kommunizieren können. [Söd17]

In meiner Applikation habe ich die Backends und die Frontends umgesetzt. Diese können miteinander kommunizieren. Der Event-Bus wurde nicht mehr umgesetzt als Teil der Projektarbeit.

### 2.5.3 Resultate aus der Programmierung der Alternative

**Chancen** Die Chancen in der Frontend Komposition sieht der Architekt in der Integration von Services verschiedener APIs.

**Risiken** Die Risiken bestehen in der gemeinsamen Verantwortung und Zuständigkeit verschiedener Teams auf einen gemeinsamen Source-Code für das Frontend. Der Code wird von verschiedenen Entwicklungsteams bearbeitet und hat keinen Eigentümer der die Verantwortung über den Code des Frontends trägt.

### 2.5.4 Diskussion der Alternative

Die Implementierung zeigt die Machbarkeit von einer Frontend Integration auf. Es wurde im Zuge dieser Arbeit einen Prototyp erstellt und getestet.

Innerhalb der vorgegeben Projektdauer konnten nicht alle Features eines Microfrontends umgesetzt werden. Der Autor hat die nachfolgenden Features nicht umgesetzt und könnten in einer Nachfolgearbeit weitergeführt werden:

- Die Trennung verschiedener Javascript-Frameworks innerhalb einers HTML-Dokumentes.
- Der Einsatz eines Event-Bus für die Kommunkation zwischen den Framework-Impementationen
- Das späte Nachladen der Frontend-Komponenten nach dem Aufbau der Website

**Ausblick** Der Architekt sieht folgende weiterführende Arbeiten:

**GraphQL** Ein Ansatz für Frontend zu Backend Kommunikation um die Bandbreite zu reduzieren [14]

**Single SPA** Dieses Projekt separiert verschiedene Frontend-Frameworks voneinander. Es erlaubt zum Beispiel eine Koexistenz von Angular und React Komponenten in der gleichen HTML Seite. [15]

**Shared Event Bus** Erlaubt die Kommunikation verschiedener getrennter (siehe Single SPA) Frontend-Komponenten auf der HTML Seite. [16]

[14] https://graphql.org/
[15] https://github.com/CanopyTax/single-spa
[16] https://github.com/chrisdavies/eev

# 3 Resultate der Arbeit

Im folgenden werden die Resultate der Arbeit dargestellt. Zu Beginn habe ich die Patterns von Richardson studiert. Aus Ihnen eigene Implemenationen ausgearbeitet (siehe 2.3 ff.). Diese Alternativen in einem Prototyp ausprogrammiert und dann mithilfe des Kriterienkatalogs verglichen. Aus dem Vergleich werde ich am Schluss Erkenntnisse ziehen.

## 3.1 Eignung der Microservice Patterns von Richardson

Als Basis dieser Arbeit habe ich in einer Vorstudie die Patterns von Richardson studiert. Die Beschreibungen der Pattern sind nicht vollständig ausgearbeitet. [Ric18g] Aus den Titel der Patterns konnte der Architekt die Intention des Textes herauslesen. Ohne den Text ist es nicht möglich auf die Details der Patterns zu referenzieren. Aus diesem Grund wird im weiteren Verlauf des Textes darauf verzichtet auf Einzelheiten der Patterns zu referenzieren und davon ausgegangen, dass der Inhalt der Patterns die Interpretation des Autors darstellt.

Die Arbeit vergleicht die folgenden Patterns von Richardson

- *Client-side ui composition*[Ric18c]
- *Server-side fragment composition*[Ric18f]

Es ist dem Architekten nicht bekannt weshalb Richardson die Patterns unstimmig benannt hat. Der Autor versteht unter den Pattern die Zusammenführung der Daten von verschiedenen Services. Der Autor versteht den Unterschied so, dass die Datenzusammenführung beim einen Pattern auf dem Client, dem Webbrowser des Users, und beim zweiten auf einem Kompositionsserver ausgeführt wird.

Nach Projektplan sollte auch das *API gateway pattern*[Ric18a] untersucht werden, dieses wurde im Verlauf des Projektes verworfen wegen dem Verwurf der Implementation des Patterns.

| Git | package manager |
|---|---|
| Java JDK 8 | https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html |
| NodeJS | https://nodejs.org/en/ |

Tabelle 3.1: Abhängigkeiten, die vorherig, separat installiert werden müssen

In einer Folgearbeit kann das Pattern *Backend for Frontend pattern*[Ric18b] weiter vertieft werden, das eine Komposition aufgrund der Anforderungen des Frontends ermöglicht.[New15]

## 3.2 Beispielapplikation

### 3.2.1 Installation der Beispielapplikation

Abhängigkeiten die vorher installiert werden müssen:

