Vrije Universiteit Amsterdam Universiteit van Amsterdam

Master Thesis

Microfrontends: Bringing microservices to the Web frontend

Author: Ali Ameer (2516247)

1st supervisor: Dr. Ivano Malavolta
2nd reader: Prof. dr. Patricia Lago

A thesis submitted in fulfillment of the requirements for
the joint UvA-VU Master of Science degree in Computer Science

August 1, 2020

"Simplicity is prerequisite for reliability" – Edsger W. Dijkstra

Handwritten annotation from *How do we tell truths that might hurt?*

Abstract

Although the MicroServices Architecture (MSA) is mostly applied to backend services, *microfrontends* have recently appeared as a way to bring much of the approaches and advantages of MSA to the frontend world. This thesis involves the compilation of a catalogue of microfrontend implementation patterns by means of conducting a Grey Literature Review (GLR). The patterns are then analyzed informally to see the effect they have on certain Quality Attributes (QA) that are deemed important in the context of microservices and microfrontends. Finally, the frontend of a benchmark microservices project is implemented using most (7 out of 9) microfrontend patterns. The GLR revealed that *Independence*, and *Scalability* are the most discussed QAs in the context of microservices and microfrontends, where *independence* is the key QA that enables other QAs too. The analysis revealed that only two microfrontend patterns enable complete independence, whereas other patterns enable a limited independence because of the runtime context (Web browser) of microfrontends. The scalability enabled by these patterns was not analyzed because *computational* scalability for microfrontends is determined by the Web browser and it is the same irrespective of which microfrontend pattern is used. The results of this thesis are useful for practitioners as well as researchers; practitioners can consult the pattern catalogue as well as the reference microfrontends application to choose the pattern most suited to their task at hand. Researchers can also make use of the pattern catalogue as well as the reference application to perform quantitative analyses and further explore the emerging topic of microfrontends.

To my parents who have always provided me with never-ending love, support, and motivation.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Ivano Malavolta, for encouraging me to pursue this topic, and for his continued support and encouragement throughout the project. Then, I would like to thank Prof. dr. Patricia Lago for agreeing to be the second reader of the thesis.

Finally, I would also like to thank my friends and family (those in Pakistan, those in the Netherlands) for their perpetual love and support.

Contents

List of Figures

List of Tables

LIST OF TABLES

List of source codes

Abbreviations

AJAX Asynchronous Javascript and XML. vii, 1, 19, 20, 59–61, 77

AMD Asynchronous Module Definition. 21

API Application Programming Interface. xi, 8, 13, 18, 62–69, 72, 84

ATAM Architecture Tradeoff Analysis Method. 8, 32, 86

BFF Backend-For-Frontend. 34

CDN Content Delivery Network. 54, 63, 64, 78

CERN The European Organization for Nuclear Research. 16, 17

CMS Content Management System. 9

CSI Client-Side Includes. iv, ix, xi, 40, 59, 61, 62, 74

CSS Cascading Style Sheets. xi, 1, 2, 16–20, 23, 48, 61, 79

DOM Document Object Model. vii, 18–20, 59, 61, 65, 69, 79

ES6 ECMAScript 6. 21

ESB Enterprise Service Bus. 15

ESI Edge-Side Includes. iv, vii, ix, xi, 8, 51–55, 74

GLR Grey Literature Review. iii, vii, 3, 25–27, 31, 32, 34–36, 81, 83–85, 87

HTML HyperText Markup Language. vii, xi, 16–20, 23, 40, 47, 48, 51, 54, 55, 57, 61, 62, 72, 73, 77, 79

Abbreviations

HTTP Hypertext Transfer Protocol. 1, 13, 15, 16, 18, 19, 21, 45, 50, 51, 53, 54, 57, 64, 70, 71, 75, 76

ISO International Organization for Standardization. 8

JS JavaScript. xi, 1, 2, 16–21, 23, 34, 48, 49, 55, 57–59, 61, 63, 65, 66, 73, 77, 79, 83

MLR Multivocal Literature Review . 26

MSA MicroServices Architecture. iii, 1–3, 5, 8, 10, 13–15, 26, 27

MVC Model View Controller. 20

MVP Model View Presenter. 20

MVVM Model–View–ViewModel. 20

npm Node Package Manager. 57, 58

PICO Population-Intervention-Comparison-Outcome. 27

QA Quality Attributes. iii, 2, 3, 8, 9, 27, 31, 32, 34, 35, 37, 38, 40, 44, 77, 81, 83, 84, 86, 87

REST Representational State Transfer. 15

SAAM Software Architecture Analysis Method. 32, 86

SASS Syntactically Awesome Style Sheets. 20

SEO Search Engine Optimization. 46, 50, 54, 57, 61, 64, 72, 74, 76, 78

SLR Systematic Literature Review. 5, 25–27

SOA Service-Oriented Architecture. iii, 14, 15, 26

SOAP Simple Object Access Protocol. 15

SPA Single Page Application. vii, xi, 9, 20, 21, 36, 55, 62–65, 67, 68, 79, 84

SRP Single Responsibility Principle. 1

SSI Server-Side Includes. iv, ix, xi, 47, 50, 51

UI User Interface. 20, 45, 50, 54, 57, 61, 71, 76, 78–80

URI Uniform Resource Identifier. 16, 43, 44, 46, 62, 65

UX User Experience. 19, 42, 45, 61, 64, 68, 71, 79

WSDL Web Services Description Language. 15

WWW World Wide Web. 16

XML Extensible Markup Language. 18

1

Introduction

1.1 Context

Microservices is an architectural style that promotes developing applications as a suite of small services where each service performs a single function (based on the Single Responsibility Principle (SRP), each service is organized around a business capability, and the services communicate with each other using lightweight mechanisms such as Hypertext Transfer Protocol (HTTP). In addition, each of the service is independently developed, tested, deployed, and maintained. Microservices enable several advantages such as an increased flexibility, ease of maintenance, ease of scalability, and better resilience. Based on these advantages, many of the industrial giants such as Amazon (3), Netflix (3), Uber (4), Zalando (5), and SoundCloud (6) have now adopted the microservices architecture that has helped them tackle growing complexity and also allowed then to scale to an ever-increasing demand.

As the field of Software Architecture has progressed to produce architectural styles such as MicroServices Architecture (MSA) to cope with increasing complexity and scale, the field of Web frontend development has also advanced to produce better ways of handling growing complexity. Web pages have evolved past their original purpose of statically delivering information, to full-fledged applications that provide a much wider functionality and a much richer experience. The introduction of JavaScript (JS) in the Web browser as a means to enable behavior, the introduction of Cascading Style Sheets (CSS) as a means to handle styling of web page content, and the introduction of Asynchronous Javascript and XML (AJAX) to enable user interactions without reloading the web page, are some of the most important driving factors that have led to this evolution. To cope with the increasing complexity of applications on the frontend, several CSS, and JS libraries and frameworks have been introduced, with the most popular ones including Twitter Bootstrap[1], Semantic UI[2], Angular[3], React[4], and Vue.js[5] (7). The core enablement of these frameworks is modularity; they allow the frontend code to be better organized into modules, or into modular concepts such as *Components* and *Services*.

The MSA is increasingly gaining recognition in academia as well as gaining adoption in industry. There are several secondary studies that have indicated a growing number of publications about microservices each year, thereby signalling a rising interest of the academic community (8), (9). Similarly, there is an increasing number of blog posts, web pages, conferences, and video talks about microservices where practitioners describe their intentions, plans, and experiences of adopting MSA at their organization, thus providing evidence of a growing interest of the industry too.

Closely observing the trends in the *microservices* movement reveals that most of the academic research focuses on backend systems only. For example, architectural patterns proposed for microservices are geared towards backend systems, QA concerns that are raised, and the solutions proposed for addressing those concerns are also for backend systems only. While a few secondary studies on microservices in the recent past have indicated the frontend as a prospective area for the progression of microservices (8), (10), there have been very few academic publications related to microservices in the Web frontend. Within the industry though, the *microfrontend*[6] movement has gained momentum although the concept is still not widely adopted.

1.2 Objective

The goal of this thesis it to **(i)** review the grey literature and see what architectural patterns have been proposed in order to introduce microservices to the Web frontend; the architectural patterns will be compiled into a catalogue. Each pattern will also be **(ii)** analyzed by listing its aspects that positively/negatively affect certain QAs that have often been mentioned in grey literature pertaining to microfrontends. The analysis will also

[1]`https://getbootstrap.com/`
[2]`https://semantic-ui.com/`
[3]`https://angular.io/`
[4]`https://reactjs.org/`
[5]`https://vuejs.org/`
[6]A term commonly used in industry to describe frontends organized using ideas from microservices

help compare the patterns with each other. Finally, a monolithic frontend application will be implemented using the identified patterns; the implementation will contextualize the pattern catalogue as well the pattern comparison, and it will also serve as a **(iii)** reference for those interested in how microfrontends can be realized.

Both practitioners and researchers can benefit from the outcomes of this thesis. Practitioners can consult the pattern catalogue as well as the reference microfrontends application to choose the pattern most suited to their task at hand. Similarly, researchers can also make use of the pattern catalogue as well as the reference application to further explore the emerging topic of microfrontends. For example, researchers can compare and contrast microfrontend patterns with those for backend systems, or they can perform quantitative analyses such as measuring code-level metrics of the reference application.

1.3 Research Questions

To meet the objectives of this thesis, the following research questions are formulated:

RQ1: What architectural patterns have been proposed for introducing the MicroServices Architecture (MSA) to the Web frontend?

RQ2: How do the architectural patterns compare along different QAs?

To address *RQ1*, a GLR will be performed to extract data from the relevant grey literature such as blog posts, and web pages. The resulting data will then be compiled into a catalogue of architectural patterns. To address *RQ2*, a sample Web application's frontend will be implemented using most of the architectural patterns in the catalogue, and the effect of the patterns on certain QAs will be analyzed; based on the analysis, the patterns will be compared and contrasted.

The remainder of this document is organized as follows: chapter 2 presents a summary of the progression of industrial and academic work on microservices over the past few years, chapter 3 presents the background about microservices, as well as about the evolution of Web frontend technologies, chapter 4 presents the method adopted to address the research questions, chapter 5 reports the results of conducting the study, chapter 6 raises points of discussion and of possible future work, and finally chapter 7 concludes the thesis.

2

Related Work

2.1 Microservices

The MSA has its roots in the industry, and it is increasingly being adopted to cope with the growing complexity and scalability demands faced by organizations. While practitioners continue to delve deeper into the subject, academia has been slow to catch up and several secondary studies on microservices regard the research into the subject to be at an early/immature stage (11), (9), (8), (12). This observation is reflected by the prevalent types of publications related to microservices. In their Systematic Literature Review (SLR), Vural et al. (13) report a higher number of *Solution Proposals* followed by *Validation Researches*, and *Evaluation Researches* whereas they also report a lack of *Experience Reports*, *Opinion Papers*; Di Francesco et al. (8), and Alshuqayran et al. (11), also report comparable findings. Similarly, Pahl & Jamshidi (9) also observe that most of the literature on microservices contains sample implementations and controlled experiments rather than detailed experience reports and proofs, and that those experiments are usually based on specific use cases rather than generic large-scale evaluations.

This gap between practice and academia can be explained by the fact that **(i)** microservices have stemmed out of industry, and that **(ii)** they have been introduced relatively recently, as compared to other more mature disciplines such as *Software Testing* for example. Despite the fact that research into microservices is at an early stage, there is an increasing interest in the subject, and almost all researchers report a growing number of publications each year, and predict this trend to continue (13), (8), (9).

Various secondary studies on microservices have identified different areas of focus of microservices research and also highlighted areas that require more attention for future research, as well as industry-led efforts. Table 2.1 summarizes those findings which are elaborated later in the following subsections. Please note that the *Period* column in the table shows the period that was considered by the authors of the secondary study.

Paper	**Period**	**Focused areas**	**Require attention**
Vural et al. (13)	till January, 2017	Functionality, Performance, Test techniques	Monitoring Deployment Scalability Security Cost Resilience
Alshuqayran et al. (11)	2014-2016	Scalability Reusability Performance Agility Maintainability	Security, Load distribution DevOps Tracing
Pahl & Jamshidi (9)	2014-Nov, 2015	n/a	n/a
Di Francesco et al. (8)	till May, 2017	Performance Scalability Maintainability Functional suitability	Security, Testability Benchmarking

Table 2.1: Areas of focus of microservices research

2.1.1 Areas of focus

Implementing the existing *functionality* of a monolithic application using the new microservices architecture, or put differently, migrating the monolith to the new architecture has naturally been one of the earliest areas of research. In addition to functionality, *performance* of microservices has also received attention from the research community. Performance of a microservices-based application is inevitably adversely affected because of the additional network latency which is not the case with monolithic applications. The remaining research has mostly been about the validation of benefits/advantages of microservices, such as increased scalability, agility, and maintainability.

2.1.2 Areas requiring attention

Comparing the current state of microservices research and practice reveals that several topics that had been previously indicated as requiring more attention, have recently received some attention by industry as well as researchers. For example, microservices benchmarking has recently received some focus with Aderaldo et al. (14) highlighting the requirements for a suite of microservices that can be used to perform repeatable research; they then compare some of the open-source benchmark suites against these requirements. Gan et al. (15) contribute a new open-source benchmark suite that is representative of large end-to-end systems, and the suite fulfills some of the requirements mentioned by Aderaldo et al. Similarly, there have been industry-led advancements into the monitoring, and tracing through microservices. The OpenTracing[1] project is one example which provides a specification that allows tracing requests as they travel through a distributed system. Jaeger[2] is an instance of a tool that implements this specification and it has seen adoption by Netflix, where it was invented (16), as well as by other organizations after it was open-sourced.

Several of the general infrastructure-level services enabled by cloud providers have also been effectively used to address microservices-related concerns. For example, computational *scalability* of microservices has been greatly eased as cloud providers automatically scale (up or down) the infrastructure resources in response to a change in load. In addition, cloud platforms provide estimates of computational *costs* that the infrastructure incurs, which can eventually be used to compute estimates of the costs incurred by the microservices that run atop the infrastructure. In addition to the services enabled by cloud providers, several open-source initiatives such as Apache Mesos[3], and Kubernetes[4] also need to be highlighted as they have enabled the orchestration of containers (one of the key enablers of microservices (9)). This has eased the *deployment*, and some of the *DevOps* concerns of microservices. Finally, Netflix really has been at the forefront of using microservices, and of generating solutions to microservices concerns that they encounter. For example, Netflix's open-source library, Hystrix[5] can be used to enable fault tolerance and resilience in microservices.

[1]`https://opentracing.io/`
[2]`https://github.com/jaegertracing/jaeger`
[3]`http://mesos.apache.org/`
[4]`https://kubernetes.io/`
[5]`https://github.com/Netflix/Hystrix`

2.1.3 Progression of microservices into the Web frontend

While the MSA is primarily discussed in the context of backend systems, some academic works have indicated the possible progression of microservices into the frontend of systems. Jamshidi et al. identify the monolithic frontends as inhibiting the goals of microservices, and suggest that modular frontends and their association as part of the underlying microservice are required to help organizations fully reap the benefits of end-to-end microservices (10). Similarly, Di Francesco et al. suggest that growing usage of mobile software might lead to microservices-based patterns for frontends of systems in the future (8).

2.2 Microfrontends

Much like microservices, the microfrontends approach is born out of industry and it still has not attracted enough attention from the academic community. There is numerous grey literature related to microfrontends but most of the grey literature **(i)** considers just one or two microfrontends patterns at most, **(ii)** only makes claims about advantages/ QAs of microfrontends but does not substantiate those claims, and **(iii)** does not provide a reference implementation of an application in a number of different microfrontend patterns.

At the time of writing, there have been very few academic works related to microfrontends. The earliest is by Harms et al. (17) that analyzes the strategies available for organizing and implementing microservices frontends. They adopt a scenario-based analysis (Architecture Tradeoff Analysis Method (ATAM)) approach for analyzing the patterns and they analyze four patterns in total, including the monolithic application which is the baseline and also including the monolithic application with an Application Programming Interface (API) Gateway. This thesis differs from their work because **(i)** they do not sufficiently describe the sample application they use, nor is their application publicly available, **(ii)** they only compare two microfrontend patterns, namely, the Edge-Side Includes pattern and the Server-Side Microfrontend per View pattern, and **(iii)** they derive their QAs from the International Organization for Standardization (ISO) Standard (18) and those highlighted by Bass et al. (19) instead of using those highlighted in the grey literature on microfrontends.

A later academic work on microfrontends is by Yang et al. (20) and it uses Mooa[1] -

[1]`https://github.com/phodal/mooa`

a microfrontend framework - to implement a Content Management System (CMS). This thesis differs from their work because **(i)** they only *mention* three microfrontend patterns, namely, Server-Side Microfrontend per View, Client-Side Composition using Iframes, and the use of Web Components, **(ii)** they do not describe their use-case application (CMS) in detail, nor is their application publicly available, and **(iii)** their purpose is not to analyze different microfrontend patterns but to simply implement their use-case using the Mooa framework.

Prior to the finalization of this thesis, a few more academic works on microfrontends have been published which signals the increasing academic interest in the topic. The work by Pavlenko et al. (21) explores the concept of microfrontends by implementing a case study of an online courses aggregator using the microfrontends approach, and by listing the issues encountered meanwhile. Their method of implementing a sample project is similar to part of the method adopted during this thesis; however, this thesis differs from their work because **(i)** although they briefly refer to some techniques to implement microfrontends, they do not analyze those techniques in detail and later implement their sample application using a *single* microfrontends technique, **(ii)** their purpose is to generally explore the concept of microfrontends and list the issues they encounter while implementing the sample project, and **(iii)** their implementation is not publicly available.

The work by Darling et al. (22) is a patent that provides a detailed architecture-level description of how a web application can be implemented using microfrontends. This thesis differs from their work because **(i)** they describe SPA-based microfrontend patterns only, **(ii)** their emphasis is on only describing the patterns and the system in detail rather than on analyzing how the patterns affect certain QAs, and **(iii)** they do not provide a concrete implementation of the patterns they describe.

The study by Peltonen et al. (23) is a secondary study that explores the motivations for implementing microfrontends and the benefits as well as issues arising from implementing them. Some of their findings are comparable to the findings of this thesis. For example, they also highlight an increasing interest in microfrontends each year, and that the interest is mostly from the practitioner community. Moreover, they also conclude that the need to *scale* development teams is one of the main motivations behind implementing microfrontends. Additionally, some of the benefits and issues of microfrontends that they list are also discussed in this thesis. Despite the similar findings, their work differs from this thesis because **(i)** despite listing some microfrontend patterns in the early part of their paper, they do not describe the patterns in detail; neither do they analyze those patterns, **(ii)** they inspect the benefits and issues of microfrontends at a more general level, rather than at the per-pattern level.