#### Execute under Mac & Linux

Clone repository

```
git clone https://github.com/mattbaumann/Micro-Frontends.git
    && cd Micro-Frontends
```

Make gradlew executable

```
chmod +x ./gradlew
```

Starting backend services

```
./gradlew bootRun
```

Starting frontend

```
cd frontend/monolithic
npm install
npm run serve
```

Building documentation

```
./gradlew javadoc
./gradlew asciidoc
```

### Execute under Windows PowerShell

Clone the respority

```
git clone https://github.com/mattbaumann/Micro-Frontends.git
    && cd Micro-Frontends
```

Starting backend services

```
./gradlew bootRun
```

Starting frontend

```
cd frontend/monolithic
npm install
npm run serve
```

Building documentation

```
./gradlew javadoc
./gradlew asciidoc
```

### Entwicklungsumgebung Eclipse

Building Eclipse Project

```
./gradlew eclipse
```

Unter Eclipse kann *File > Import Project* ausgewählt werden und das Projekt als *Existing Project* importiert werden.

### Entwicklungsumgebung IDEA

Unter dem Startbildschirm von IDEA kann *Import Project* ausgewählt werden und der Projekt-Ordner ausgewählt werden. Beim Import-Wizard *Existing Code-Model* und dann *Gradle* auswählen. Den Wizard dann abschliessen.

## 3.3 Kriterien-basierter Vergleich von Architektur Optionen

### 3.3.1 Einführung

Dieses Kapitel vergleicht die Architektur-Alternativen anhand des Kriterienkatalogs.

Die Schlussfolgerungen können aus der Beispielapplikation, aus der Theorie oder nicht gezogen werden. Aus diesem Grund habe ich die Schlussfolgerungen in verschiedene Güteklassen eingeteilt.

**Literatur** Die Werte des Vergleichs konnten aus der Literatur verweist werden.

**Implementation** Die Werte des Vergleichs konnten durch die Beispielsapplikation gezeigt werden.

**Theorie** Die Werte des Vergleichs konnten theoretisch durch eine Schlussfolgerung aufgezeigt werden.

**Meinung** Die Werte entsprechen der Meinung des Architekten.

**Nicht möglich** Der Vergleich konnte nicht gezeigt werden.

### 3.3.2 Migrationsarbeit

#### Ausarbeitungszeit des Konzeptes: Wie lange brauchte ich das Konzept auszuarbeiten?

Diese Werte wurden durch die *Implementation* bestimmt.

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| 9 Stunden | 4 Stunden | 4 Stunden |

Bei der Frontend-Komposition sind viele Technologien notwendig gewesen um die Komposition aufzubauen. Ich hatte noch nicht mit diesen Technologien gearbeitet und musste mich in die Thematik einlesen.

Der Kompositionsserver für die anderen Alternativen konnte mit Thymeleaf und WebClient[1] aufgebaut werden.

[1] https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-webclient.html

#### Ausarbeitungszeit des Prototyps: Wie lange dauert die Ausarbeitung des ersten Prototyps der Architektur-Alternative?

Diese Werte wurden durch die *Implementation* bestimmt.

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| 8 Stunden | 10 Stunden | 4 Stunden |

Die Portal-Komposition hat eine lange Entwicklungszeit, weil es schwierig war Thymeleaf zu konfiguriren, dass der Renderer die Fragmente von einem externen Service holt.

Die Frontend-Komposition hatte viele Technologien involviert mit welchen ich noch nicht gearbeitet hatte. Aus diesem Grund ist die produktivität geringer gewesen als bei der Backend-Komposition.

#### Denkfehler im Code: Wie viele Denkfehler hatte ich im Code?

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| 3 Fehler | 2 Fehler | 0 Fehler |

Die Frontend-Komposition hat sich als schwierige Technologie herausgestellt. Dank der Entscheidung Typescript[2] im Frontend einzusetzen konnten weitere, schwerwiegende Fehler frühzeitig erkannt werden.

Die Portal-Komposition hatte zweimal schwerwiegende Fehler, die wegen ungenauem Loggings der Thymeleaf Bibliothek schwierig zu erkennen waren.

#### Schrittweiser Übergang: Kann eine externe Seite im Micro-Frontend angezeigt werden?

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| Ja (Literatur) | Ja (Implementation) | Nein (Implementation) |

Bei der Frontend-Komposition kann mit einem IFrame-Element eine externe Seite eingebunden werden.

Die Portal-Komposition kann mittels der Fragment-Auflösung von Thymeleaf externe Inhalte in die Page rendern.[Thy18]

[2] https://www.typescriptlang.org/

Die Backend-Komposition nutzt den WebClient um die Daten beim Hintergrundservice abzurufen und rendert diese im Kompositionsserver mithilfe von lokalen Thymeleaf-Templates. Diese Technologie kann ohne das Zurückgreifen auf die Portal-Komposition keine Funktionsblöcke von der alten Website übernehmen.