2.3 Summary

The MSA is born out of industry and it is gaining a lot of traction therein. On the contrary, academic research about microservices has been slow to catch up; however, there is an increasing interest among the academic community which is indicated by the growing number of microservices-related publications each year. Initial areas of focus of microservices research included migration, functionality, performance impact, and the validation of the claimed benefits of microservices. Other areas such as benchmarking, monitoring and tracing, fault tolerance and resilience, security, and cost estimation still require more attention from the research community. Current research related to microservices primarily focuses on backend systems; although a few secondary studies on microservices have predicted that MSA will eventually progress into the design and development of the frontend of systems too. Recently, two academic works on microfrontends have appeared but neither has investigated and analyzed the patterns that can be used to implement microfrontends.

3

Background

3.1 Monolithic Architecture

Applications developed using the *monolithic architecture* are composed as a single unit where all functionality is encapsulated into a single application. Code-level modularity in a monolithic application may be attained by composing the application out of several modules as deemed necessary; however, the application itself is deployed as a single executable. Dragoni et al. define a monolithic application as "... a software application whose modules cannot be executed independently" [p. 1](12).

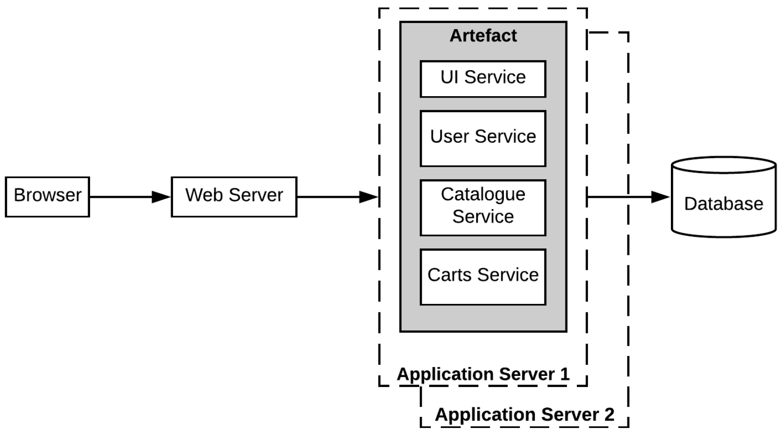

Figure 3.1: A sample monolithic application deployed as a single artefact. Adapted from (1)

.

Figure 3.1 shows a typical (sample) e-commerce Web application that comprises of different services. While the application has a modular structure where each of the services is separated, all the services in the application still need to be deployed together as a single artefact.

3.1.1 Problems posed by Monolithic Applications

Simpler monolithic applications are easier to develop, test, deploy, and scale; however, these advantages diminish as the monolithic applications grows beyond a certain size. After they do so, monolithic applications pose the following problems:

Independence Monolithic applications limit the independence afforded by the different teams in an organization. The different parts of the application *cannot* be completely independently designed, developed, tested, and deployed. Additionally, the monolith represents a technology lock-in, thus all parts of the application need to be implemented using the same technology stack. This makes it hard to choose the right tool for the right job as the technology choices made at earlier phases of application development constrain those at later stages.

Maintainability As simpler monolithic applications grow into complex ones, they accumulate technical debt and become harder to maintain. The inherent complexity cluttered into a single application makes it harder for a developer to understand all the different parts of the application in order to make a change, thereby negatively affecting the developer productivity. This in turn also negatively affects the *agility* of the organization as a whole.

Scalability Conceptually, it seems easier to scale monolithic applications by simply replicating multiple copies of the monolith and placing the copies behind a load balancer. At a deeper level, such replication is wasteful since not all components of the monolith will be equally exerted in response to an increased load. Consequently, monolithic applications negatively affect the *scalability* of the system as a whole.

Agility Monolithic applications negatively affect *agility* and speed of delivery because it becomes hard to isolate part of an application in order to change, test, and release it. Despite making a small change to just a part of the application, the whole monolith needs to be built, tested, and released which obviously takes longer. As far as deployment is concerned, although it seems easier to deploy a single monolithic application instead of separately deploying multiple small applications, deploying a monolith is more cumbersome because changes to the different parts of the application need to be coordinated before they can be deployed. One team may have to wait for another team before it can release a change to the monolith.

Reliability In some aspects, monolithic applications negatively affect reliability, whereas in other aspects they positively affect reliability. For example, a bug/issue in one part of a monolithic application inevitably affects other parts of the application thereby causing the whole application to dysfunction. It also becomes harder to debug an issue because it is harder to separate the fault and fix it. On the other hand, monolithic applications do not involve communication over the network as MSA-based applications do. Therefore, monolithic applications are not subject to unreliability caused due to communication failures.

3.2 Microservices Architecture

Owing to the shortcomings of the monolithic architecture, the MSA has recently emerged as a way to build applications that overcome those shortcomings and help organizations scale as complexity increases.

While there is not a precise definition for the microservices architecture, a commonly used definition is that by Lewis & Fowler where they define the MSA style as an "... approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies" (3).

Figure 3.2 shows the sample monolithic application from figure 3.1 split into the microservices architecture using one possible configuration of the services. As compared to the monolithic application, each service is now logically split; therefore, it can have its own choice of technology, its own source code repository, its own deployment pipelines, its own underlying infrastructure, and ideally its own database. All this separation allows independent development and deployment of the services, thereby helping overcome the challenges posed by monolithic applications.

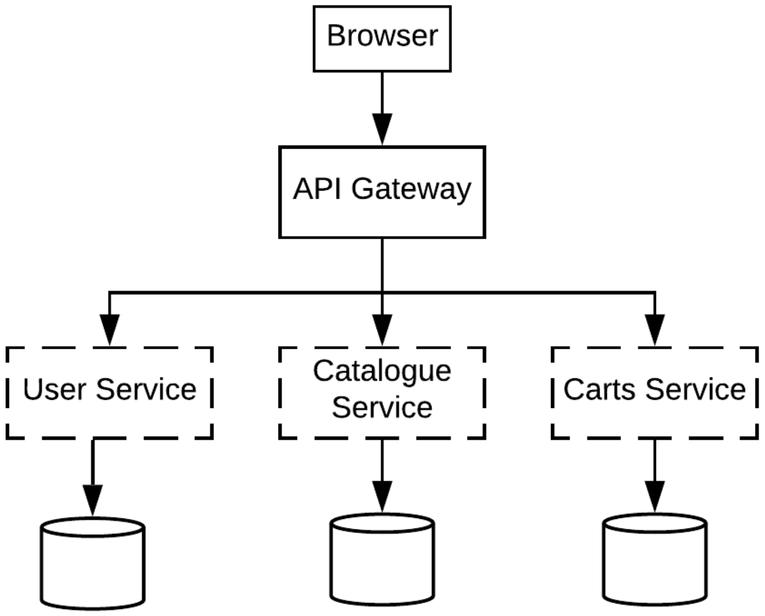

Figure 3.2: The sample monolithic application split into the microservices architecture

3.2.1 Emergence of the term "Microservice"

Observing that MSA is a fairly new topic, it is interesting to consider how the term was coined. While the detailed etymology of the term remains unknown, according to Lewis & Fowler the term *microservice* was first brought up during a workshop of Software Architects in May, 2011. Later in 2014, the same group agreed on the term *microservice*, to describe an approach to develop applications (3). According to Zimmerman, the term originates from agile developer communities and has often been appearing in blogs posts and online articles since 2014, which was the first time Lewis & Fowler mentioned the term (24). Despite its origins, the term is increasingly being used through industry and academia to refer to an architecture where an application is comprised of a number of small services that are independently developed and deployed.

3.2.2 Relation to Service-Oriented Architecture (SOA)

There are differing views within the community as to whether the microservices architecture is a continuation of the SOA or if it is a different architectural paradigm altogether.

SOA and microservices are dissimilar in a few aspects such as the integration mechanisms, and the mentality behind building the services. Services in SOA are known to be integrated using heavy mechanisms such as Enterprise Service Bus (ESB), and involve complex protocols such as Simple Object Access Protocol (SOAP) to communicate between the services, and Web Services Description Language (WSDL) to describe the interfaces of the services. On the other hand, microservices promote the use of lightweight integration mechanisms such as Representational State Transfer (REST) over Hypertext Transfer Protocol (HTTP); one microservice can communicate with another by simply making (direct) HTTP calls to the endpoints exposed by the other microservice. As for the mentality behind building the services, SOA promotes the idea of reuse where services should share-as-much-as-possible (25). On the contrary, microservices promote the share-as-little-as-possible mentality, where a microservice should be as small as possible so as to allow its easy replaceability, and even allow multiple versions of a microservice to coexist.

Despite the few differences, SOA and microservices share a lot in common. Both have the concept of a *Service* which is an isolated piece of functionality - usually corresponding to a business functionality - that has clear boundaries, and well-defined contracts. The goal behind having multiple services is to group related functionality into a service, where functions within a service have high cohesion whereas the services as a whole have low coupling among each other. These similar characteristics enable both SOA, and microservices to scale individual services by launching multiple instances of the services, to allow for distributed development by organizing teams around services, and to allow integration of heterogeneous systems because internally each service can be implemented using any technology as long as it implements the relevant interface.

According to Zimmerman, the differences between SOA and MSA do not concern the architectural style in itself, but they concern how the architectural style is concretely implemented (24). Based on the few differences and numerous similarities between SOA and microservices, it can be concluded that microservices are not an entirely new architectural approach and have infact stemmed from SOA. This view is shared by many practitioners as well as researchers. Newmann regards microservices as a particular way of doing SOA much like Scrum is a particular way of doing Agile Software Development (26). Similarly, microservices have often been referred to as "SOA done right" (24), and as "fine-grained SOA" (10).

3.3 Evolution of the Web frontend

As the development of *backend* web services has experienced progression and the emergence of new architectures and techniques, so too has the development of the Web *frontend*.

3.3.1 Background

The World Wide Web (WWW) was invented by Tim Berners-Lee in 1989, as a means to exchange information between scientists working at universities and institutes around the world, especially those working at The European Organization for Nuclear Research (CERN). The fundamental idea behind the Web was to combine the prevalent technologies behind computers, data networks, and hypertext into a global information system (27). The concept behind the Web was to allow hypertext documents to be linked together, and for those documents to be accessible using a Web browser. The Web has since followed a client-server architecture where a Web client (or browser) requests a resource that is provided by a Web server.

The three fundamental concepts that the Web (still) centers around are Uniform Resource Identifier (URI), HTML, and HTTP. The URI is a unique address assigned to each resource on the Web, and a client uses a URI to request the resource from the server. HTML is the markup language that is used to enclose the content contained in a web page. Finally, HTTP is the network protocol that is used to exchange the resources, and mediate communication between a Web client and a Web server.

Since its inception, the Web has evolved past its original purpose and has transformed into a global information system. The nature of documents and applications on the web has also greatly evolved; instead of just presenting static information, the web now provides a much richer interaction between a user and the web page. This richer interaction is enabled by an ever-increasing set of features that are supported by Web browsers. In its current form, the three fundamental technologies behind the Web's *frontend* include HTML, CSS, and JS.

3.3.2 HTML

HTML is the markup language that was invented by Tim Berners-Lee and it serves as the publishing language of the Web (28). The content for a web page is enclosed in tags such as paragraph (p), headings (h1-h6), lists (ol, ul), and hyperlink (a) etc. Each of these tags assigns a meaning to the enclosed content, and the tags as a whole assign a structure to the overall HTML web page that is viewed using a Web browser. The code listing 1 shows a sample HTML web page that contains some of the standard HTML tags.

```html
<html>
  <head>
    <title>Sample Document</title>
  </head>

  <body>
    <h1>An HTML Document</h1>
    <p>This is a <i>simple</i> document.
  </body>
</html>
```

Listing 1: Source code of a sample HTML document

3.3.3 Cascading Style Sheets

CSS is the styling language that was invented in 1994, by Håkon Wium Lie who also worked at CERN (29). The central idea behind CSS is to separate the presentation of an HTML document from its structure. In this sense, the styling information for elements (tags) in an HTML document should ideally be presented in a separate document that contains directions on how the Web browser should style the elements. Another principal behind the inception of CSS was to balance between the web page author styles and the styles of the end user (29). This idea still exists in how browsers apply default styling to the HTML elements, which is overridden by the styles defined in the web page author's style sheets, which in turn can be overridden by styles defined by the end user (30). The code listing 2 shows a sample CSS stylesheet that contains some CSS rules that can be applied to the HTML document presented in the previous section. Line 1 of the listing changes the color of all *h1* elements in the web page to red, whereas line 2 of the listing boldens the text of all *p* elements.

3.3.4 JavaScript

JavaScript is an interpreted programming language that was created by Brendan Eich in 1995, during his time at Netscape Communications which was one of the leading companies

```css
h1 { color: red; }
p { font-weight: bold; }
```

Listing 2: Source code of a sample CSS stylesheet

working on the development of Web browsers at the time (31). As CSS enables adding presentation and styling of web pages, JS enables adding *behavior* to web pages by making them programmable. Currently, all major Web browsers expose a JS-based API that allows Web frontend developers to enable interactivity on the web page. The code listing 3 shows how simple forms of interactivity can be enabled on a web page using JS. Line 1 of the listing obtains a reference to the first *h1* element on a web page whereas line 2 attaches an event handler function to the element, where the function prints some text to the console when the element is clicked.

```javascript
1   const heading = document.getElementsByTagName('h1')[0];
2   heading.onclick = () => console.log('Clicked');
```

Listing 3: Source code of a sample JS file

Document Object Model (DOM) Internally, browsers represent a web page as a hierarchical tree-like structure, namely, the DOM. The DOM is a general concept that can be used to represent HTML as well as XML documents; as for HTML documents, each of the HTML tags in a web page corresponds to a *node* of the DOM tree. Figure 3.3 shows the DOM tree of a sample HTML web page.

Web browsers enable programmatic access to the DOM tree, as well as to other cross-cutting concerns such as localStorage[1], history[2], location[3], HTTP requests[4] etc. All these features have allowed web pages to evolve from static documents that merely present information, to interactive applications that deliver a richer experience much like that of desktop applications. Aral Balkan presents a documents-to-applications continuum where he differentiates between *websites* which are content-centric documents, and *web applica-*

[1]https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage
[2]https://developer.mozilla.org/en-US/docs/Web/API/History_API
[3]https://developer.mozilla.org/en-US/docs/Web/API/Location
[4]https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

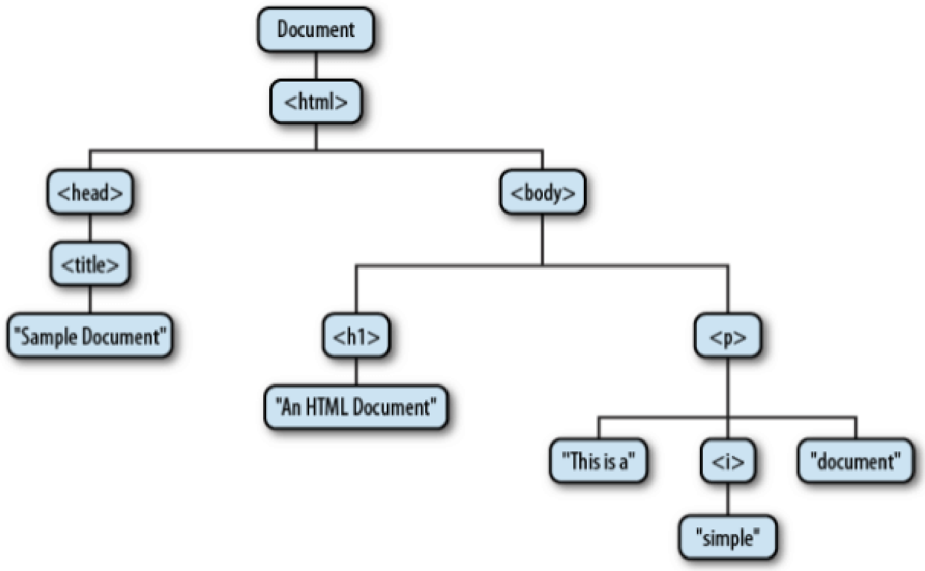

Figure 3.3: The DOM of a sample HTML page [p. 362](2)

tions which are behavior-centric, and argues that a web page does not have to be one or the other, but it may fall somewhere within the continuum (32).

3.3.5 Web frontend development

Whether a web page is geared more towards being a static document or towards an interactive application, Web frontend development is all about delivering an HTML web page, and its associated CSS, and JS to the end-users who view the web page in their Web browsers.

Web frontend development has experienced great progress in the past years. One watershed change in this was the introduction of XMLHttpRequest[1] or AJAX, which allowed JS code on a web page to make HTTP requests, without the need to reload the web page. Prior to the introduction of AJAX, data within a web page was submitted to the server using the *form* HTML element, which obviously required submitting the form and consequently reloading the web page. The introduction of AJAX enabled a better User Experience (UX) and eventually led to the writing of more complex frontend applications.

To deal with the increasing complexity of Web frontend applications, several build tools as well as libraries and frameworks started appearing. As for CSS, pre-processors such

[1]`https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest`

as Syntactically Awesome Style Sheets (SASS), and Less brought programmability to developing CSS. Features such as variables, functions, inheritance, modules, operators etc. allowed better manageability as the size of the CSS codebase grew. CSS libraries further enabled the reuse of styling code for commonly-used patterns of HTML components that had emerged. Examples of common HTML components include navigation bars, banners, and form elements etc. Twitter's Bootstrap remains one of the most popular CSS library with a few million downloads each week (33), and other notable examples of CSS libraries include SemanticUI[1] and Foundation[2].

As for JS, jQuery has been, and still remains one of the most-used Web frontend development libraries (7). It aids traversal and manipulation of the DOM, event-driven programming of the DOM nodes, as well as easier handling of AJAX requests. Despite its popularity, it lacked the means to suitably organize complex web applications which led to the appearance of frameworks such as Knockout.js[3], Backbone.js[4], and Angular.js[5]. These patterns introduced the Model–View–ViewModel (MVVM), Model View Presenter (MVP), and Model View Controller (MVC) application design patterns (respectively) to Web frontend development. The use of such application design patterns aided the separation of two central concerns of Web frontend development: data, and UI. With jQuery, *data* has to be manually inserted into the *UI* by performing direct DOM manipulations whereas the aforementioned frameworks abstract away the link between data and the UI by providing high-level concepts such as Models(data) and Views(UI). Developers then organize their applications around Models and Views and the framework takes care of updating the UI in response to a change in data.