#### Schrittweiser Übergang: Kann die alte Seite übernommen werden und im Aussehen auf die neue Seite angepasst werden?

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| Nein (Literatur) | Ja (Implementation) | Nein (Implementation) |

Bei der Frontend-Komposition mit IFrames die Site nicht umgestaltet werden. [MDN18]

Es können Bereiche von anderen Websiten ausgespart werden und dann mit eigenen *Cascading Style Sheets* (CSS) [W3C18a] Regeln die Ansicht auf die neue Website angepasst werden.

Bei der Backend-Komposition kann das UI der alten Website nicht übernommen werden, da der Kompositionsserver kein HTML verarbeitet.

### 3.3.3 Aktualisierbar in Produktion

#### Unterbrechungsfreie Aktualisierbarkeit: Kann das Microfrontend zusammen mit dem Microservice ausgetauscht werden?

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| - (Nicht angesehen) | Ja (Implementation) | Nein (Implementation) |

Im Frontend wurde diese Fragestellung nicht weiter verfolgt um im Zeitplan zu bleiben.

Bei der Portal-Integration muss nur der Applicationserver mit den UI-Fragmenten ausgetauscht werden.

Bei der Backend-Komposition liegen die Layout-Fragmente der Microfrontends auf dem Kompositionsserver, darum müssen sowohl die Anwendungsserver als auch der Kompositionsserver ausgetauscht werden.

### 3.3.4 Unabhängigkeit der Services

**Ausser der Abhängigkeit auf den Applikationserver darf das Frontend keine weitere Verbindungen benötigen.**

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| Ja (Implementation) | Ja (Implementation) | Ja (Implementation) |

Alle Implementationen brauchen eine Abhängigkeit auf den Applikationserver um die Daten zu laden. Sie kommen mit dieser Abhängigkeit aus.

**Der Service darf keine Abhängigkeiten auf andere Services besitzen in der „Build-Phase“.**

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| Ja (Implementation) | Ja (Implementation) | Ja (Implementation) |

Alle Alternativen haben keine Build-Dependency auf andere Services.

Diese Forderung hängt von der verwendeten Kommunikationsbibliothek ab. Beispielsweise braucht WebClient keine Informationen über das Nachrichtenschema. Im Gegensatz dazu generiert beispielsweise die Bibliothek WCF[3] zur Build-Zeit Code zum Verbindungsaufbau und braucht deshalb Informationen über das Message-Format des Kommunikationpartners.[Mic17]

Diese Forderung ist sehr Technologieabhängig, es kann jedoch ausgeschlossen werden, dass das Pattern eine solche Abhängigkeit verursacht.

**Die Microfrontend Architektur darf nicht die Unabhängigkeit der Microservices erweichen**

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| Nein (Implementation) | Nein (Implementation) | Nein (Implementation) |
| Ja (Literatur) | Nein (Meinung) | Nein (Meinung) |

[3] https://docs.microsoft.com/en-us/dotnet/framework/wcf/

Die Implementation ist so geschrieben, dass die Frontend-Komponenten nicht in eigenen Projekten sind. Damit müssen alle Microfrontends ersetzt werden beim Deployment des Artefakts.

Im Frontend könnte mit *Polymer Project*[Goo18] und *Web Components*[Moz18b] die Trennung von Komponenten in einzelne Artefakte möglich sein. Dies wurde innerhalb dieser Arbeit nicht untersucht und könnte einen Ansatz für eine Folgearbeit bieten.

Aus technologischen Gründen unter Spring kann der Kompositionsserver nicht in verschiedene Deployment Units aufgetrennt werden. Aus diesem Grund sind die Frontend-Implementationen nicht unabhängige Betriebsystemprozesse und Komponenten. Es ist ein Server bei welchem alle UI-Ressourcen kompiliert werden und nicht austauschbar sind.

#### Services müssen einzeln aktualisierbar sein.

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| Ja (Implementation) | Ja (Implementation) | Ja (Implementation) |

Die Services haben in meiner Implementation keine Abhängigkeiten untereinander.

### 3.3.5 Effekt auf die Geschwindigkeit

#### Der Nutzer hat das Gefühl einer flüssigen Ladegeschwindigkeit

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| Ja (Implementation) | Ja (Implementation) | Ja (Implementation) |

Alle Implementationen laden im Millisekundenbereich. Die Ladezeit der Frontend-Komposition ist leicht spürbar, da fast eine Sekunde. Nach dem Laden der UI ist diese Variante die schnellste, da sie kein Backend-Rendering benötigt.