These frameworks mark the start of the SPA-era with numerous new ones appearing each year while React.js, Angular(JS), and Vue.js remain the most popular ones (7). Most of the SPA frameworks further extend the idea of Model-View separation and enable writing modular code by providing an opinionated structure and higher-level concepts such as *Components* and *Services*. A component can be seen as an isolated and independent part of the UI where each component has some data and view associated to it. This also allows reusing the same component multiple times throughout the UI. A service can be seen as a

[1]`https://semantic-ui.com/`
[2]`https://get.foundation/sites.html`
[3]`https://knockoutjs.com/`
[4]`https://backbonejs.org/`
[5]`https://angularjs.org/`

cross-cutting concern that can be reused throughout the application. A common example of a service would be an HTTP service that abstracts the making of HTTP requests. While Angular directly defines the concept of services, React.js and Vue.js do not directly define the concept of services and instead rely on the use of programming language-level modules and functions in place of services.

Historically, client-side JS has not supported the concept of modules whereas server-side JS (Node.js[1]) has supported the concept of modules from the start. To overcome this, several build tools such as Gulp.js and Webpack have appeared that have allowed frontend developers to organize their codebase around modules. A separate build step is then also required to concatenate all the JS source code and generate fewer final artifacts which can be referenced by a web page. Up until 2011, JS did not natively support modules which resulted in the emergence of community-driven module systems such as CommonJS and Asynchronous Module Definition (AMD). More recently though with the emergence of ECMAScript 6 (ES6)(34) in 2015, JS has started to natively support modules (35), and in response, Web browsers have also started supporting in-browser modules thereby enabling the writing of module-based code (36).

Although modern Web frontend development practices have enabled more modularity as enabled by libraries and (SPA) frameworks, the frontend applications still gravitate towards monoliths as the complexity and the size of the application grows. This in turn greatly limits the independence of individual teams working on the same monolithic application thereby restricting the scalability of the organization as a whole.

3.4 Microfrontends

Microfrontends have recently emerged as an approach to bring the concept of microservices to Web frontend development. The term *microfrontends* appeared in 2016 under the technology radar of ThoughtWorks[2], which is an IT consultancy firm (37). This organization has also played a key part in the propagation of the *microservices* concept, especially through two key figures: James Lewis and Martin Fowler. While the microservices movement has gained sufficient adoption for backend systems, the concept has not been actively adopted for frontend development. Over time, complex frontend applications have evolved

[1]`https://nodejs.org/en/`
[2]`https://www.thoughtworks.com`

into monoliths which limit the benefits enabled by the microservices architecture of the underlying backend applications.

The central concept behind microfrontends is that in addition to delivering the backend part of a feature, a team also delivers the fronted part of that feature. This allows the organization to reap the full benefits enabled by end-to-end microservices. In this sense, microfrontends aim to bring the same benefits to the web frontend as microservices bring to the web backend. Figure 3.4 shows how microfrontends are typically adopted in an organization. In addition to working on the backend (including the database), a team is also responsible for the frontend of the feature it is developing.

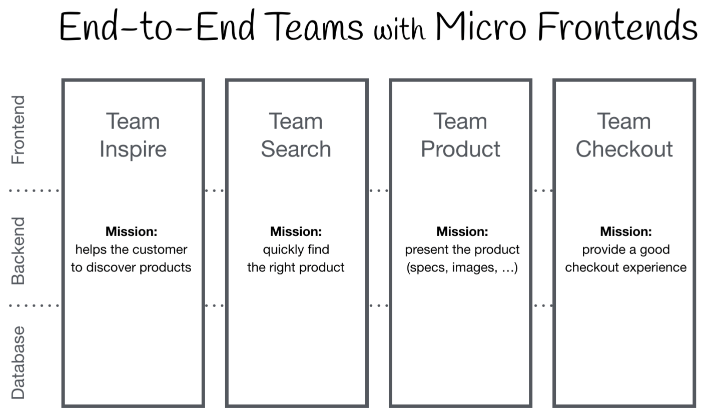
End-to-End Teams with Micro Frontends
Frontend
Backend
Database
Team Inspire
Mission: helps the customer to discover products
Team Search
Mission: quickly find the right product
Team Product
Mission: present the product (specs, images, ...)
Team Checkout
Mission: provide a good checkout experience

Figure 3.4: Vertical teams: a team is also responsible for the frontend of the feature it is developing (PR 1)

More details about the context around microfrontends as well as the problems they help solve can be read in an upcoming section: Root Pattern.

Similar to the concept of microservices, the concept of microfrontends has also been introduced by the industry. Several companies such as Zalando (38), DAZN (39), OpenTable (40), HelloFresh (41), Upwork (42), and IKEA (43) have adopted this architecture for their frontend applications. After adopting frontend microservices, HelloFresh reports advantages such as an increase in development velocity despite a steep initial learning curve, isolation and (possible) code freedom, faster error tracking and fixing, and easier development experience and maintainability of the application (41). Similarly, although DAZN does not explicitly report the gains of adopting microfrontends, it does report that its adoption of the microfrontends architecture stemmed out of a need for speed of delivery, scalability, and better maintenance of code quality (39).

3.5 Summary

The microservices architecture promotes splitting backend applications into a set of smaller services that are independently developed, tested, deployed and maintained. This independence allows microservices to overcome many of the shortcomings associated with monolithic applications thereby enabling organizations to scale by better dealing with the increasing complexity of backend applications. The Web's frontend has also become much more complex and has evolved from static documents to dynamic applications that afford much richer interactions with the users. These richer interactions are enabled by an ever-increasing set of features implemented by Web browsers; nevertheless, the Web's frontend still centers around three fundamental technologies: HTML, CSS, and JS. Several frontend frameworks, have emerged that have helped better handle the increasing complexity of the applications; however, as the applications grow beyond a certain size, they still gravitate towards monoliths bringing with them all the associated shortcomings. Recently, *microfrontends* have appeared as an approach that bring much of the ideas of microservices to Web frontend development.

4

Method

4.1 Implementation patterns for Microfrontends

To address RQ1, a Systematic Literature Review (SLR) was to be conducted. While SLRs may serve many purposes such as positioning new research based on existing research, identifying gaps in existing research, SLRs may also help summarize the existing findings of a research field to help further interpret and evaluate those findings (44). An initial search on *Google Scholar* with search terms such as "microfrontend(s)", "frontend microservices" revealed that there is a lack of academic publications related to this topic. On the other hand, using the same search terms on *Google Search* revealed that this topic is actively under discussion in various non-academic channels such as blog posts, web pages, videos etc. Consequently, given the lack of academic publications related to microservices in the Web frontend on the one hand, and the presence of numerous, relevant grey literature on the other hand, a Grey Literature Review (GLR) was performed in order to address RQ1.

Grey Literature can be considered (research) material that is produced outside of formal, academic publishing channels. SLRs typically only consider published academic literature as a means to address the research questions. Although, one of the most popular guidelines for performing SLRs in Software Engineering recommends consulting grey literature to ensure full coverage of the research, and to counter publication and systematic bias (44), grey literature is rarely included in systematic reviews. Since grey literature usually does not contain the scientific rigour required of academic publications, relying on grey literature is not as strong as relying on academic literature. However, not including grey literature inevitably affects the comprehensiveness of an SLR as it lacks the perspective/voice of the practitioner which is more often expressed in the form of grey literature. The inclusion of grey literature in a systematic review is of even more importance in a field such as Software Engineering which is practitioner-oriented and application-oriented, where the industry leads innovation and research in some areas. This is especially true for the area of microservices as the concept of SOA, and then microservices have emerged through the industry.

One reason for not including grey literature during systematic reviews may be that the guidelines for carrying out an SLR based on academic literature may not be entirely suited to carrying out a systematic review based on grey literature. This issue has recently gained attention with several researchers proposing guidelines specifically tailored to GLRs. Garousi et al. present guidelines for incorporating grey literature when conducting a literature review (45), whereas Rainer et al. propose heuristics that can be used to improve the search for grey literature (46). Following those guidelines and heuristics, several studies have undertaken GLRs and Multivocal Literature Review (MLR) to incorporate information from the industry into the information from academia (47), (48).

For this thesis, based on the suggested guidelines and heuristics, a review protocol was devised to carry out a GLR to help address RQ1. In principle, the idea is to follow a review protocol similar to the one that is followed while conducting an SLR and only modify the individual steps to cater for grey literature (45). The remaining part of this section describes the review protocol that was used to conduct the GLR.

4.1.1 Research questions

The research questions are central to any literature review, be it an SLR, MLR, or GLR. They drive the entire review process since the search keywords are derived from the research questions, and the data extraction and data synthesis phases are devised in a way to help answer the research questions. For this literature review, RQ1 served as the primary research question and it is repeated as follows,

P-RQ1: What architectural patterns have been proposed for introducing the MicroServices Architecture (MSA) to the Web frontend?

In addition to the primary research question, the secondary research questions are as follows,

S-RQ1: Which libraries/frameworks can be used to implement microservices in the Web frontend?

S-RQ2: Which QAs are enabled by implementing MSA in the Web frontend?

As recommended by common SLR guidelines, research questions ideally should be relevant to practitioners as well as to researchers. Keeping this in mind, the PICO criteria was applied to the research questions to frame them and table 4.1 shows the results of doing so.

Population	- Software Architects. - Software Developers. - Researchers interested in microservices and/or in microfrontends.
Intervention	Not applicable.
Comparison	Not applicable.
Outcome	- Catalogue of architectural patterns for implementing microservices in the frontend Web. - A comparison of different microfrontend patterns. - A reference microfrontends application.

Table 4.1: PICO criteria applied to the research questions

4.1.2 Search strategy

A systematic review protocol requires defining a search strategy as part of the protocol which includes defining *sources* of information that will be searched as well as the *search terms* that will be used to perform the search. The search strategy needs to be aligned in a way that the research questions are effectively addressed. For this study, *Google Search* was used as the text search engine as it is one of the most popular search engines and it is an often used and recommended source for performing GLRs (45), (46). Separate searches on blogs such as Medium.com[1], and InfoQ[2] were not deemed necessary since trial runs of the review protocol revealed that the Google Search results also contained results from the aforementioned blogs. In addition to the primary resources retrieved after filtering the resources based on inclusion/exclusion criteria, a single-level backward snowballing was also performed on the list of primary resources to include additional relevant resources.

To search for the relevant grey literature, keywords in the research question were identified. These keywords served as the queries that Google Search was queried for. Moreover, different synonyms and common variants of the keywords were also considered to ensure comprehensiveness of the search. Table 4.2 presents the query terms extracted for each of the keyword identified in the primary research question (P-RQ1).

Keyword	**Query**
architectural	architectural, architecture
pattern	pattern(s)
microservice	microservice, micro-service, micro service
web	web
frontend	frontend, front-end, front end, client-side, client side

Table 4.2: Query terms extracted for keywords

Note that the search query had to be iteratively refined based on the perceived quality, and coverage of the search results obtained during trial runs of the review protocol; figure 4.1 shows how the search query was evolved during the process. During the first iteration, the search query contained keywords from the original research question (P-RQ1) as well as some additional keywords such as "microfrontend", "ui composition", and "frontend microservices" which were added to the query after they were identified as often-discussed in the community. The first search query was not specific enough as it also returned many results related to backend microservices patterns rather than frontend microservices patterns. Consequently, the query was evolved to focus on "microfrontend" patterns only. The second and third queries subsequently proved too restrictive and they only returned

[1]`https://medium.com/`
[2]`https://www.infoq.com/`

fewer than 20 results. Therefore, the final query (seen inside a red border in figure 4.1) was intentionally kept general while still focusing on microfrontends only.

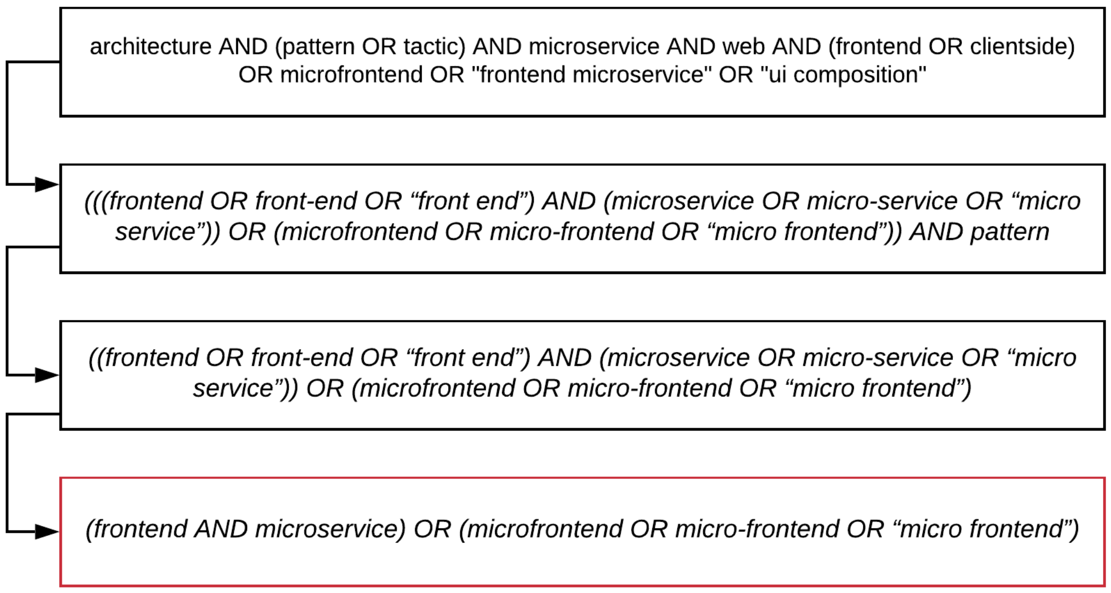
architecture AND (pattern OR tactic) AND microservice AND web AND (frontend OR clientside) OR microfrontend OR "frontend microservice" OR "ui composition"
(((frontend OR front-end OR “front end”) AND (microservice OR micro-service OR “micro service”)) OR (microfrontend OR micro-frontend OR “micro frontend”)) AND pattern
((frontend OR front-end OR “front end”) AND (microservice OR micro-service OR “micro service”)) OR (microfrontend OR micro-frontend OR “micro frontend”)
(frontend AND microservice) OR (microfrontend OR micro-frontend OR “micro frontend”)

Figure 4.1: Evolution of the search query after carrying trial runs of the review protocol

4.1.3 Inclusion/Exclusion criteria

The study selection criteria provides a means to select the most relevant resources from a pool of prospective primary resources. As the name entails, the inclusion/exclusion criteria is used to keep, or discard a resource as deemed relevant or not to the research questions. The following text shows the inclusion and the exclusion criteria along with a brief rationale where deemed necessary. Note that because the search query was intentionally kept general to include all the possible content related to microfrontends, the exclusion criteria was consequently more extensive to allow excluding resources that discuss microfrontend aspects other than architectural patterns.

Inclusion criteria

- **I1:** A resource focusing on microservices.
- **I2:** A resource focusing on the Web frontend.

- **I3:** A resource discussing at least one pattern to implement microfrontends.
- **I4:** A resource describing at least one pattern in sufficient detail, as opposed to merely referring to the pattern by name.

Exclusion criteria

- **E1** A resource discussing patterns for traditional Web frontend applications rather than microfrontend patterns.
- **E2** A resource discussing patterns for non-Web (e.g mobile) applications.
- **E3** A resource discussing patterns for backend applications.
- **E4** A resource that is *not* in English.
- **E5** A resource that is a duplicate; for example, a blog post on a platform by the same author and with effectively the same contents as a web page on the author's website can be considered duplicates.
- **E6** A resource reporting only the basic principles about microfrontends, rather than patterns of microfrontends; it is common to find blog posts that discuss microfrontends generally, but do not discuss how to implement them.
- **E7** A resource reporting only trivial implementations using microfrontends (e.g., introductory tutorials); it is common to find resources that briefly walk the reader through a specific example rather than discussing a higher-level implementation pattern.
- **E8** A resource written for promotional purposes; for example, a resource that contains information about microfrontends but eventually builds the article to refer to a paid service that the organization of the author is providing.
- **E9** White literature (i.e., academic papers, books); this thesis focuses on grey literature only; therefore, white literature needs to be excluded.
- **E10** Exclude the following resource types,
  - Videos and Webinars; these resources require great time and effort to analyze and they also usually lack the structure and development that a written resource has.

- Question-Answer based forums; these resources usually lack the structure and development that a written resource such as a blog post usually has.
- Github issues; same explanation as above.
- Tweets; same explanation as above.
- Github repositories (README); these resources usually only present the operational aspects of the source code and do not contain information about the architectural patterns.
- npm[1] pages; same explanation as above.
- Spam web pages; for example, web pages that do not have any meaningful and coherent content but they have included random keywords related to microfrontends just so they can attract page views.

4.1.4 Data extraction

After arriving at the final list of primary resources, the following fields were extracted from each resource,

- Year of publication; used to assess the overall trend in the field.
- Architectural patterns mentioned/discussed by the author; used to answer P-RQ1.
- Advantages/QAs mentioned by the author; used to answer S-RQ1.
- Frameworks mentioned by the author; used to answer S-RQ2.

4.1.5 Miscellaneous details

The actual search was performed on *January 15, 2020* using a private window of the Firefox Web Browser to ensure that the search results are as least personalized as possible and thus independently replicable and verifiable. Despite this, a search performed using the same search query from a different location, or on a different date may yield different search results because Google Search is known to customize search results based on several factors including the client's location and it is also known to update its index as well as its indexing algorithm periodically. To remedy this, the replication package[2] for the GLR is made publicly available and it contains the search results saved as web pages, a spreadsheet

[1] `https://www.npmjs.com/`

[2] `https://github.com/aliameer/microfrontend-patterns/tree/master/academic/replication-package`

with all the search results against the inclusion/exclusion criteria, and another spreadsheet with the final primary resources including the data extracted from those resources.

4.2 Comparison of Microfrontend patterns

To compare the patterns against each other and how they affect the different properties/ QAs, a scenario-based architecture analysis was *to be* adopted. Most formal scenario-based analysis techniques require the architecture of a software system to be described in great detail, and they are also a group activity that requires the involvement of different stakeholders so as to get a more complete view of the intended use of the software system. Scenario-based architectural analysis techniques proved unsuitable for analysing microfrontend patterns during this project because **(i)** after extracting data during the GLR, the final patterns were not described in great detail but were at a higher level, **(ii)** this thesis was conducted by a single individual and it was not possible to conduct the architecture analysis as a group-based activity as prescribed by formal scenario-based architecture techniques such as Software Architecture Analysis Method (SAAM), and Architecture Tradeoff Analysis Method (ATAM) for example.

Consequently, an informal analysis of the patterns was performed using the closed card sorting technique (49). Closed card sorting involves the sorting of content cards into predefined categories. For this thesis, the concrete QAs such as independence, maintainability etc. that were obtained as a result of RQ2 acted as the predefined categories whereas the different aspects of the patterns acted as the content cards. For each pattern, the following process was adopted.

- List each aspect of a pattern as a content card.
- Assign the content card to one or more of the predefined categories which are represented as QAs.
- Analyze whether the aspect has a positive or a negative effect on each QA.

For example, if a pattern required the presence of a *proxy server*, a content card with the label *proxy server* was created. Then, this card was assigned to the QAs maintainability, resilience, ease of migration, and performance. Finally, the effect of having a *proxy server* on each of the QAs was analyzed: it increases the effort required to *maintain* the software system, it may introduce a single point of failure in the system, it decreases the *ease* *of migration* to this pattern as it requires additional upfront setup of the proxy, and it decreases the *performance* of the pattern by adding to the network latency. Figure 4.2 shows a possible overview of the closed cart sorting technique for the example just explained.

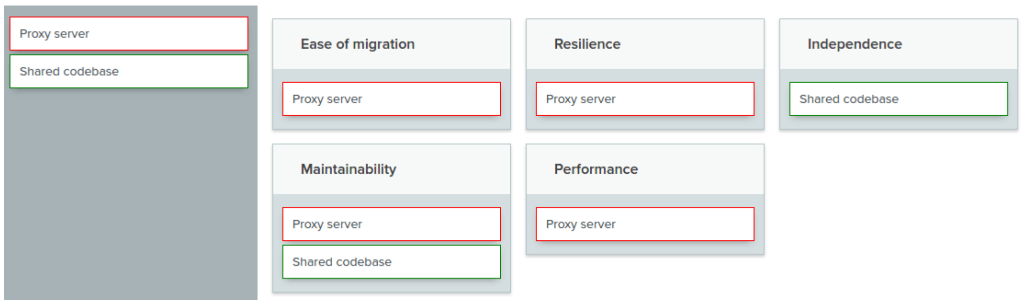

Figure 4.2: An example of closed card sorting for two sample aspects of a pattern

4.2.1 Reference microfrontends application

To assist the analysis, a sample microfrontends application would be useful in how it would act as a reference by providing context to the analysis. A search on GitHub for applications implementing microfrontends revealed that most of such applications are "toy" examples that **(i)** do not have an underlying backend application, **(ii)** are fairly simplistic and usually contain two microfrontends at most, and **(iii)** only implement a single microfrontends pattern. This can be explained by the fact that much like microservices, microfrontends are also usually implemented by medium-to-large-sized organizations. Such organizations do not always have an incentive to open-source their code; in fact, their code differentiates them from their competitors so they have a good reason *not* to open-source their code.

Considering this, a reference microservices application was identified and its frontend was implemented using *most* of the microfrontend patterns. The implementation aided the pattern analysis and it will also serve as a future reference[1] for practitioners interested in implementing microfrontends (50). The following projects were considered as possible candidates for implementing their frontends using the microfrontend patterns.

Acme Air (NodeJS) (51): A web application for a fictitious airline company, where all the services are implemented in NodeJS. The application includes an ExpressJS web

[1]`https://github.com/aliameer/microfrontend-patterns`

application which relies on a separate *Authentication* service. Although this microservices benchmark has been used in several academic works, it is fairly simplistic and only contains two primary services. Therefore, it was *not* selected.

Music Store (52): A web application for a fictitious music store, where all the services are implemented in C# .NET. The application includes services that are typical of an e-commerce application. All the services for this project are implemented using the same programming language; therefore, this project was *not* selected.

eShopOnContainers (53): A web application for a fictitious clothing store, where all the services are implemented in C# .NET. The application includes services that are typical of an e-commerce application. All the services for this project are implemented using the same programming language and this project was unstable (did not successfully compile) at the time of writing; therefore, this project was *not* selected.

SocksShop (54): A web application that simulates the user-facing part of an e-commerce website that sells socks. The application includes services that are written in different programming languages including JavaScript, Go, and Java. This project *was chosen* as it contains polyglot services, and also polyglot persistence backends as compared to other projects that have services written in the same programming language. Moreover, the project contains a Backend-For-Frontend (BFF) (26) that serves as a proxy between the frontend client and the individual services; the BFF is written in JavaScript, a programming language that the author of this thesis is better versed in.

4.3 Summary

Owing to the lack of academic literature on microfrontends, a GLR was carried out to compile a catalogue of implementation patterns for microfrontends. The GLR also helped identify the relevant QAs in the context of microfrontends, as well as the prevalent tools and technologies that can help implement microfrontends. After identifying the patterns, they were *qualitatively* compared against each other using an *informal* analysis that involves conjecturing the effect a pattern and its element have on a certain QA. The analysis was aided by implementing the frontend of a reference microservices-based application using most of the implementation patterns. This implementation will also serve as a future reference[1] for practitioners interested in implementing microfrontends (50).

[1] `https://github.com/aliameer/microfrontend-patterns`

5

Results

This chapter first presents the general results of carrying out the GLR, including the libraries and frameworks that can be used to implement microfrontends as well as the properties/QAs enabled by adopting microfrontends, as mentioned by the authors of the primary resources. Then, it presents a catalogue of implementation patterns for microfrontends that also contains the results of analyzing the patterns. Finally, it briefly describes the implementation of a reference microfrontends project.

5.1 GLR results

Figure 5.1 presents the number of resources at each stage of the GLR. Running the search query resulted in a total of *386* search results which were reduced to a mere *22* after applying the inclusion/exclusion criteria. This drastic reduction was expected because the search query was intentionally kept general so as to encompass all the grey literature related to microfrontends. As for snowballing, out of all the resources referenced by the primary resources, only a single resource passed the inclusion/exclusion criteria, and interestingly, this resource turned out to be a more detailed version of an existing resource in the list of primary resources. Therefore, effectively snowballing did not result in the addition of any new resources, which can be observed in figure 5.1 where the count of resources before and after snowballing remains *22*.

Figure 5.2 presents the number of resources published per year. It is hard to arrive at any conclusion based on the presented data, and especially given that 4 resources do *not* have an associated publication year as indicated in the figure by *n/a* (not applicable). Despite this, the year 2019 has clearly seen an increasing interest in microfrontends and

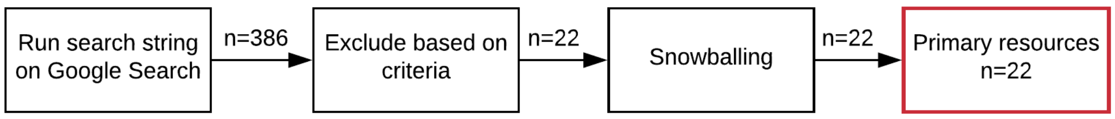

Figure 5.1: Count of resources at each stage of GLR

architectural patterns to implement them and it can be predicted that this trend will continue through 2020 as microfrontends are further explored. Note that as the search was carried out on January 15, 2020, there was just one primary resource from the year 2020; the results for the year 2020 are partial, and this is indicated in the figure by the hatched bar.

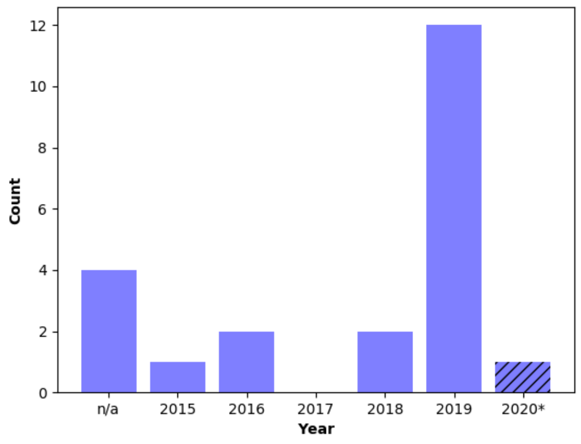

Figure 5.2: Count of resources per year of publication

5.1.1 Microfrontend libraries/frameworks

As for *S-RQ1*, table 5.1 presents the frameworks, and/or the libraries that are mentioned throughout the primary resources that could be used to implement microfrontends. Based on the counts, Single-SPA and Zalando's Tailor.js (which is part of their Project Mosaic) have been mentioned the most. The popularity of Single-SPA aligns well with the increasing popularity of SPAs in general. The popularity of Tailor.js may be attributed to the popularity and growth of Zalando itself, as well as to the fact that it was introduced earlier (in 2016 (P 2)) than some of the other frameworks such as frint.js (introduced in

2017 (55)). Please note that these libraries/frameworks are also mentioned later under the *Implementation* section of the relevant pattern in the catalogue.

Framework/library	**Count**	**Mentioned by**
Single-SPA	8	(P 3), (P 4), (P 5), (P 6), (P 7), (P 8), (P 9), (P 10)
Zalando Tailor.js	8	(P 6), (P 8), (P 11), (P 12) (P 1), (P 5), (P 13), (P 2)
frint.js	3	(P 7), (P 8), (P 9)
nodesi	2	(P 1), (P 14)
Compoxure	2	(P 1), (P 15)
Podium	2	(P 3), (P 8)
OpenComponents	2	(P 16), (P 12)
hinclude	2	(P 10), (P 14)
h-include	2	(P 10), (P 14)
Mooa, Ara, Luigi, Piral, include-fragment-element, html-include	once each	(P 4), (P 5), (P 8), (P 8), (P 14), (P 14) respectively

Table 5.1: Number of times different frameworks/libraries were mentioned

5.1.2 Quality Attributes

As for *S-RQ2*, table 5.2 presents a summary of the properties/QAs that the primary resources have mentioned in the context of microservices and microfrontends. The first column presents the QA whereas the *Context* column presents the alternate terms, or some brief context that further explains the QA. For example, *independence* has been mentioned in the context of independent testing, as well as in the context of tech autonomy afforded by a team. Similarly, *maintainability* has been mentioned in the context of *evolvability* as well as in the context of *manageability.* In general, it can be seen that *independence* is the most mentioned QA followed by *scalability.*

QA	**Context**	**Count**	**Mentioned by**
Independence	Development Deployment Testing Tech autonomy Flexibility Isolation	18	(P 1), (P 17), (P 3), (P 4), (P 5), (P 15), (P 6), (P 18), (P 7), (P 19), (P 20), (P 11), (P 9), (P 2), (P 16), (P 12), (P 10), (P 14)
Scalability	Scale frontend development Scale web application	9	(P 17), (P 5), (P 15), (P 21), (P 19), (P 8), (P 9), (P 10), (P 14)
Maintainability	Evolvability Modularity High cohesion Low coupling Easy to manage	8	(P 17), (P 3), (P 5), (P 7), (P 20), (P 9), (P 12), (P 14)
Velocity	Adapt to changes Faster development Agility	4	(P 7), (P 19), (P 9), (P 12)
Resilience	Reliability and Fault tolerance	3	(P 1), (P 6), (P 11)
Ease of migration	-	2	(P 17), (P 3)
Performance	Fast loading	1	(P 6)

Table 5.2: Quality attributes mentioned by each of the resources

Because of the nature of how grey literature is organized and how it differs from white literature, most of the time the QAs were *only mentioned*; the authors did not elaborate on how the microfrontends will help achieve the mentioned QA. Moreover, the meaning of some QAs was not properly explained all the times; for example, *scalability* was mentioned quite a few times but the authors did *not* always explain what it really means in the context of microfrontends. In the case of microservices, *scalability* has a computational aspect to it where due to the microservices architecture, a service is able to computationally scale in response to an increased load. In the case of microfrontends on the other hand, the computational aspect of scalability does not apply because all the frontend code runs in the same context, namely, inside the Web browser where the computational resources are the same irrespective of the architecture of the frontend code. Therefore, in the case of microfrontends, *scalability* refers to the *development aspect* where a team is able to scale in response to an increased number of features that need to be implemented.

5.2 Pattern Catalogue

The pattern catalogue is split into three parts: the first part describes patterns that are implemented primarily on the server-side, the second part describes patterns that are implemented primarily on the client-side, and the third part describes patterns that are implemented using a coordinated effort between the client and the server; hence, the third set of patterns is termed as *hybrid* patterns. Figure 5.3 shows an overview of all the patterns that are contained in the catalogue. In total there are 9 patterns, where 2 patterns (enclosed by dashed borders) are slight variants of other patterns.

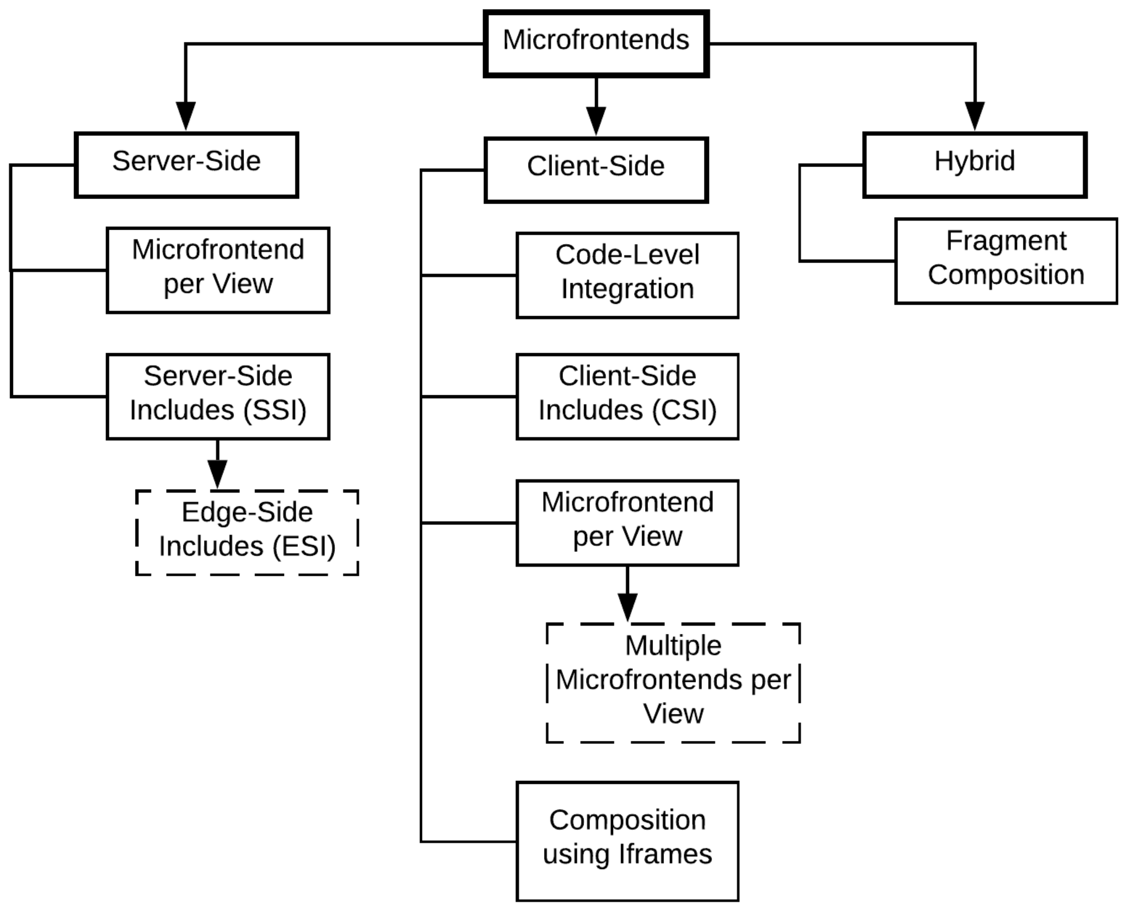

Figure 5.3: An overview of all the microfrontend patterns

Please note that no pattern assumes complete isolation between the client and the server; the patterns are classified as such (client-side or server-side) based on where bulk of the work related to microfrontends is performed. For example, the *Client-Side Includes* pattern is not completely oblivious to what the server returns; in fact, the server must return an HTML web page that contains certain HTML elements. On encountering these elements while parsing the web page returned by the server, the client then performs the bulk of the microfrontend-related actions such as fetching the resources referenced by the elements and inserting them at the appropriate place in the web page.

Also note that the patterns were analyzed for all QAs from table 5.2 except for scalability, and velocity. While all the other QAs are more profoundly affected by the architectural pattern used to implement microfrontends, scalability, and velocity are not as profoundly affected by the architectural pattern used; rather, they are more profoundly affected by other organizational aspects such as the development methodology (waterfall or agile), the organizational culture, and the development practices etc. For example, an organization that uses the waterfall development methodology is not able to adapt as quickly to market changes; therefore, its velocity is low. On the other hand, an organization that uses the agile development methodology is quickly able to adapt to market changes; therefore, its velocity is high. Moreover, the aforementioned QAs are more suitably analysed in a real-world setting of an organization building software, or at least in a group-based setting. Because this thesis was conducted by one individual, any conclusions made about the possible effects of microfrontend patterns on development scalability, and organizational velocity would have been speculative.

To describe the implementation patterns, a template will be used which has been adapted from the template used by Gamma et al. (56) and that used by Brown & Woolf (57). The final template includes the following fields,

- **Title:** Name of the pattern.
- **Context:** Situational factors that lead to a problem which the pattern helps solve.
- **Problem:** The actual problem that is solved by the pattern.
- **Solution:** A high-level overview of the pattern and the solution it enables.
- **Picture:** A graphical representation of the pattern and the different entities that interact in order to help realize the pattern.

- **Result/Analysis:** How does the pattern help solve the identified problems? What are the advantages, and disadvantages of using the pattern?
- **Implementation:** Details about how the pattern can be implemented; perhaps a reference to well-known technologies that can be used to implement the pattern.
- **Sample Code:** A brief snippet of code that shows how the pattern may be implemented using the relevant technologies.

In addition to a template for documenting patterns, Brown & Woolf (57) also describe a *Root Pattern* that is used to introduce the problem domain at a high-level which allows the reader of a pattern to establish a context to work within. This thesis also makes use of the Root Pattern to establish the need for microfrontends; subsequent patterns then use the Root Pattern as their base, and are more specific ways to move towards/ implement microfrontends.

Root Pattern: Microfrontends

Context: The complexity of a software product is increasing, and so are the scalability needs of the organization developing the product. The organization needs to introduce new features at a fast pace, and doing so requires expanding the development team. While the microservices architecture has been employed for the backend systems to cope with the increasing complexity and to allow for scalability, the frontend is still developed as a monolithic application. Each team delivers the backend part of a feature, but has to wait for the dedicated frontend team to implement the frontend part of that feature. Even if a team takes responsibility of implementing the frontend part of the feature on its own, the monolithic nature of the existing frontend application makes it hard for the team to implement the feature in isolation.

Problem: How to enable the advantages of using microservices for the *Web frontend* too?