**Die Ladegeschwindigkeit muss so schnell wie möglich sein.**

**Die Zeit zur Auslieferung der Seite (TTFB) muss so schnell wie möglich sein.**

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| Ja (Implementation) | Ja (Implementation) | Ja (Implementation) |

| Messwert | Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|---|
| median | 0.032 ms | 0.193 ms | 0.183 ms |
| std. Dev | 0.051 | 0.023 | 0.044 |
| mean | 0.066 ms | 0.199 ms | 0.1926 ms |
| No. Tests | 5 | 5 | 5 |

Die ersten drei Resultate beschreiben die gesamte Ladezeit einer Ansicht der Kompositionsvariante.

| Messwert | Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|---|
| median | 990 kB | 363 kB | 363 kB |
| std. Dev | 0.577 | 1.41 | 1.26 |
| mean | 990 kB | 363 kB | 363 kB |
| No. Tests | 4 | 4 | 3 |

Die mittleren drei Resultate sind die Anzahl übertragener Bytes der Kompositionsvarianten. Bei den Server-Integrationen wird die Grösse einer Ansicht gemessen und bei der Frontend-Komposition die Grösse der gesammten Applikation.

| Messwert | Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|---|
| median | 0.027 ms | 0.017 ms | 0.018 ms |
| std. Dev | 0.003 | 0.0024 | 0.015 |
| mean | 0.027 ms | 0.0165 ms | 0.018 ms |
| No. Tests | 4 | 4 | 4 |

Die untersten Datensätze beschreiben beschreiben die TTFB der Varianten.

Wie aus der „No. Tests“ Zeile der Ergebnisse zu entnehmen ist, ist in dieser SA eine sehr kleine Anzahl Tests verrichtet worden. Der Autor geht davon aus, dass die Anzahl zu klein ist um genaue Aussagen über die Geschwindigkeiten zu treffen. Die Tabelle sollte für einen groben Überblick

genommen werden, ist aber nicht als Feinvergleich der Implementierungen zu nehmen.

#### Die Zeit zwischen der Auslieferung und dem Fertigstellen des Renderings (Rendering Time) muss so schnell wie möglich sein.

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| - (Nicht möglich) | - (Nicht möglich) | - (Nicht möglich) |

Ich konnte diesen Wert nicht messen, da in modernen Browsern das Rendering parallel zum Laden des Source-Codes ausgeführt wird. Wegen der Parallelität lässt sich nicht entscheiden, ob der Browser auf Daten gewartet hat oder der Renderer die Seite aufbaute.

### 3.3.6 Theoretische Skalierbarkeit

#### Die Kommunikation über das HTTP-Protokoll mit dem Applikationsserver muss stateless sein

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| Ja (Implementation) | Ja (Implementation) | Ja (Implementation) |

In keinem Service wurde ein State eingeführt. Der Kompositionsserver ist ein Domain-spezifischer stateless Proxy.

#### Neue Instanzen können im Webserver Tier hochgefahren werden um die Anfragelast zu bewältigen.

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| Ja (Implementation) | Ja (Implementation) | Ja (Implementation) |

Weil die Kommunikation mit dem Browser stateless ist, ist dies zu jeder Zeit möglich.

#### Der Webserver darf keinen State, inklusive keine Session, des Nutzers speichern.

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| Ja (Implementation) | Ja (Implementation) | Ja (Implementation) |

Die Backend und Portal Integration sind stateless Proxies. Sie besitzen keine Informationen über vorherige Anfragen und müssen keine Annahmen über Anfragen in der Zukunft treffen können. Sie bekommen eine Anfrage und senden diese weiter an die Applikationsserver.

Der Webserver für die Frontend-Komposition ist stateless und liefert die HTML-Seite aus.

Die Frontend-Integration selber ist nicht stateless. Dies ist auf die Wahl des React-Frameworks zurück zu führen. Die Implementation muss die Daten nicht mehr anfragen, wenn diese bereits im Memory des Clients liegen. [Fac18]

### 3.3.7 Informationssicherheit

#### Die Vertraulichkeit (confidentiality) der Daten muss im Transport sichergestellt werden

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| Ja (Meinung) | Ja (Meinung) | Ja (Meinung) |

Durch *transport layer security* (TLS) [IET18] kann die Verbindung zwischen Client und Server verschlüsselt werden. [IET18] Dies wurde innerhalb des Projektrahmens nicht mehr realisiert. Aus diesem Grund, ist es die Meinung des Architekten, dass die Transportsicherheit bei allen Verfahren möglich ist. Der Autor weist auf den Aufbruch der TLS-Verbindung beim Kompositionsserver hin, dies kann ein Sicherheitsrisiko darstellen, da wegen dem Aufbruch der Verbindungen, auf diesem Server alle Verbindungen lesbar werden.

#### Die Integrität der Daten muss sichergestellt werden

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| Ja (Implementation) | Ja (Implementation) | Ja (Implementation) |

Die Integrität der Daten kann sichergestellt werden durch Einsatz von Verschlüsselung zwischen den Services untereinander und dem Client. Wie bei der vorherigen Fragestellung erwähnt, können alle Alternativen adäquat verschlüsselt werden.