- While developing a feature, a team wants to only focus on the feature it is responsible for without worrying about the features developed by other teams.
- A team wants to test the feature it has developed but instead of only testing that feature, the team also has to run code that tests *other* features developed by other teams, which takes longer for the tests to complete.

- A team wants to deploy an update to the feature it has developed but it has to wait before a feature developed by another team is deployed first.
- A team wants to use a new technology but it *cannot* due to the restrictions placed by earlier technology decisions that were made made during the development of other features.
- Trying to comprehend the code for one feature requires understanding other features too; thus making the application harder to maintain and manage.
- There is an error in one part of the application but the whole application is dysfunctional; moreover, it is hard to isolate the error and assign it to the specific team developing the feature where the error originates.
- The organization wants to launch a new feature in response to an opportunity in the market but doing so takes very long and the organization misses out on the opportunity.
- As a whole, the organization is *not* able to scale due to a lack of independence between teams.
- All features but *feature Z* are fast and performant, but the application hence the UX is limited by the least performant feature Z.

Solution: In addition to using the microservices architecture for the backend systems, the microserivces architecture should also be used for the frontend. A team should work on a feature in an end-to-end manner, including the frontend part for that feature.

Result/Analysis: As each team is responsible for the full feature including its frontend, the team is independent in terms of developing, testing, deploying, and maintaining the feature from start to finish. This also allows autonomy in terms of choosing the right technology for the task at hand. Moreover, smaller codebases lead to better understandability, hence to better maintainability of the software product as a whole. If the microfrontends are appropriately integrated with proper error boundaries, the resilience of the application also improves and an error in one part of the application does *not* cause other parts of the application to fail. Similarly, if certain microfrontend patterns are employed, the perceived performance of the application also improves. Finally, although the organizational (development) scalability, and the velocity of delivering software products are more profoundly affected by other factors - such as the organization culture, the development methodology, operational aspects etc. - than by the patterns used to develop the applications. Nevertheless, it can be conjectured that using microfrontend patterns improves development scalability as well as the velocity much like using microservices patterns does for backend systems.

Picture: Please refer back to figure 3.4.

Implementation, Sample Code, Related Patterns: Not applicable.

Server-Side Patterns

P1 - Server-Side Microfrontend per View

Context: Same as the Root Pattern context.

Problem: How to split a website/web application such that multiple teams in the organization working on different features can work *completely* independently?

Solution: Each team delivers its feature/microfrontend as a full web page or web application which is hosted standalone. Based on the client's request, a proxy server routes the request to the appropriate microfrontend, usually depending on some parameter in the request URI. The different microfrontends are then integrated together only using hyperlinks. Although the use of a proxy server is not required, it is often used to implement this pattern as it allows for some central bookkeeping in the system.

Picture: Figure 5.4 shows the interaction between the client and the server when this pattern is used. In this case, because the request path contains */mf-a*, the proxy routes the request to the *mf-a* microfrontend.

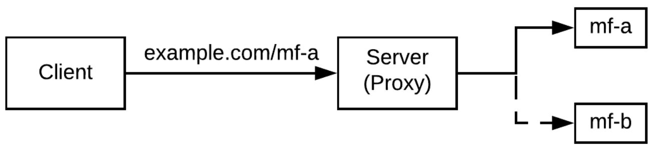

Figure 5.4: Proxy forwards the request to the relevant microfrontend

Result/Analysis:

In all the analysis tables that follow, the *Positive* column presents the aspects of the pattern that positively affect the mentioned QA/property and the *Negative* column presents the aspects of the pattern that negatively affect the QA/property.

QA/Property	Positive	Negative
Independence	As the microfrontends are "physically" separated, each team has complete independence in developing, testing, and deploying its application, as well as in choosing its tech stack. Needless is to say that the microfrontends have runtime independence too as they run inside separate web pages.	n/a
Maintainability	- When compared to a monolithic application, this pattern leads to smaller codebases for each microfrontend which results in better understandability, and eventually better maintainability of code. - There is very little to no coupling between the infrastructure or applications as they are connected using hyperlinks only. If the URI of a microfrontend changes, it can set up redirects on its own without requiring other microfrontends to change.	- More effort is required to maintain the separate infrastructure for each microfrontend. - More effort is required to maintain the proxy server.
Resilience	The microfrontends are completely standalone web pages/applications that get a separate runtime (hence separate *window*[1] object), error containment, as well as fault isolation.	The proxy may become a single point of failure.

[1]https://developer.mozilla.org/en-US/docs/Web/API/Window

Ease of migration	Does not *require* the use of any libraries/frameworks specifically designed to migrate to this pattern.	- In a monolithic application, each microfrontend may rely on functionality provided by other microfrontends. However, as a standalone web page/application, each microfrontend must be refactored towards self-containment. Therefore, migration to this pattern requires non-trivial refactoring of each microfrontend. - Migrating to this pattern requires first setting up a proxy server and then provisioning separate infrastructure for each microfrontend. Forwarding of HTTP headers from the proxy to the microfrontends may also need to be configured. - Ideally, there should not be any sharing between the microfrontends; therefore, each microfrontend must separately implement even the common web page elements such as navigation bars, banners, header, footer etc. Consequently, more effort is required to migrate to this pattern.
Performance	As compared to the monolith, the microfrontends comprise of comparatively smaller applications; therefore, the overall filesize of each client-side application is smaller.	- The proxy adds to the network latency. - As the microfrontends are connected to each other using hyperlinks, moving from one microfrontend to another requires a full web page reload; therefore, the UX (hence the perceived performance) deteriorates. - The UX may further deteriorate as a result of inconsistencies between the UI as each team may implement its own version of common web page elements in order to avoid coupling.

Miscellaneous	n/a	If each microfrontend is implemented as a sub-domain, the SEO presence of a website is split because search engines regard a subdomain as a separate entity; therefore, they would index and rank it separately (58). This splitting of rank produces a negative effect on the SEO presence of a website.

Table 5.3: Analysis of Server-Side Microfrontend per View pattern

Implementation: On a higher level, this pattern can be implemented using a mechanism capable of forwarding client requests to the relevant microfrontend; therefore, a web server acting as a reverse proxy is suitable for usage. Any popular web server such as Apache HTTP Server[1] or nginx[2] may be used to perform the forwarding of requests whereas the individual microfrontends may be implemented and deployed in any technology considered suitable.

Sample Code: The code listing 4 illustrates this for the case of an nginx proxy server which is configured to forward requests to the relevant microfrontend depending on the request *path*. As indicated by line 1, if the URI path of the request to the proxy matches */mf-a/*, the proxy forwards the request to the *mf-a* microfrontend, as indicated by line 2.

```
1 location /mf-a/ {
2   proxy_pass https://mf-a.com;
3 }
4
5 location /mf-b/ {
6   proxy_pass https://mf-b.com;
7 }
```

Listing 4: Partial configuration for sample nginx proxy that routes requests to microfrontends based on path

[1]https://httpd.apache.org/docs/2.4/
[2]https://docs.nginx.com/nginx/admin-guide/web-server/

P2 - Server-Side Includes (SSI)

Context: While the *Server-Side Microfrontend per View* pattern allows complete independence, it requires the overhead of a proxy server that is capable of forwarding requests to the right microfrontend, and it may also lead to inconsistencies between different microfrontends due to complete isolation. Also, for legacy monolithic applications with a lot of different features, it may not be possible to develop and deploy a separate web application for each feature; a more gradual move towards microfrontends is more desirable in such cases.

Problem: How to *compose* a web page/application out of different microfrontends on the *server-side*?

Solution: Instead of serving a complete web page on its own, the web server composes a web page out of different microfrontends. Each microfrontend resides locally on the Web server as a partial HTML fragment and it is included into the template of the web page at runtime.

Picture: Figure 5.5 shows a typical interaction between the client and the server when this pattern is employed. As a request for a web page arrives, the locally residing HTML fragments for *mf-a* and *mf-b* are included into the web page during runtime.

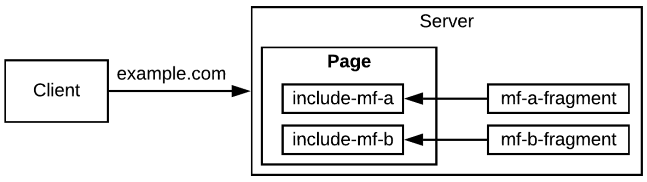

Figure 5.5: A web page is composed of fragments residing locally on the Web server

Result/Analysis:

QA/Property	Positive	Negative

Independence	- In theory, each team gets independence in how it can develop, test, and deploy its own HTML fragment as long as the different fragments are eventually integrated together and are made available at the Web server. - In practice however, each team has independence in only *deploying* changes to its microfrontend.	- The codebase is shared, and separation between microfrontends is enabled by establishing (naming) conventions for example by assigning team-based prefixes to HTML elements, JS functions, and CSS rules. - As the codebase is shared, the same deployment process must be followed by all the teams. - As the codebase is shared, the teams must use the same testing framework, and all tests must be run even when only a small change is deployed. - Deploying a change to one microfrontend causes the Web server (also containing other microfrontend templates) to be redeployed. - In theory, each microfrontend can include JS and CSS libraries/frameworks it wants to; however, in practice this is rarely done so, and the organization sets some technology constraints that the teams should follow. The base HTML file references some central JS and CSS libraries/frameworks, and the microfrontends are free to use them, but are discouraged to add any new libraries/frameworks on their own. - As the microfrontends run within the same web page, there is no runtime independence.

Maintainability	No extra infrastructure is required; therefore, there are no additional infrastructure-level maintainability concerns.	- Shared codebase makes it harder to keep a consistent view of the code. - As there is no isolation between a microfrontend and the web page that contains it, each microfrontend can make certain assumptions about what the web page provides. For example, if the web page is known to reference a certain JS library, a microfrontend can freely use that library. Although this is good for performance, it inevitably creates coupling between a web page and the microfrontends contained therein, which eventually makes it harder to maintain the overall application. - If a microfrontend is updated, the web server needs to be completely redeployed, thereby signalling the coupling between the microfrontends and the infrastructure.
Resilience	There are no extra infrastructure components; hence, there are fewer points of failure.	- As the microfrontends run within the same web page, it is possible for an issue/bug in one microfrontend to cause another microfrontend to dysfunction. For example, if one microfrontend erroneously changes a property/function on the *window* object that another microfrontend relies on, the latter microfrontend may dysfunction as a result. - Moreover, there is no error containment or fault isolation at the application level.

Ease of migration	- At the application level, only requires minimally refactoring the existing monolithic application according to the naming conventions established for each microfrontend and organizing it around partial views/templates. - This pattern does not require any extra infrastructure for the microfrontends, neither does it require any proxy server. Therefore, infrastructure-level concerns such as forwarding HTTP headers are also not applicable.	Depends on how this pattern is implemented, but if it is implemented at the Web server level, it would likely require the use of a server-side templating engine/package such as Twig, Jinja etc.
Performance	- Provides a load time performance comparable to that of the original monolithic application because the composition of microfrontend templates during runtime is the only additional step in this pattern. - As the microfrontends are composed into a single web page, the user need not switch between pages, and the UI also remains consistent. - This pattern does not affect the filesize of the client-side application.	n/a
Miscellaneous	As compared to a monolith, the SEO performance remains unaffected because the full web page/application is served after completely rendering it on the server-side.	n/a

Table 5.4: Analysis of Server-Side Includes (SSI) pattern

Implementation: On a higher level, this pattern can be implemented using transclusion mechanisms on the server-side. As a web server parses a web page and encounters certain tags that indicate the presence of microfrontends, the web server will fetch the microfrontends *locally* and insert them into the web page before returning a response to the client.

This pattern may be realized using the SSI feature, and specifically the *#include* directive supported by common Web servers such as Apache HTTP Server and nginx. Alternatively, this pattern may be implemented at the backend Web development framework level by using the *include* feature of common templating engines such as Jinja[1] for Python, and Twig[2] for PHP.

Sample Code: The code listing 5 presents an HTML document that includes different microfrontends into a web page using SSI. Both lines 4-5 will cause the web server to look locally for partial templates of microfrontends in the parent directory and insert the content of those templates into the HTML document before returning it to the web client.

```
1 <html>
2   ...
3   <body>
4     <!--#include virtual="../mf-a.html" -->
5     <!--#include virtual="../mf-b.html" -->
6   </body>
7 </html>
```

Listing 5: Source code of an HTML document that includes microfrontends using SSI

P3 - Edge-Side Includes (ESI) - Variant of P2

Context: The SSI pattern enables the implementation of microfrontends in the form of HTML fragments owned by separate teams. Although this allows limited independence between the fragments, this does require the fragments to be available locally on the Web server. Consequently, an update of one microfrontend therefore causes the redeployment of the Web server containing the other fragments too.

Problem: How to decouple the fragments from the parent Web server, when composing a web page on the server-side?

Solution: When composing a web page on the server, instead of fetching microfrontends locally, the microfrontends are fetched from a remote location possibly using HTTP. A web page comprises of microfrontends represented as ESI *include* tags, and as the server

[1]`https://jinja.palletsprojects.com/en/2.11.x/`
[2]`https://twig.symfony.com/`

processes these tags while parsing the web page, it requests each of the microfrontends from the location indicated in the *src* attribute of the ESI tag. Finally, after receiving a response from each microfrontend, the ESI-enabled server inserts the responses into the web page and returns a full-blown web page to the client.

Picture: Figure 5.6 shows a general overview of the solution explained in the previous text.

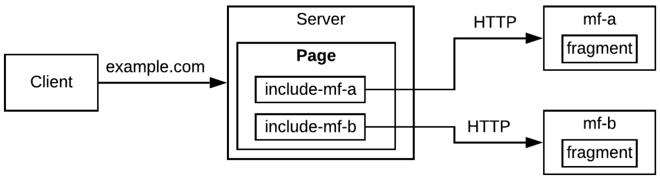

Figure 5.6: Microfrontends are fetched by the server after it parses the ESI tags in the web page

Result/Analysis:

QA/Property	**Positive**	**Negative**
Independence	As the codebases are not shared between the teams, each team has independence to develop, test, and deploy its microfrontend as long as it follows the established (naming) conventions, as well as technology constraints set by the organization.	As the microfrontends run within the same web page, there is no runtime independence.
Maintainability	- Smaller codebases for each microfrontend lead to better understandability, and hence better maintainability of code. - No extra proxy server is required. - There is very little to no coupling between the infrastructure or applications.	More effort is required to maintain the separate infrastructure for each microfrontend.

Resilience	There is infrastructure-level fault isolation, and error containment. For example, if one of the microfrontends fails while the server is waiting for a response from it, all the other microfrontends can continue to serve their responses.	- As compared to a monolithic application, the fact that each microfrontend is separately hosted and needs to be requested using HTTP introduces more points of possible failures. - As the microfrontends run within the same web page, it is possible for an issue/bug in one microfrontend to cause another microfrontend to dysfunction. For example, if one microfrontend erroneously changes a property/function on the *window* object that another microfrontend relies on, the latter microfrontend may dysfunction as a result. - Moreover, there is no error containment or fault isolation at the application level.
Ease of migration	At the application level, only requires refactoring the existing monolith application according to the naming conventions established for each microfrontend.	- Migrating to this pattern requires enabling ESI processing on the server and, it also requires provisioning separate infrastructure for each microfrontend. - Factors such as forwarding HTTP headers (including cookies) from the server to individual microfrontends need to be determined. - Depending on how this pattern is implemented, a server-side package/framework may be required to enable the processing of ESI tags.

Performance	- Caching web pages at the server after processing their ESI tags may help alleviate some of the performance concerns. - As the microfrontends are composed into a single web page, the user need not switch between pages, and the UI also remains consistent. - This pattern does not affect the filesize of the client-side application.	- Each microfrontend is fetched by making additional HTTP calls; therefore, the response time is naturally higher than that of a monolithic application. - The slowest microfrontend determines the response time of the server.
Miscellaneous	As compared to a monolith, the SEO performance remains unaffected because the full webpage/application is served after completely rendering it on the server-side.	n/a

Table 5.5: Analysis of Edge-Side Includes (ESI) pattern

Implementation: On a higher level, this pattern requires a server that is capable of processing ESI (include) tags. ESI as a technology was proposed in 2001 by Akamai technologies (59) and since has been implemented by CDNs such as Akamai, Fastly, and Cloudflare as well as by caches such as Varnish, and Squid; therefore, either of these can be used as an ESI-enabled server. Other technologies that can be used to implement this pattern include nodesi package for ExpressJS, and other frameworks such as Compoxure[1], Podium[2], Ara[3], and Zalando's Tailor.js[4].

Sample Code: The code listing 6 presents an HTML document that includes different microfrontends into a web page using ESI. Both lines 4-5 will cause the web server to request microfrontends from remote locations and insert the responses in place of the *<esi:include>* into the HTML document before returning it to the web client.

[1]`https://github.com/tes/compoxure`

[2]`https://podium-lib.io/`

[3]`https://ara-framework.github.io/website/`

[4]`https://github.com/zalando/tailor`

```html
<html>
  ...
  <body>
    <esi:include src="https://mf-a.com" />
    <esi:include src="https://mf-b.com" />
  </body>
</html>
```

Listing 6: Source code of an HTML document that includes microfrontends using ESI

Client-Side Patterns

P4 - Client-Side Code Level Integration

Context: While composition of different microfrontends at the server-side *only* is especially useful for legacy monoliths, there need to be patterns that allow building microfrontends on the client-side, especially given the increasing popularity and complexity of client-side SPAs. Moreover, composing the complete web page on the server-side *only*, negatively affects the perceived performance of the application as it takes longer for the client to receive an initial response.

Problem: How to enable the composition of microfrontends on the client-side during *build-time*?

Solution: Each team implements its feature (microfrontend) and distributes the implementation as a JS package. The packages are then integrated into a *main* container application using build-time tools.

Picture: Figure 5.7 shows a general overview of this pattern. The individual microfrontends are held as JS packages in a repository, and a container application imports these packages. After the container application is deployed, it is included by the web page.

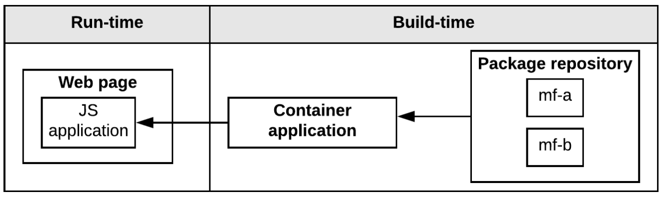

Figure 5.7: Microfrontends are integrated into the container application during build-time

Result/Analysis:

QA/Property	**Positives**	**Negatives**
Independence	As the codebases are not shared between the teams, each team has independence to develop, test, and deploy its microfrontend as long as it follows the established (naming) conventions, as well as technology constraints set by the organization.	As the microfrontends run within the same web page, there is no runtime independence.
Maintainability	Smaller codebases for each microfrontend lead to better understandability, and hence better maintainability of code.	- Although this pattern does not require each microfrontend to be separately hosted, it does require each microfrontend to be available in a package repository. Thereby, it also requires additional maintenance effort to ensure the proper availability of microfrontends in the package repository. - The container application is coupled to each microfrontend; the container application needs to be redeployed after even a single microfrontend is updated.

Resilience	As no extra servers are required, points of failure in the system are reduced. Moreover, package registries such as npm ensure their high availability.	- As the microfrontends run within the same web page, it is possible for an issue/bug in one microfrontend to cause another microfrontend to dysfunction. For example, if one microfrontend erroneously changes a property/function on the *window* object that another microfrontend relies on, the latter microfrontend may dysfunction as a result. - Moreover, there is no error containment or fault isolation at the application level.
Ease of migration	- At the application level, only requires refactoring the existing monolith application according to the naming conventions established for each microfrontend. - As no extra servers are required, concerns such as HTTP header forwarding applicable.	- Requires setting up a package repository for each microfrontend. - Requires the use of module bundlers, and of build-time tools.
Performance	- The Code Splitting[1] and Lazy Loading[2] features of modern module bundlers improve the load time, hence the perceived performance of web applications. - As the microfrontends are composed into a single web page, the user need not switch between pages, and the UI also remains consistent.	- If the container application uses a runtime module bundler to import the microfrontends, the size of the module bundler will add to the overall filesize of the frontend application code. - As part of the application will move from being embedded into the HTML page to being requested as external JS files, the overhead of additional network requests will increase.
Miscellaneous	n/a	The SEO performance of a web page whose content is included on the client-side is inferior to that of a monolithic application rendered on the server-side.

Table 5.6: Analysis of Client-Side Code Level Integration pattern

[1]`https://webpack.js.org/guides/code-splitting/`

[2]`https://webpack.js.org/guides/lazy-loading/`

Implementation: This pattern can be implemented using a JS package manager such as Node Package Manager (npm)[1]. Each team publishes its client-side JS code to an npm registry as a package. A parent application serves as the container of the microfrontends and it imports the packages and uses them as necessary. Finally, the JS code artefact for the container application is built using build-time tools such as Gulp[2], or Webpack[3]. This code artefact then contains the code for all the individual microfrontends too.

Sample Code: The code listing 7 shows the manifest file (package.json) of a container application that uses packages from two other microfrontends. Lines 6-7 declare a dependency on the microfrontends which will be installed after the command *npm install* is run for this manifest file.

```
1 {
2   "name": "@my-company/container-application",
3   "version": "1.0.0",
4   "description": "A microfrontends-based webapp",
5   "dependencies": {
6     "@my-company/mf-a": "^2.3.4",
7     "@my-company/mf-b": "^4.5.6",
8   }
9 }
```

Listing 7: Manifest file of container application used to implement Code-Level Integration pattern

The code listing 8 shows the source code of the container application which imports (line 1-2) the actual microfrontends that were declared as a dependency in the manifest file.

```
1 require('mf-a');
2 require('mf-b');
3 ...
```

Listing 8: Source code of the JS container application used to implement Code-Level Integration pattern

[1]https://www.npmjs.com/
[2]https://gulpjs.com/
[3]https://webpack.js.org/

P5 - Client-Side Includes (CSI)

Context: Code-level integration of microfrontends allows only a limited independence to the teams that are developing them; every time one microfrontend is updated, the dependencies of the containing application need to be updated for the changes to take effect.

Problem: How to allow *runtime* fetching and composition of microfrontends on the client-side?

Solution: As a web page is loading, fetch each microfrontend at the client-side using AJAX and insert the microfrontend into the DOM at the desired position.

Picture: Figure 5.8 shows a general overview of the solution explained in the previous paragraph. The web page contains slots for the microfrontends, and the JS on the web page fetches the microfrontends using AJAX requests which are then inserted into their respective slots.

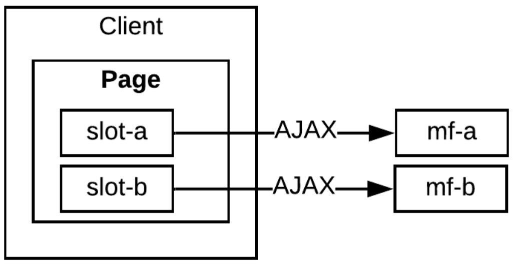

Figure 5.8: Microfrontends are included in the web page by fetching the microfrontends using AJAX calls

Result/Analysis:

QA/Property	Positives	Negatives
Independence	As the codebases are not shared between the teams, each team has independence to develop, test, and deploy its microfrontend as long as it follows the established (naming) conventions, as well as technology constraints set by the organization.	As the microfrontends run within the same web page, there is no runtime independence.

Maintainability	- Smaller codebases for each microfrontend lead to better understandability, and hence better maintainability of code. - No extra proxy server is required. - There is very little to no coupling between the infrastructure or applications.	More effort is required to maintain the separate infrastructure for each microfrontend.
Resilience	There is infrastructure-level fault isolation, and error containment. For example, if one of the microfrontends fails while the web browser is waiting for a response from it, all the other microfrontends can continue to serve their responses.	- As compared to a monolithic application, the fact that each microfrontend is separately hosted and needs to be requested using AJAX introduces more points of possible failures. - As the microfrontends run within the same web page, it is possible for an issue/bug in one microfrontend to cause another microfrontend to dysfunction. For example, if one microfrontend erroneously changes a property/function on the *window* object that another microfrontend relies on, the latter microfrontend may dysfunction as a result. - Moreover, there is no error containment or fault isolation at the application level.
Ease of migration	- At the application level, only requires refactoring the existing monolith application according to the naming conventions established for each microfrontend. - The cookies may have to be forwarded when requesting a microfrontend; however, doing so using AJAX on the client-side is fairly trivial.	- Requires setting up separate infrastructure for each microfrontend. - Requires the use of some client-side code/library capable of making AJAX requests and then embedding the response in the web page. Additionally, if the response contains script tags, the library must ensure that the scripts are executed.

Performance	As the microfrontends are composed into a single web page, the user need not switch between pages, and the UI also remains consistent.	- As each microfrontend is fetched on the client-side using AJAX, the total network requests increase as a result. - The UX deteriorates as the user experiences a "jump" in the content while the different microfrontends are fetched and displayed. - Client-side helper used to fetch and display microfrontends increases the filesize of the frontend application but the client-side helper is usually a small JS snippet; therefore, the increase in filesize is negligible.
Miscellaneous	n/a	The SEO performance of a web page whose content is included on the client-side is inferior to that of a monolithic application rendered on the server-side.

Table 5.7: Analysis of Client-Side Includes (CSI) pattern

Implementation: On a higher level, this pattern can be implemented by making an AJAX request to an endpoint and then inserting the response into the document, possibly using the *innerHTML*[1] property of a DOM node. There are several CSI libraries that can be used to implement this pattern with some popular ones such as hinclude.js[2], h-include.js[3], html-include and include-fragment-element[4]. A more basic approach is to use jQuery's *load*[5] function to load the HTML and CSS for the microfrontend, and then combine it with jQuery's *getScript*[6] function to load the JS associated with that microfrontend.

Sample Code: The code listing 9 shows part of a web page that is composed on the client-side using jQuery. Lines 8-9 cause each microfrontend to be loaded using AJAX, and have it is inserted into the appropriate *<div>* in the document.

[1]`https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML`
[2]`http://mnot.github.io/hinclude/`
[3]`https://github.com/gustafnk/h-include`
[4]`https://github.com/github/include-fragment-element`
[5]`https://api.jquery.com/load/`
[6]`https://api.jquery.com/jquery.getscript/`

```html
<html>
  ...
  <body>
    <div id="mf-a"></div>
    <div id="mf-b"></div>

    <script>
      $('#mf-a').load('http://mf-a.com');
      $('#mf-b').load('http://mf-b.com');
    </script>
  </body>
</html>
```

Listing 9: Source code of an HTML document that loads microfrontends using CSI

P6 - Client-Side Microfrontend per View

Context: SPAs are increasingly popular as they have provided a means to manage the growingly complex frontend code. SPAs enable writing modular code by organizing the code into *Components*, and *Services*. Components can be seen as representations of the different elements on a web page, whereas Services can be seen as cross-cutting functionality that is used by several web page elements. Despite the modularity, at some point an SPA grows into a hard-to-manage monolith; especially if several teams are working on the same SPA.

Problem: How to split a monolithic SPA such that one *View* is controlled by one SPA?

Solution: Each team delivers its feature as a separate SPA. A client-side container application provides a high-level API that loads/unloads the relevant SPA depending on the URI. Additionally, the high-level API also controls access to cross-cutting concerns such as *localStorage*, *navigator* etc. To avoid sharing (and hence coupling) between SPAs, the high-level API may also perform some cleanup after one SPA in unloaded and before another SPA is loaded. Typically, each SPA must implement certain life-cycle methods, that the high-level API invokes at the right time.

Picture: Figure 5.9 shows a web page that contains one SPA per view. The web page has a slot for containing an SPA and the client-side code loads/unloads the relevant SPA into the slot as the route changes on the client-side.

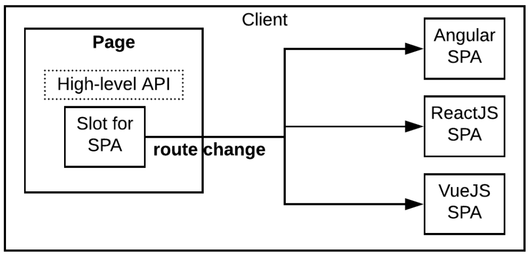

Figure 5.9: One SPA in one View

Result/Analysis:

QA/Property	**Positives**	**Negatives**
Independence	Each team has independence to develop, test, and deploy its microfrontend as an SPA as long as that SPA follows the API exposed by the client-side framework that is used to manage the different SPAs. This also implies that each team is free to choose its SPA framework.	Although the different SPAs run within the same web page, the fact that only one SPA is operative inside a single *view* provides a limited form of client-side runtime independence.
Maintainability	- Smaller codebases for each microfrontend lead to better understandability, and hence better maintainability of code. - The different microfrontends can be statically hosted to the *same* CDN.	- Each microfrontend - usually in the form of a JS package - must be statically hosted possibly to a CDN. While this shifts some of the responsibility to the CDN provider, the teams developing the microfrontends still have some responsibility of ensuring that their microfrontends are highly available. - The client-side framework has a strict API; therefore, the SPAs are coupled to the client-side framework.

Resilience	- Each microfrontend is statically hosted, possibly to CDNs that are highly available. - As each microfrontend runs in a separate view, an exception in one microfrontend does not cause other microfrontends to break. - The API provided by the client-side framework usually allows SPAs to perform cleanup before they are unloaded; this ensures that there is lesser interference between the SPAs. - The framework used to build and coordinate the different SPAs also allows isolating failures.	As the microfrontends still run within the same web page, erroneous changes to the *window* object may still cause some microfrontends to dysfunction if they rely on a property/function of the window that is erroneously changed.
Ease of migration	Only requires static hosting of SPA; does not require running complex web servers and concerns such as HTTP header forwarding.	- Requires substantial refactoring of the existing codebase to follow the API provided client-side framework that manages the coordination of the different SPAs. - Each microfrontend needs to be statically hosted, possibly to a CDN. This will require at least a build pipeline that uploads the microfrontend to a CDN.
Performance	The Code Splitting and Lazy Loading features of modern module bundlers allow fetching only the current microfrontend being viewed by the user.	- The UX deteriorates if the transition while navigating between different microfrontends is not smooth. Such transitions are usually much faster as compared to the classic page reload; however, they may be noticeable in case they take longer than 100ms (60). - The client-side framework and well as the SPA frameworks increase the filesize of the overall frontend application.
Miscellaneous	n/a	The SEO of an SPA-based application deteriorates as compared to that of a content-based web page.

Table 5.8: Analysis of Client-Side Microfrontend per View pattern

Implementation: On a higher level, this pattern requires some client-side code that is capable of loading and unloading a single, SPA at a time based on the URI. This pattern can be implemented using the Single-SPA framework[1], using frint.js[2] (limited to React and Vue.js), or using the Mooa[3] framework (limited to Angular only). Each SPA then needs to implement methods that allow it to *bootstrap*, *mount*, and *unmount* to the DOM.

Sample Code: The code listing 10 shows this pattern implemented using the Single-SPA framework. Lines 4-13 use SystemJS[4] for in-browser module (microfrontend) loading. The microfrontends as well as the Single-SPA framework are loaded as JS modules. The modules can be loaded during build-time as well, but that approach leads to coupling because the code for the container application needs to be re-released every time one of the microfrontends is changed. Lines 22-44 perform the actual loading of the modules and their registration using Single-SPA's API. In the listing, the *activeWhen* function is notable as it determines when each microfrontend is activated, and visible to the user.

```
1  <html>
2    <head>
3      ...
4      <meta name="importmap-type" content="systemjs-importmap">
5      <script type="systemjs-importmap">
6        {
7          "imports": {
8            "mf-a": "http://mf-a.com/main.js",
9            "mf-b": "http://mf-b.com/main.js",
10           "single-spa": "https://cdn.../single-spa.min.js"
11         }
12       }
13     </script>
14     ...
15   </head>
```

[1]https://single-spa.js.org/
[2]https://frint.js.org/
[3]https://github.com/phodal/mooa
[4]https://github.com/systemjs/systemjs

```

  <body>
  ...
    <div id="spa-slot"></div>

    <script>
      System.import('single-spa').then(
        (singleSpa) => {

          singleSpa.registerApplication({
            name: 'mf-a',
            app: () => System.import('mf-a'),
            activeWhen: () => location.pathname.startsWith('/mf-a'),
            customProps: {
              domElementGetter: () => document.getElementById('spa-slot')
            }
          });

          singleSpa.registerApplication({
            name: 'mf-b',
            app: () => System.import('mf-b'),
            activeWhen: () => location.pathname.startsWith('/mf-b'),
            customProps: {
              domElementGetter: () => document.getElementById('spa-slot')
            }
          });

        singleSpa.start();
      });
    </script>
  </body>
</html>
```

Listing 10: Source code of an application that implements a microfrontend per view using Single-SPA

The code listing 11 shows a sample application that follows the Single-SPA API. This application is distributed as a JS package, and it is usable as the *mf-a*, or *mf-b* applications from the previous code listing.

```
...
export function bootstrap(props) {
    ...
    return Promise.resolve();
}

export function mount(props) {
    ...
    return Promise.resolve();
}

export function unmount(props) {
    ...
    return Promise.resolve();
}
```

Listing 11: Source code of an SPA that follows the Single-SPA API

P7 - Client-Side Multiple Microfrontends per View - Variant of P6

Context: The true concept of microfrontends revolves around the ability to have multiple applications (SPAs) on the same web page. For example, one microfrontend can be running on Angular, whereas another one running on ReactJS, and yet another one running on VueJS. This is fairly common for larger organizations that want to build a complex application which comprises of several other smaller applications.

Problem: How to allow a web page to contain *multiple* SPAs (microfrontends) within the same View?

Solution: In principal, this pattern extends the Client-Side Microfrontend per View pattern by allowing multiple microfrontends to coexist on the same web page.

Picture: Figure 5.10 shows a web page that contains multiple SPAs on the same web page realized using some client-side code that acts as a high-level API to manage the SPAs.

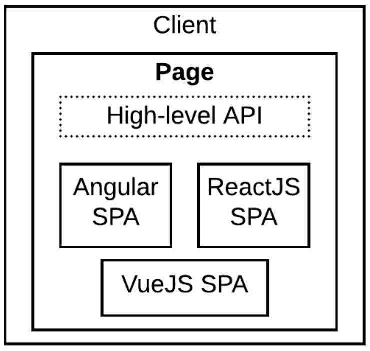

Figure 5.10: Multiple SPAs on the same web page

Result/Analysis:

Table 5.9 only lists the factors of this variant pattern that differ from its base pattern (P7).

QA/Property	Positives	Negatives
Independence	Each team has independence to develop, test, and deploy its microfrontend as an SPA as long as that SPA follows the API exposed by the client-side framework that is used to manage the different SPAs. This also implies that each team is free to choose its SPA framework.	As the SPAs run within the same view, there is no runtime independence.
Performance	Better UX as compared to P7 because the user need not transition from one microfrontend to another, but the user can engage with the different microfrontends within the same view.	n/a

Table 5.9: Analysis of Client-Side Multiple Microfrontends per View pattern

Implementation: On a higher level, this pattern requires the presence of a high-level, SPA framework-agnostic, client-side API that allows the management of multiple SPAs. Single-SPA, Piral[1], Mooa and frint.js are such frameworks where Single-SPA supports most modern SPA frameworks, Piral only supports ReactJS, Mooa only supports Angular whereas frint.js supports ReactJS and VueJS.

Sample Code: The sample code for the base pattern (P7) may be reused for this pattern with only changes to the *activeWhen* property of each application which should have the same value if both the applications are to appear in the same view.

[1]`https://piral.io/`

P8 - Client-Side Composition using Iframes

Context: If different microfrontends are included in the same web page, they all share the web page's DOM and other browser APIs. This sharing may generate unintended consequences; for example, a variable defined on the global *window* object by one microfrontend may be overwritten by another microfrontend, thereby possibly causing the former microfrontend to dysfunction.

Problem: How to provide a greater level of isolation on the client-side for multiple microfrontends within the same View?

Solution: Compose a web page on the client-side using iframes, where each microfronted is embedded into the web page as a separate iframe. Browsers provide a separate browsing context for each iframe, which is suitable for ensuring the desired isolation between microfrontends on the same web page.

Picture: Figure 5.11 shows a simple overview of this pattern where a web page comprising of two microfrontends is composed at the client-side by fetching the microfrontends and placing them as iframes into the web page.

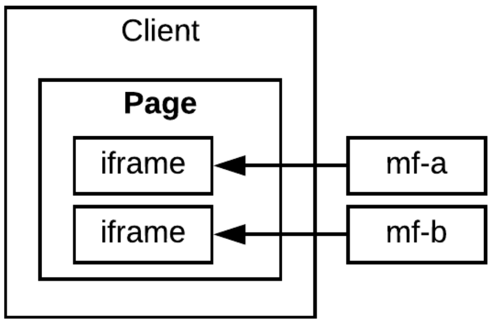

Figure 5.11: Client constructs a web page using microfrontends embedded as separate iframes

Result/Analysis:

QA/Property	Positives	Negatives

Independence	As web browsers provide a separate browsing context for each iframe, each team has complete independence in developing, testing, and deploying its application, as well as in choosing its tech stack. Needless is to say that the microfrontends have runtime independence too as they run inside separate browsing contexts.	n/a
Maintainability	- Smaller codebases for each microfrontend lead to better understandability, and hence better maintainability of code. - No extra proxy server is required. - There is very little to no coupling between the infrastructure or applications.	More effort is required to maintain the separate infrastructure for each microfrontend.
Resilience	As each microfronted runs inside an iframe, the browser provides a separate browsing context, error containment, as well as fault isolation.	As compared to a monolithic application, the fact that each microfrontend is separately hosted and needs to be requested using HTTP introduces more points of possible failures.