#### Die Verfügbarkeit (availability) muss zu jeder Zeit sichergestellt sein

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| Nein (Implementation) | Nein (Implementation) | Nein (Implementation) |

Keine Alternative garantiert die Verfügbarkeit der Daten. Die Frontend-Komposition hat den Vorteil, dass sie ohne einen Kompositionsserver auskommt und eine Komponente weniger hat, die nicht verfügbar sein kann.

#### Die Services dürfen keine Daten und keine Frontend-Fragmente ausliefern an unberechtigte, dritte Personen.

| Frontend-Komposition | Portal-Komposition | Backend-Komposition |
|---|---|---|
| Nicht untersucht (Implementation) | Nicht untersucht (Implementation) | Nicht untersucht (Implementation) |

Das AAA wurde innerhalb dieser Arbeit nicht untersucht und kann als Fortsetzung dieser Arbeit aufgegriffen werden.

## 3.4 Diskussion der Ergebnisse des Vergleichs der Alternativen

Der Autor wertet die Ergebnisse des Vergleichs der Alternativen nicht. Die optimalste Alternative nach dem Kriterienvergleich muss nicht die beste Alternative sein. Die Bewertung der Alternativen hängt stark vom Anwendungsfall ab und muss einzeln in jedem Fall untersucht werden. Diese Arbeit evaluierte Fragestellungen, die für eine solche Bewertung herbeigezogen werden können.

Die Alternativen haben alle Chancen und Risiken.

Der Autor wird im weiteren Verlauf eine Diskussion führen über ausgewählte Erkenntnisse aus dem Vergleich der Alternativen.

### 3.4.1 Ausarbeitungszeit der Prototypen

**Backend-Komposition**

Die Ausarbeitungszeit der Prototypen zeigte eine klare Reihenfolge von der Komplexität auf. Die *Backend-Komposition* ist die einfachste Komposition zum implementieren.

Statt die Daten aus der Datenbank zu laden, werden die Daten von einem Hintergrundservice geladen. Die Restlichen Arbeitsschritte, Daten aus Object Tree extrahieren und Template rendern sind analog zu lokalem Datenrendering.

**Portal-Komposition**

Die Portal-Komposition ist die zweitschwierigste Komposition zum implementieren. Die Fragmenteinbindung von externen Services in Thymeleaf stellte ein schwieriges und fehleranfälliges Unterfangen dar. Dies führte zu langer Fehlersuche.

**Frontend-Komposition**

Die Frontend-Komposition war die schwierigste Komposition, da es sehr viele Technologien beinhaltete. Über den Server musste CORS konfiguriert werden und auf dem Client musste ich ReactJS lernen.

### 3.4.2 Unterbrechungsfreie Aktualisierbarkeit: Kann das Microfrontend zusammen mit dem Microservice ausgetauscht werden?

**Portal Komposition**

Mit der *Portal Komposition* konnte gezeigt werden, dass ein Frontend auf dem Applikationsserver abgelegt werden kann. Das Frontend kann mit dem Austausch des Applikationsservers in einem Schritt aktualisiert werden.

Weil bei der Portal Komposition die HTML-Dokumente lokal gerendert werden, ist eine Aktualisierung des Domain-Models nur die Aktualisierung des Applikationsservers nötig.

**Backend Komposition**

Das UI der Backend Komposition kann nicht mit der Aktualisierung des Applikationsservers ersetzt werden. Bei dieser Alternative muss der Webserver ersetzt werden.

Solange das Übertragungsprotokoll nicht ändert, muss der Applikationsserver nicht ersetzt werden. Wenn die Domain-Model der Applikation

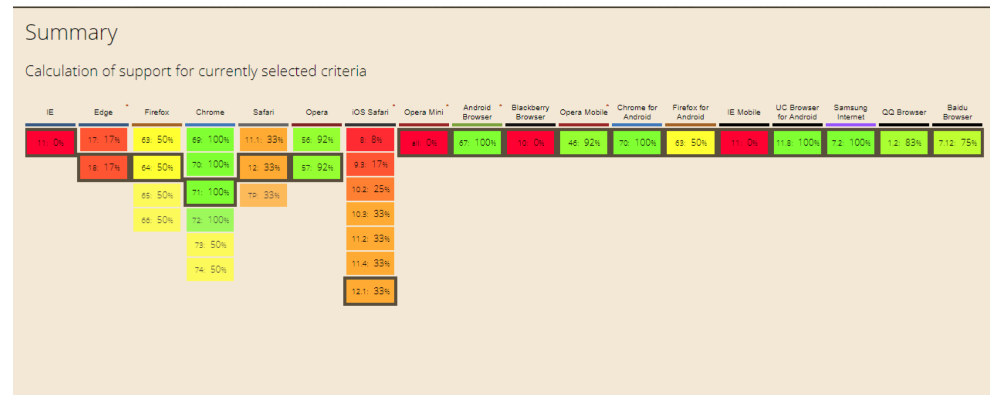


Abbildung 3.1: caniuse Abfrage für WebComponents

ändert muss der Applikationsserver, der Webserver und das Übetragungsprotokoll zwischen den zwei Tiers aktualisiert werden.