Ease of migration	Only requires splitting the existing application across domains, and then embedding each microfrontend into its own iframe.	- As iframes are sandboxed, communication between the microfrontends needs to be mediated by the parent window. This requires establishing conventions for events and the data that will be exchanged between the microfrontends. - Moreover, the size (width and height) of iframes is set by the parent window. Therefore, as an iframe is loaded, it needs to exchange its desired size with the parent window which will set it. - Depending on the implementation, the communication between the iframes may require the use of client-side helper libraries. - Requires separate infrastructure for each microfrontend to be set up before this pattern can be adopted.
Performance	- As the microfrontends are composed into a single web page, the user need not switch between pages.	- Iframes can cause UI quirks thereby leading to inconsistencies in the UI. - As iframes are separate browsing contexts within a web page, the browser must fetch each iframe by making extra HTTP requests; this additional network latency causes the UX to deteriorate. - Although, sharing static resources between microfrontends causes coupling, not sharing commonly used resources means the browser must redownload them for each microfrontend, thereby negatively affecting performance. - Communication between iframes requires extra client-side helpers which increases the filesize of the client-side code.

Miscellaneous	n/a	- Iframes have a negative impact on SEO as compared to "flat" HTML web pages. - The iframes-based approach is not suited for microfrontends that show content beyond the space allocated to them. For example, if an iframe is only allocated a 100 pixels by 100 pixels space in the parent web page, it cannot show content items such as drop downs, and modals outside of the space allocated to it.

Table 5.10: Analysis of Client-Side Composition using Iframes pattern

Implementation: The iframe tag is widely supported by all major browsers; each microfrontend can be embedded in a web page using the iframe tag. The browser then takes care of fetching the web page for each microfrontend and displaying it within the containing web page. To communicate between the microfrontends, there are a few possibilities: the native *postMessage*[1] API can be used to broadcast messages between the iframes and the parent document, whereas Publish-Subscribe libraries can also be used that provide a higher-level API over the native *postMessage* API.

Sample Code: The code listing 12 shows a web page that is composed using two microfrontends embedded as iframes. Lines 4-5 show the use of the *iframe* elements; the web browser loads the content contained in the *src* attribute of the tag. Lines 8-14 show how an event handler function can be attached to the *message* event which is invoked every time a browser *window* receives a message. The web page containing the iframes acts as the *parent* to each of the iframes that can pass *messages* to each other via the *parent.*

```
1 <html>
2   ...
3   <body>
4     <iframe src="https://mf-a.com" />
5     <iframe src="https://mf-b.com" />
6 
```

[1]https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage

```
    <script>
      window.addEventListener('message', (msg) => {
        const microfrontendId = msg.data[0];
        const messageType = msg.data[1];
        const messageData = msg.data[2];

        // 1. Parse message.
        // 2. Perform some action based on `messageType`.
      });
    </script>
  </body>
</html>
```

Listing 12: Source code of an HTML document that loads microfrontends using iframes

The code listing 13 shows part of the JS code for a microfrontend embedded as an iframe. Lines 1-7 attach an event handler function that listens to *messages* that the iframe receives. Lines 10-13 dispatch a *message* to the *parent* web page that will either perform some action or relay the message to another microfrontend, depending on the type of the message it has received.

```
1      window.addEventListener('message', (msg) => {
2        const microfrontendId = msg.data[0];
3        const messageType = msg.data[1];
4        const messageData = msg.data[2];
5
6        // 1. Parse message
7        // 2. Perform some action based on `messageType`.
8      });
9
10     window.parent.postMessage(
11       ['mf-a', 'mf-b:do-action', actionData],
12       '*'
13     );
```

Listing 13: Sample source code for event-handling routines for a microfrontend embedded as an iframe

Hybrid Patterns

P9 - Hybrid Fragment Composition

Context: Server-Side-only composition of microfrontends negatively affects the perceived performance of a web page as it takes longer for the web server to return a response to the web client. Client-Side-only composition is not ideal either because it is not SEO-friendly, and because the user experiences a "jump" in the page as content is fetched and displayed on the web page.

Problem: How to allow the benefits of server-side as well as client-side composition of microfrontends?

Solution: As a client requests a web page, the server returns the "above-the-fold" content of a web page such as the navigation bar, and the banner using ESI whereas the remaining content is then loaded at the client-side using CSI, in an asynchronous manner.

Picture: Figure 5.12 shows a simple overview of this pattern where a web page comprises of two components; *mf-a* component which contains the *pre-render* property, and *mf-b* component which does not. As the client requests the web page, the server first fetches the *mf-a* microfrontend which is separately hosted. After the server receives the response, it inserts the rendered version of the component into the web page and immediately returns the response to the client. The client parses the web page and as it encounters the *mf-b* component, it fetches *mf-b* asynchronously using CSI.

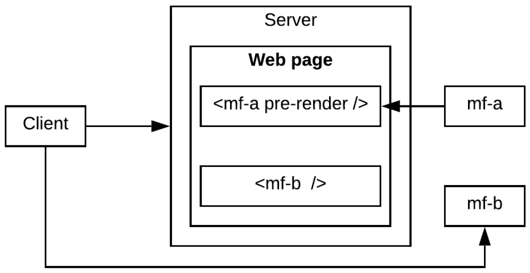

Figure 5.12: The "above-the-fold" microfronted is return immediately whereas other microfrontends are fetched asynchronously

Result/Analysis:

QA/Property	Positives	Negatives

Independence	As the codebases are not shared between the teams, each team has independence to develop, test, and deploy its microfrontend as long as it follows the established (naming) conventions, as well as technology constraints set by the organization.	As the microfrontends run within the same web page, there is no runtime independence.
Maintainability	- Smaller codebases for each microfrontend lead to better understandability, and hence better maintainability of code. - No extra proxy server is required. - There is very little to no coupling between the infrastructure or applications.	More effort is required to maintain the separate infrastructure for each microfrontend.
Resilience	There is infrastructure-level fault isolation, and error containment. For example, if one of the microfrontends fails while the web browser is waiting for a response from it, all the other microfrontends can continue to serve their responses.	- As compared to a monolithic application, the fact that each microfrontend is separately hosted and needs to be requested using HTTP introduces more points of possible failures. - As the microfrontends run within the same web page, it is possible for an issue/bug in one microfrontend to cause another microfrontend to dysfunction. For example, if one microfrontend erroneously changes a property/function on the *window* object that another microfrontend relies on, the latter microfrontend may dysfunction as a result. - Moreover, there is no error containment or fault isolation at the application level.

Ease of migration	At the application level, only requires refactoring the existing monolith application according to the naming conventions established for each microfrontend.	- Requires setting up a server that is capable of recognizing, and then requesting microfrontends that need to be pre-rendered. - The server may have to determine the forwarding of HTTP headers. - Requires each microfrontend to be separately hosted. - Client-side helper is required that is able to asynchronously fetch certain microfrontends.
Performance	- The "above-the-fold" microfrontends can be immediately returned to the user thus improving the perceived performance of the web page. - As the microfrontends are composed into a single web page, the user need not switch between pages, and the UI also remains consistent.	- As compared to a monolithic application, the microfrontends need to be individually fetched by making additional network requests. - Client-side helper that is used to asynchronously fetch microfrontends increases the filesize of the client-side code.
Miscellaneous	The "above-the-fold" content is server-side rendered so it contributes well towards SEO; however, content that is asynchronously fetched on the client-side adversely affects SEO.	n/a

Table 5.11: Analysis of Hybrid Fragment Composition pattern

Implementation: On a higher level, this pattern requires a web server that is capable of recognizing and immediately serving the critical sections of a web page, whereas the less critical sections of the page are served later on. The OpenComponents[1] framework allows the implementation of this pattern by enabling Client-Side as well as Server-Side rendering of components. In this way, the above-the-fold components can be rendered on the server-side and returned to the user right away whereas other components are fetched and rendered on the client-side.

[1] `https://opencomponents.github.io/`

Sample Code: The code listing 14 shows a sample HTML document that uses a hybrid approach to implement microfrontends. Line 4 shows the "header" component which is marked with the *pre-render* property that causes the web server to to immediately render the header and return it to the Web browser. Line 5 show the "body" component which is fetched later by client-side JS by means of an AJAX request. Line 7 refers to the client-side helper generated by the OpenComponents framework; the helper parses the *<oc-component>* elements on the web page and performs subsequent actions such as fetching microfrontends using AJAX requests.

```html
1  <html>
2    ...
3    <body>
4      <oc-component href="https://mf-header.com" pre-render  />
5      <oc-component href="https://mf-body.com" />
6      ...
7      <script src="https://mf.com/oc-client.js"></script>
8    </body>
9  </html>
```

Listing 14: Source code of an HTML document that loads microfrontends using a hybrid approach

5.3 Comparison of Patterns

Table 5.12 provides an overview of the QA/property supported by each of the patterns. The first column of the table represents a QA/property whereas the first row of the table represents the patterns. A plus sign (+) in any cell indicates that the pattern in question supports the property in that row, and doing so has a positive effect on the system. Conversely, a minus sign (-) in any cell indicates that the pattern in question does *not* support the said property and therefore this has a negative effect on the system. Finally, a tilde (∼) indicates that the pattern neither clearly supports nor does it clearly *not* support the property in question. In such cases, please refer back to the detailed analysis of the pattern in the catalogue for an explanation on how the pattern affects the property in question.

Table 5.12: Overview of patterns

	P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**
Independence									
Development	+	-	+	+	+	+	+	+	+
Testing	+	-	+	+	+	+	+	+	+
Deployment	+	∼	+	+	+	+	+	+	+
Tech autonomy	+	∼	∼	∼	∼	∼	∼	+	∼
Runtime	+	-	-	-	-	∼	-	+	-
Maintainability									
Smaller individual codebases	+	-	+	+	+	+	+	+	+
All microfrontends hosted together	-	+	-	-	-	∼	∼	-	-
Fewer operational infrastructure such as proxy servers, CDNs	-	+	+	+	+	∼	∼	+	+
Decoupled infrastructure and applications	+	-	+	-	+	-	-	+	+
Resilience									
Fewer points of failure	-	+	∼	+	∼	+	+	∼	∼
Separate global *window* object	+	-	-	-	-	-	-	+	-
Fault isolation	+	-	∼	-	∼	+	+	+	∼
Error containment	+	-	∼	-	∼	+	+	+	∼
Ease of migration									
Fewer refactoring of existing application	-	+	+	+	+	-	-	∼	+
Fewer infrastructure required	-	+	-	-	-	∼	∼	-	-
No forwarding of extra HTTP headers	-	+	-	+	+	+	+	-	-
Fewer use of extra client and/or server-side framework(s)	+	∼	∼	-	-	-	-	∼	-
Performance									
Minimally extra network latency	-	+	-	-	-	+	+	-	-
Stay on same webpage	-	+	+	+	+	∼	+	+	+
Consistent UI	-	+	+	+	+	∼	∼	-	+
Smaller filesize of client-side application	+	+	+	-	∼	-	-	-	-
Miscellaneous									
Better SEO	-	+	+	-	-	-	-	-	∼

Based on the overview of patterns, certain general as well as specific conclusions can be made about the patterns. Firstly, the patterns have been split into categories only for the ease of organization; however, the split mirrors the progression of Web frontend applications. Server-side patterns are usually more suited to the classic monolithic applications where the server returns the complete web page including most of the HTML content; the accompanying CSS and JS perform manipulations on the existing content. Client-side patterns on the other hand are more suited to modern approaches to frontend applications including SPAs that relying almost exclusively on DOM manipulations for the insertion of content in the web page using JS. Nevertheless, the patterns are not always mutually exclusive to each other. For example, iframes can be used in conjunction with any other pattern if deemed useful.

Secondly, all patterns help improve the maintainability if compared against a monolithic application; this is solely based on the observation that all patterns allow a level of independence greater than that allowed by a classic monolithic application. If the level of independence is considered in greater detail, *Server-Side Microfrontend per View* and *Client-Side Composition using Iframes* also allow runtime independence whereas all other patterns do not allow runtime independence. While runtime independence helps improve the resilience of the application, it does bring with it some challenges. For Server-Side Microfrontend per View, as the microfrontends are "physically" separated, additional effort is required to implemented the pattern in the first place. For Client-Side Composition using Iframes, iframes bring UI and UX related challenges and they also require the setting up of communication mechanisms between the microfrontends.

5.4 Microfrontends reference application

As stated previously, the frontend of a benchmark microservices project (SockShop (54)) was implemented using most (7 out of 9) microfrontend patterns as part of this thesis. The objective was to provide context for the analysis of the patterns as well as to act as a reference (50) implementation for microfrontend patterns that could be useful in the future for practitioners as well as researchers.

Figure 5.13 shows the UI of the application where it contains three microfrontends: User (enclosed in a red border), Carts (enclosed in a blue border), and Catalogue (enclosed in a green border). The original SockShop (54) project also contained several other domains

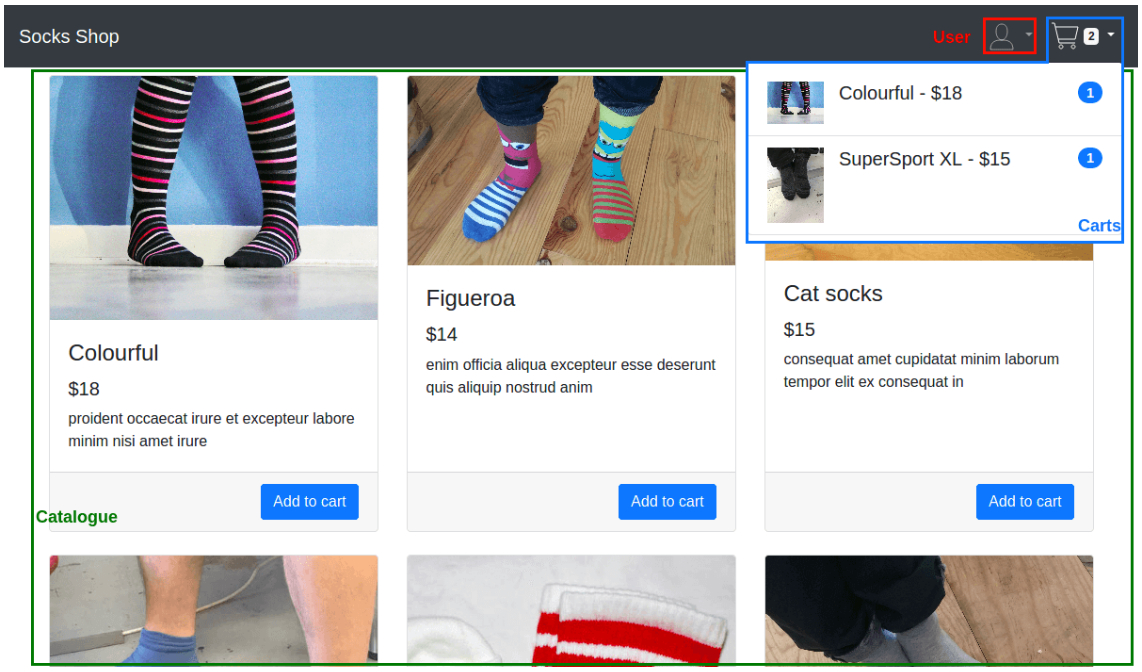
Socks Shop
User
2
Colourful - $18
1
SuperSport XL - $15
1
Carts
Colourful
$18
proident occaecat irure et excepteur labore minim nisi amet irure
Add to cart
Figueroa
$14
enim officia aliqua excepteur esse deserunt quis aliquip nostrud anim
Add to cart
Cat socks
$15
consequat amet cupidatat minim laborum tempor elit ex consequat in
Add to cart
Catalogue

Figure 5.13: Socks Shop - A reference microfrontends application

(hence also other microservices) which are typical to the e-commerce domain; however, only the three aforementioned microfrontends were chosen so as to scope down the implementation and make it manageable within the timeframe of this thesis. A typical user flow through the UI requires a user to,

1. To register and/or login (handled by the User microfrontend).
2. To view a list of socks (handled by the Catalogue microfrontend).
3. To add an item in the catalogue to the user's Cart (handled by the Carts microfrontend).

Please note that all patterns in the catalogue but *Server-Side Microfrontend per View* and *Client-Side Microfrontend per View* were implemented. The former pattern was not implemented since it requires a sufficient set up: separate infrastructure per microfrontend which is not feasible given the timeframe of this thesis. The latter pattern was not implemented since its more general variant (Client-Side Multiple Microfrontends per View) was implemented instead; still implementing the base patter would have resulted in needless repetition. The final implementation can be seen in the Github repository (50) of the project; the documentation in the project contains more operational as well as technical details about the project as well as detailed information about each pattern. Those details have not been included here as they are not central to the research questions of the thesis.

5.5 Summary

The GLR revealed that there has been an increasing interest in microfrontends and the interest is expected to continue increasing as Web frontend applications become more complex and harder to manage. Moreover, independence and scalability remain the most mentioned QAs in the context of microservices and microfrontends.

A total of 9 implementation patterns for microfrontends were compiled into a catalogue; the patterns were grouped into: Client-Side, Server-Side, and Hybrid patterns. In general, all patterns provide better maintainability as compared to a monolithic application. As for Independence, only two patterns (Server-Side Microfrontend per View and Client-Side Composition using Iframes) provide runtime independence too.

6

Discussion and Future Work

6.1 Discussion

Microservices are increasingly adopted by organizations to deal with the growing complexity of backend systems, and to deal with the growing scalability demands. Recently, microfrontends have appeared as a way to bring much of the concepts and advantages of microservices to the Web frontend. During this project, a GLR was conducted to compile a catalogue of implementation patterns for microfrontends. The patterns were then analysed to see the effect they have on QAs that are considered important in the context of microservices, and microfrontends. Finally, the frontend of a benchmark microservices application was implemented using most of the microfrontend patterns; the implementation helped analyse the effect each pattern has on certain QAs and the implementation also serves as a future reference[1] for practitioners as well as researchers.