**Web Compontents** Die Meinung des Autors ist, dass mit der Nutzung von WebComponents eine Microfrontent-Architektur im Browser aufgebaut werden kann. Dann können WebComponents auf den Applikationsserver abgelegt werden. Durch den Einsatz von *HTML Imports*[W3C18b] kann das HTML-Dokument auf die Services aufgeteilt werden und durch den Browser zusammengefügt werden.

Die WebComponents Technologie ist zur Zeit der Abgabe dieser Arbeit experimentell und wenig unterstützt durch die Browserlandschaft. Dies zeigte eine *caniuse*[4] Abfrage. Die Auswertung der Brwoserunterstützung aller Spezifikationen zur Technologie ist unter 3.1 ersichtlich. Chrome und seine Derivate implementieren die Spezifikation schon. Andere Browserhersteller sind noch nicht soweit, dass diese Technologie produktiv einsetzbar ist.

**Zusammenfassung** Der Autor schätzt die Einfachheit der Portal Komposition. Die Backend Komposition hat gute Trennung der Belange. Der Autor geht davon aus, dass die Frontend-Komposition mit WebComponents in der Zukunft der Standard für Microfrontends wird. Bis dahin, sind alle Alternativen gute Lösungen.

### 3.4.3 Die Microfrontend Architektur darf nicht die Unabhängigkeit der Microservices erweichen

**Aufgetretene Risiken** Diese Anforderung konnte die Beispielapplikation nicht vollständig erfül-

[4] https://caniuse.com/#search=web%20components

len. Der Entwickler konnte zeigen, dass der Code aller Implementationsformen voneinander unabhängig ist. Aufgrund von Projektbeschränkungen konnte der Architekt die Integrationen nicht in verschiedene Services auftrennen um eine Three-Tier-Architektur aufzubauen.

**Chancen** Eine weiterer Anknüpfungspunkt für eine Folgearbeit ist die Analyse und prototypische Implementation von WebComponents. [Moz18b]

**Zusammenfassung** Ich konnte im Zuge dieser Arbeit die Möglichkeit aufzeigen, dass es möglich ist alle Kompositionsalternativen aufzutrennen und überlasse die Auftrennung einer Folgearbeit.

### 3.4.4 Erkenntnisse über mögliche Technologien

**WebComponents** Für den Browser gibt es die WebComponents-Technologie. [Moz18b] Für die Diskussion der Chancen und Risiken von WebComponents siehe Sektion 3.4.2.

**server-side includes** Die Technologie *server side includes* (SSI)[5] wird im Zusammenhang mit HTML-Fragmentenkomposition häufig erwähnt. Die Konvention erlaubt unter Apache Server[6] das Zusammenfügen von Websitenteile verschiedener Applikationen. Diese Technologie steht unter Spring nicht zur Vefügung und wurde deshalb in dieser Arbeit nicht weiter untersucht.

**edge-side includes** Die Technologie *edge side includes* (ESI) [W3C01][7] kann verwendet werden um Inhalte verschiedener URL's auf einem Proxy-Server in ein gemeinsames HTML-Dokument zusammen zu führen. Diese Technologie wird stark von Akamai[8] und fastly[9] genutzt um Bilder und weitere Cache Inhalte in Kundenseiten zu integrieren.[Wis14] Weil die Proxy-Variante nicht weiterverfolgt wurde ist diese Technologie nicht näher betrachtet worden in der Arbeit.

## 3.5 Ausblick auf weiterführende Arbeiten

Der Autor sieht folgende Anknüpfungspunkte um diese Arbeit weiter zu führen.

[5] https://www.w3.org/Jigsaw/Doc/User/SSI.html
[6] https://httpd.apache.org/
[7] https://www.akamai.com/fr/fr/support/esi.jsp
[8] www.akamai.com
[9] https://www.fastly.com/

Der Vergleich der vorgestellten Architekturen mithilfe weiterer Kriterien. Beispielsweise kann die Leistungsfähigkeit der Alternativen mit Gatling[10] getestet werden. Auch ist unklar ob eine automatisierte continous integration[11][Pau+17] mit diesem Konzept möglich ist.

Die Nutzung von WebComponents[Moz18b] und HTML Imports [W3C18b] um die Microfrontends weiter zu separieren. Unter *Micro frontends—a microservice approach to front-end web development*[Söd17] wird vorgeschlagen einen Message-Bus für die Kommunikation zwischen den Microfrontends zu nutzen.

Eine andere Möglichkeit ist eine Untersuchung des *Backends for Frontends* Patterns von Newman. Es kann dann auch einen Vergleich zu den Alternativen in dieser Arbeit gezogen werden.

Eine Weiterführung der Beispielapplikation von dem Prototyp, der in dieser Arbeit erstellt wurde, in eine umfangreichere Version, die die Alternativen genauer einführt.

[10] https://gatling.io/

[11] https://www.thoughtworks.com/continuous-integration

# Abkürzungen

**AAA** Authentication, Authorization, Accounting

**API** Application Programming Interface

**CI** Continuous Integration

**CIA** Confidentiality, Integrity and Availability

**CORS** Cross-Origin Resource Sharing

**CRUD** Create, Read, Update and Delete

**CSS** Cascading Style Sheets

**DBMS** Database Management System

**DTO** Data Transfer Objects

**ESI** edge side includes

**HTML** Hypertext Markup Language

**REST** Representational State Transfer

**SOA** Service Oriented Archtitecture

**SSI** server side includes

**TLS** transport layer security

**UI** User Interface

# Abbildungsverzeichnis


1.1 Eine traditionelle Service Architektur . . . . . . . . . . . 4
1.2 Eine Mikrofrontend Architektur . . . . . . . . . . . . . . . 5

2.1 Portal Komposition . . . . . . . . . . . . . . . . . . . . . 20
2.2 Interaktion bei Portal Komposition . . . . . . . . . . . . 22
2.3 PortalCompositionController.java, die Funktion zum Komponieren von Seiten . . . . . . . . . . . . . . . . . . . . 23
2.4 Portal.html, das Page Template in welches andere Services integriert werden . . . . . . . . . . . . . . . . . . . . 23
2.5 Backend Komposition . . . . . . . . . . . . . . . . . . . . 25
2.6 Interaktion bei Backend Komposition . . . . . . . . . . . 26
2.7 BackendCompositionController.java . . . . . . . . . . . . 26
2.8 Frontend Komposition . . . . . . . . . . . . . . . . . . . . 29
2.9 Interaktion bei Frontend Komposition . . . . . . . . . . . 30
2.10 PortalCompositionController.java, die Funktion zum Komponieren von Seiten . . . . . . . . . . . . . . . . . . . . 30

3.1 caniuse Abfrage für WebComponents . . . . . . . . . . . 46

# Tabellenverzeichnis

1.1 Zusammenfassung Kriterien für Migrationsarbeit . . . . 9
1.2 Kriterien für Aktualisierbar in Produktion . . . . . . . . . 11
1.3 Kriterien für Unabhängigkeit der Services . . . . . . . . . 12
1.4 Kriterien für Effekt auf die Geschwindigkeit . . . . . . . 14
1.5 Kriterien für theoretische Skalierbarkeit . . . . . . . . . . 15
1.6 Kriterien für Informationssicherheit . . . . . . . . . . . . 16

3.1 Abhängigkeiten, die vorherig, separat installiert werden müssen . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

# Literatur


[Auc18] Chase Aucoin. *The Microservice Manifesto*. 2018. URL: http://microservicemanifesto.com/.

[Coc16] A. Cockcroft. *The Evolution of Microservices*. Englisch. presentation at 2016 ACM Lerning Webinar. ACM. 2016. URL: learning.acm.org/webinar_pdfs/EvolutionOfMicroservices_WebinarSlides.pdf.

[com18] ISO 25010 committee. *ISO 25010*. 2018. URL: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?limit=3&limitstart=0.

[Cur05] Edward Curry. *Message-Oriented Middleware*. John Wiley und Sons, Ltd, 2005. 1-28. ISBN: 978-0-470-86208-7. DOI: 10.1002/0470862084.ch1. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/0470862084.ch1.

[Fac18] Facebook. *State and Lifecycle*. 2018. URL: http://www.reactjs.org/docs/state-and-lifecycle.html.

[Fan12] Charles Fan. *CRAP and CRUD: From Database to Datacloud*. 2012. URL: https://blog.dellemc.com/en-us/crap-and-crud-from-database-to-datacloud/.

[Fow03] Martin Fowler. *P of EAA: Data Transfer Objects*. 2003. URL: https://martinfowler.com/eaaCatalog/dataTransferObject.html.

[Goo18] Google. *Polymer Project*. 2018. URL: https://www.polymer-project.org/.

[Hoh15] Gregor Hohpe. *Enterprise integration patterns : designing, building and deploying messaging solutions*. eng. 19th print. The Addison-Wesley signature series. Boston: Addison-Wesley, 2015. ISBN: 978-0-321-20068-6.

[HT08] Andrew Hunt und David Thomas. *The pragmatic programmer : from journeyman to master*. [22nd printing]. Boston: Addison-Wesley, 2008. ISBN: 0-201-61622-X.

[IET18] IETF. *RFC8446*. Dez. 2018. URL: https://datatracker.ietf.org/doc/rfc8446/.

[Jam18] Pooyan Jamshidi. „Microservices: The journey so far and challenges ahead“. In: *IEEE Software* 35.3 (2018), S. 24–35. ISSN: 0740-7459. DOI: 10.1109/MS.2018.2141039.

[Lis18] Juan Liska. *Issue #1773 - Spring 5 support*. 2018. URL: https://github.com/springfox/springfox/issues/1773.

[MDN18] MDN. *<iframe>: The Inline Frame element*. 2018. URL: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe.

[Mic17] Microsoft. *ServiceModel Metadata Utility Tool (Svcutil.exe)*. 30. März 2017. URL: https://docs.microsoft.com/en-us/dotnet/framework/wcf/servicemodel-metadata-utility-tool-svcutil-exe.

[Moz18a] Mozilla. *Cross-Origin Resource Sharing (CORS) - HTTP | MDN*. 2018. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

[Moz18b] Mozilla. *Web Components*. 2018. URL: https://developer.mozilla.org/en-US/docs/Web/Web_Components.

[New15] Sam Newman. *Backends for Frontends*. Nov. 2015. URL: https://samnewman.io/patterns/architectural/bff/.

[Pau+17] C. Pautasso u. a. „Microservices in Practice, Part 1: Reality Check and Service Design“. In: *IEEE Software* 34.1 (Jan. 2017), S. 91–98. ISSN: 0740-7459. DOI: 10.1109/MS.2017.24.

[Ric18a] Chris Richardson. *API gateway pattern*. 2018. URL: https://microservices.io/patterns/apigateway.html.

[Ric18b] Chris Richardson. *Backend for Frontend pattern*. 2018. URL: https://microservices.io/patterns/apigateway.html.

[Ric18c] Chris Richardson. *Client-side ui composition*. 2018. URL: https://microservices.io/patterns/ui/client-side-ui-composition.html.

[Ric18d] Chris Richardson. *Decompose by business capability*. 2018. URL: https://microservices.io/patterns/decomposition/decompose-by-business-capability.html (besucht am 10. 01. 2019).

[Ric18e] Chris Richardson. *Microservices Patterns*. manning publications, 2018. 477 S. ISBN: 978-1-61729-454-9. URL: https://www.manning.com/books/microservices-patterns (besucht am 01. 10. 2018).

[Ric18f] Chris Richardson. *Server-side fragment composition*. 2018. URL: https://microservices.io/patterns/ui/server-side-page-fragment-composition.html.

[Ric18g] Chris Richardson. *What are microservices?* 2018. URL: https://microservices.io/.

[San01] Hawke Sandro. *REST - Semantic Web Standards*. 2001. URL: https://www.w3.org/2001/sw/wiki/REST (besucht am 10. 01. 2019).

[Söd17] Tom Söderlund. *Micro frontends—a microservice approach to front-end web development*. 2017. URL: https://medium.com/@tomsoderlund/micro-frontends-a-microservice-approach-to-front-end-web-development-f325ebdadc16.

[Spr12] Springfox. *Springfox: Automated JSON API documentation for API's built with Spring*. 2012. URL: https://github.com/springfox/springfox.

[Thy18] Thymeleaf. *Tutorial: Using Thymeleaf*. Englisch. Okt. 2018. URL: https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#template-resolvers.

[W3C01] W3C. *ESI Language Specification 1.0*. ESI Language Specification 1.0. Aug. 2001. URL: https://www.w3.org/TR/esi-lang (besucht am 10. 01. 2019).

[W3C17] W3C. *HTML 5.2*. HTML 5.2. 14. Dez. 2017. URL: https://www.w3.org/TR/html/ (besucht am 10. 01. 2019).

[W3C18a] W3C. *CSS Current Status - W3C*. Englisch. Nov. 2018. URL: https://www.w3.org/standards/techs/css#w3c_all.

[W3C18b] W3C. *HTML Imports*. 23. Okt. 2018. URL: https://w3c.github.io/webcomponents/spec/imports/.

[Wir11] Rebecca Wirfs-Brock. *Introducing Landing Zones*. Englisch. Juli 2011. URL: http://wirfs-brock.com/blog/2011/07/20/introducing-landing-zones/.

[Wis14] Simon Wistow. *Using ESI, Part 1: Simple Edge-Side Include*. 27. Aug. 2014. URL: https://www.fastly.com/blog/using-esi-part-1-simple-edge-side-include (besucht am 10. 01. 2019).