6.1.1 Findings

As the implementation patterns were compiled, three categories of patterns emerged: server-side patterns, client-side patterns and hybrid patterns. The server-side patterns are more suited to traditional monolithic web pages that are content-oriented. These patterns can be seen as the first steps towards microfrontends as they require the least number of structural changes. Existing web pages and applications only need to be split up according to their business domains and the existing code needs to be slightly adjusted based on the naming conventions established within the teams. On the other hand, client-side patterns cater to the modern application-oriented web pages where each domain inserts content into the web page by means of JS. Therefore, these patterns require a greater restructure and

[1] `https://github.com/aliameer/microfrontend-patterns`

refactoring of existing web pages. For example, some patterns require the creation of a separate SPA for each domain and also require ensuring that each SPA complies with the API offered by the client-side framework used to coordinate the microfrontends.

The GLR revealed that *Independence* and *Scalability* are two of the most emphasized QAs in the context of microservices and microfrontends. Independence is the core QA that actually enables many of the other QAs. For instance, microfrontends are more *maintainable* because they are independent as compared to monolithic applications. Independence in this context primarily applies to development, tech autonomy, testing, deployment, and maintenance. As explained in a previous chapter, *scalability* in the context of microfrontends only refers to development scalability rather than computational scalability. While some of the primary resources explicitly refer to "scaling frontend development", others loosely use the term "scalability" without qualifying it.

Microfrontends are an interesting concept that will see increased adoption in the future. If *independence* is considered the central QA that enables other QAs too, only two microfrontend patterns allow "absolute" independence during run-time too: *Server-Side Microfrontend per View*, and *Client-Side Composition using Iframes*. Strictly speaking, the former pattern cannot be regarded as a "pure" microfrontend pattern in that it merely involves the integration of completely separate applications using hyperlinks; the pattern has been included in the pattern catalogue for the sake of completeness. The latter pattern on the other hand is not suited to all use-cases and the independence enabled by iframes brings other problems such as setting up communication channels between microfrontends. All the remaining patterns allow a limited form of independence where each microfrontend must run within the same web page and work within the context of the web page as established by the web browser.

This highlights an important difference between the context of microservices and microfrontends: while microservices may allow absolute independence between the different services as they are "physically" separated, microfrontends can only allow a limited independence because eventually, all the microfrontends need to run on the same web page within the browser. The microfrontends must share the global *window* object as well as other APIs provided by the web browser; therefore, certain naming conventions between the teams need to be established and followed. In this sense, microfrontends require a much tighter integration among each other as compared to microservices. An important point to consider is that although microfrontends have stemmed from microservices, it may not be possible to attain all the advantages enabled by the microservices architecture when applied to Web frontend, mainly because of the run-time (context) differences in the case of microfrontends.

6.1.2 Limitations

This project has a few limitations mostly because of the small scale it was conducted on. Firstly, the GLR was conducted by a single person (the only author of the thesis) which means there is a tendency for bias to creep in and go unnoticed despite the adoption of guidelines for conducting systematic reviews. Secondly, the reference application is very simplistic in terms of the interactions it allows and in terms of the number of different microfrontends it has: only three. The interactions were simplified, and the number of microfrontends was limited to scope down the nature of the project; however, as one can imagine, a real-world e-commerce application allows much more complex interactions and routinely has far more domains (hence microfrontends).

Thirdly, the reference application was implemented by a single person which greatly limits the range of code styles and code organization that emerge in bigger real-world projects. Consequently, it is not possible to perform a meta-analysis of the reference application such as looking at how code commits are split and structured across the project. Finally, informal architecture/pattern analysis techniques had to be used to assess the patterns as opposed to formal architecture analysis techniques. The latter usually require more detailed architectural descriptions and the involvement of different stakeholders of the software system, each bringing in their viewpoint of the software system.

6.2 Future Work

While conducting this project and performing a GLR during it, several lessons were learnt that can be useful in the future. Firstly, the approach adopted to address a topic is greatly determined by the advancements that topic has seen so far. For example, the topic of microfrontends has stemmed out of the industry and there is very little academic work about it; consequently, this thesis addresses a *general* aspect of compiling and analyzing patterns rather than performing some very specific quantitative experiment. Hopefully, such general and high-level works on microfrontends can pave the way for more specific works to follow. Secondly, basing a research work on grey literature needs to be carefully pondered over simply because of how grey literature differs from white literature. The language used in grey literature is not as precise as that used in white literature; moreover, the claims made in the former are not always substantiated as those made in the latter. Therefore, observations made in grey literature need to be carefully analyzed before basing further work on them.

The concept of microfrontends is gaining popularity and it will continue to do so much like microservices have. In general, there has been very little academic work on microfrontends; therefore, more academic attention is required to further explore the concept of microfrontends. A simple follow-up study in the future can hopefully help reveal more microfrontend patterns; perhaps, those that allow greater runtime independence among microfrontends. Moreover, this proof-of-concept project only contains three microfrontends which is far simpler as compared to real-world products that routinely contain dozens of domains. In the future, a bigger microservices project should be used to implement the reference microfrontends project.

Additionally, the analysis of the patterns as well the development of a microfrontends benchmark should be performed by a group. This will allow the use of more formal architecture analysis techniques such as SAAM and ATAM. It will also allow meta-analysis of a microfrontends-based project to see how it differs from a monolithic application in terms of code-level metrics such as coupling and cohesion. Finally, if this project is replicated in a real-world setting, it would be interesting to analyze the patterns for QAs such as development scalability, and velocity.

7 Conclusion

During this thesis, a Grey Literature Review (GLR) was carried out to compile a catalogue of microfrontend implementation patterns. The patterns were then analyzed to see how they affect different Quality Attributes. The GLR led to a total of nine patterns organized into three sets: Server-Side, Client-Side, and Hybrid patterns. The server-side patterns are more oriented towards traditional content-centric monolithic applications whereas the client-side patterns are more oriented towards modern behavior-centric monolithic applications whose content is inserted on the client-side. The GLR also revealed that *Independence*, and *Scalability* are the most discussed QAs in the context of microservices and microfrontends, where *independence* is the key QA that enables other QAs too.

The analysis of the patterns revealed that only two microfrontend patterns enable complete independence, whereas other patterns enable a limited independence because of the runtime context (Web browser) of microfrontends. This leads to the observation that although microfrontends aim to bring the benefits of microservices to the Web frontend, it may not be possible to reap all the benefits in the Web frontend because of the difference in runtime contexts of the two. While microservices are physically separated, microfrontends need to run together on the same web page within a web browser thereby limiting independence.

GLR Primary Resources

[P1] **Micro Frontends - extending the microservice idea to frontend development**. `https://micro-frontends.org/`. [Online; accessed 15-January-2020].

[P2] **Better streaming layouts for frontend microservices with Tailor**. `https://jobs.zalando.com/tech/blog/frontend-microservices-tailor/`. [Online; accessed 15-January-2020].

[P3] **The Strengths and Benefits of Micro Frontends**. `https://www.toptal.com/front-end/micro-frontends-strengths-benefits`. [Online; accessed 15-January-2020].

[P4] **Micro-frontend Architecture in Action with six ways.** `https://dev.to/phodal/micro-frontend-architecture-in-action-4n60`. [Online; accessed 15-January-2020].

[P5] **Micro Frontend Curry**. `https://levelup.gitconnected.com/micro-frontend-curry-506b98a4cfc0`. [Online; accessed 15-January-2020].

[P6] **Breaking down the last monolith - Microfrontends**. `https://dev.to/aregee/breaking-down-the-last-monolith-micro-frontends-hd4`. [Online; accessed 15-January-2020].

[P7] **How To Implement Micro-Frontend Architecture With Angular**. `https://medium.com/bb-tutorials-and-thoughts/how-to-implement-micro-frontend-architecture-with-angular-e6828a0a049c`. [Online; accessed 15-January-2020].

[P8] **Patterns for Microfrontends**. `https://gist.github.com/FlorianRappl/c11736a0edaf4bc4d4b929928735382b`. [Online; accessed 15-January-2020].

[P9] **Micro Frontends - Microservice Approach To The Modern Web**. https://www.slideshare.net/Codemotion/ivan-jovanovic-micro-frontends-codemotion-rome2019. [Online; accessed 15-January-2020].

[P10] **Independent micro frontends with Single SPA library**. https://blog.pragmatists.com/independent-micro-frontends-with-single-spa-library-a829012dc5be. [Online; accessed 15-January-2020].

[P11] **Micro-frontends: porting the micro-service approach into the frontend**. https://frontend-development.com/micro-frontends-porting-the-micro-service-approach-into-the-frontend/. [Online; accessed 15-January-2020].

[P12] **Web Micro Frontends Introduction**. https://medium.com/@amirilovic/web-microfrontends-introduction-451f72e68c8c. [Online; accessed 15-January-2020].

[P13] **Microfrontends - part 2: integration and communication**. https://medium.com/stepstone-tech/microfrontends-part-2-integration-and-communication-3385bc242673. [Online; accessed 15-January-2020].

[P14] **Microservice Websites**. https://gustafnk.github.io/microservice-websites/. [Online; accessed 15-January-2020].

[P15] **The monolithic frontend in the microservices architecture**. https://xebia.com/blog/the-monolithic-frontend-in-the-microservices-architecture/. [Online; accessed 15-January-2020].

[P16] **Adopting a Micro-frontends architecture**. https://medium.com/dazn-tech/adopting-a-micro-frontends-architecture-e283e6a3c4f3. [Online; accessed 15-January-2020].

[P17] **Micro Frontends**. https://martinfowler.com/articles/micro-frontends.html. [Online; accessed 15-January-2020].

[P18] **Five things to consider before choosing microfrontends**. `https://rangle.io/blog/five-things-to-consider-before-choosing-micro-frontends/`. [Online; accessed 15-January-2020].

[P19] **Exploring micro-frontends**. `https://medium.com/@benjamin.d.johnson/exploring-micro-frontends-87a120b3f71c`. [Online; accessed 15-January-2020].

[P20] **Microfrontends - bringing JavaScript frameworks together (React, Angular, Vue etc)**. `https://medium.com/javascript-in-plain-english/microfrontends-bringing-javascript-frameworks-together-react-angular-vue-etc-5d401`
[Online; accessed 15-January-2020].

[P21] **Managing frontend in the microservices architecture**. `https://allegro.tech/2016/03/Managing-Frontend-in-the-microservices-architecture.html`. [Online; accessed 15-January-2020].

References

[1] Richardson. **Monolithic Architecture Patterns**. `https://microservices.io/patterns/monolithic.html`. [Online; accessed 06-March-2020].

[2] David Flanagan. *JavaScript: the definitive guide*. " O'Reilly Media, Inc.", 2006.

[3] Fowler Lewis. **Microservices**. `https://martinfowler.com/articles/microservices.html`, March 2014. [Online; accessed 24-December-2019].

[4] **Service-Oriented Architecture: Scaling the Uber Engineering Codebase As We Grow**. `https://eng.uber.com/service-oriented-architecture/`. [Online; accessed 12-May-2020].

[5] **From Monolith to Microservices, Zalando's Journey**. `https://www.infoq.com/news/2016/02/Monolith-Microservices-Zalando/`. [Online; accessed 12-May-2020].

[6] **Building Products at SoundCloud—Part II: Breaking the Monolith**. `https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-2-breaking-the-monolith`. [Online; accessed 12-May-2020].

[7] **Developer Survey Results 2019**. `https://insights.stackoverflow.com/survey/2019`. [Online; accessed 9-April-2020].

[8] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. **Architecting with microservices: A systematic mapping study**. *Journal of Systems and Software*, **150**:77–97, 2019.

[9] Claus Pahl and Pooyan Jamshidi. **Microservices: A Systematic Mapping Study.** In *CLOSER (1)*, pages 137–146, 2016.

[10] Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. **Microservices: The journey so far and challenges ahead**. *IEEE Software*, **35**(3):24–35, 2018.

[11] Nuha Alshuqayran, Nour Ali, and Roger Evans. **A systematic mapping study in microservice architecture**. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51. IEEE, 2016.

[12] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. **Microservices: yesterday, today, and tomorrow**. In *Present and ulterior software engineering*, pages 195–216. Springer, 2017.

[13] Hulya Vural, Murat Koyuncu, and Sinem Guney. **A systematic literature review on microservices**. In *International Conference on Computational Science and Its Applications*, pages 203–217. Springer, 2017.

[14] Carlos M Aderaldo, Nabor C Mendonça, Claus Pahl, and Pooyan Jamshidi. **Benchmark requirements for microservices architecture research**. In *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, pages 8–13. IEEE, 2017.

[15] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. **An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems**. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[16] **Evolving Distributed Tracing at Uber Engineering**. `https://eng.uber.com/distributed-tracing/`. [Online; accessed 26-March-2020].

[17] Holger Harms, Collin Rogowski, and Luigi Lo Iacono. **Guidelines for Adopting Frontend Architectures and Patterns in Microservices-Based Systems**. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 902–907, New York, NY, USA, 2017. Association for Computing Machinery.

[18] **ISO/IEC 25010**. `https://iso25000.com/index.php/en/iso-25000-standards/iso-25010`. [Online; accessed 24-March-2020].

[19] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[20] Caifang Yang, Chuanchang Liu, and Zhiyuan Su. **Research and Application of Micro Frontends**. *IOP Conference Series: Materials Science and Engineering*, **490**:062082, apr 2019.

[21] Andrey Pavlenko, Nursultan Askarbekuly, Swati Megha, and Manuel Mazzara. **Micro-frontends: application of microservices to web front-ends**. *Journal of Internet Services and Information Security (JISIS)*, **10**(2):49–66, 2020.

[22] Jonathan Michael Darling, Dustin Nation, and Jibins Joseph. **Systems and methods for developing a web application using micro frontends**, June 9 2020. US Patent 10,678,600.

[23] Severi Peltonen, Luca Mezzalira, and Davide Taibi. **Motivations, Benefits, and Issues for Adopting Micro-Frontends: A Multivocal Literature Review**. *arXiv preprint arXiv:2007.00293*, 2020.

[24] Olaf Zimmermann. **Microservices tenets**. *Computer Science-Research and Development*, **32**(3-4):301–310, 2017.

[25] Luciano Baresi and Martin Garriga. **Microservices: The Evolution and Extinction of Web Services?** In *Microservices*, pages 3–28. Springer, 2020.

[26] Sam Newman. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[27] **A short history of the Web**. `https://home.cern/science/computing/birth-web/short-history-web`. [Online; accessed 12-February-2020].

[28] **2 - A history of HTML**. `https://www.w3.org/People/Raggett/book4/ch02.html`. [Online; accessed 6-April-2020].

[29] **A brief history of CSS until 2016**. `https://www.w3.org/Style/CSS20/history.html`. [Online; accessed 6-April-2020].

[30] **How to Add a User Stylesheet in Firefox**. https://davidwalsh.name/firefox-user-stylesheet. [Online; accessed 6-April-2020].

[31] **Javascript**. https://en.wikipedia.org/wiki/JavaScript#Creation_at_Netscape. [Online; accessed 6-April-2020].

[32] **Sites vs. Apps defined: the Documents-to-Applications Continuum.** https://ar.al/notes/the-documents-to-applications-continuum/. [Online; accessed 7-April-2020].

[33] **npm - Bootstrap**. https://www.npmjs.com/package/bootstrap. [Online; accessed 9-April-2020].

[34] **ECMAScript 2015 Language Specification**. http://www.ecma-international.org/ecma-262/6.0/. [Online; accessed 15-April-2020].

[35] **ExploringJS - Modules**. https://exploringjs.com/es6/ch_modules.html. [Online; accessed 15-April-2020].

[36] **JavaScript modules**. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules. [Online; accessed 15-April-2020].

[37] **Microfrontends | Technology Radar**. https://www.thoughtworks.com/radar/techniques/micro-frontends. [Online; accessed 7-March-2020].

[38] **Project Mosaic | Microservices for the Frontend**. https://www.mosaic9.org/. [Online; accessed 7-May-2020].

[39] **Micro-frontends, the future of Frontend architectures**. https://medium.com/dazn-tech/micro-frontends-the-future-of-frontend-architectures-5867ceded39a. [Online; accessed 15-April-2020].

[40] **OpenComponents - microservices in the front-end world**. http://tech.opentable.co.uk/blog/2016/04/27/opencomponents-microservices-in-the-front-end-world/. [Online; accessed 7-May-2020].

[41] **Front-end Microservices at HelloFresh**. https://engineering.hellofresh.com/front-end-microservices-at-hellofresh-23978a611b87. [Online; accessed 15-April-2020].

[42] **Modernizing Upwork with Micro Frontends**. `https://www.upwork.com/blog/2017/05/modernizing-upwork-micro-frontends/`. [Online; accessed 7-May-2020].

[43] **Experiences Using Micro Frontends at IKEA**. `https://www.infoq.com/news/2018/08/experiences-micro-frontends/`. [Online; accessed 7-May-2020].

[44] Staffs Keele et al. **Guidelines for performing systematic literature reviews in software engineering**. Technical report, Technical report, Ver. 2.3 EBSE Technical Report. EBSE, 2007.

[45] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. **Guidelines for including grey literature and conducting multivocal literature reviews in software engineering**. *Information and Software Technology*, **106**:101–121, 2019.

[46] Austen Rainer and Ashley Williams. **Heuristics for improving the rigour and relevance of grey literature searches for software engineering research**. *Information and Software Technology*, **106**:231–233, 2019.

[47] Päivi Raulamo-Jurvanen, Mika Mäntylä, and Vahid Garousi. **Choosing the right test automation tool: a grey literature review of practitioner sources**. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 21–30. ACM, 2017.

[48] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. **The pains and gains of microservices: A systematic grey literature review**. *Journal of Systems and Software*, **146**:215–232, 2018.

[49] Donna Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[50] **Microfrontend Patterns**. `https://github.com/aliameer/microfrontend-patterns`. [Online; accessed 19-June-2020].

[51] **Acme Air NodeJS**. `https://github.com/acmeair/acmeair-nodejs`. [Online; accessed 25-March-2020].

[52] **MusicStore**. `https://github.com/SteeltoeOSS/Samples/tree/2.x/MusicStore`. [Online; accessed 25-March-2020].

[53] **.NET Microservices Sample Reference Application**. `https://github.com/dotnet-architecture/eShopOnContainers`. [Online; accessed 25-March-2020].

[54] **Sock Shop : A Microservice Demo Application**. https://github.com/microservices-demo/microservices-demo. [Online; accessed 25-March-2020].

[55] **Introducing FrintJS**. https://travix.io/introducing-frintjs-a9a8cdd29812. [Online; accessed 29-March-2020].

[56] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[57] Kyle Brown and Bobby Woolf. **Implementation patterns for microservices architectures**. In *Proceedings of the 23rd Conference on Pattern Languages of Programs*, pages 1–35, 2016.

[58] **SEO Best Practices with Cloudflare Workers, Part 1: Subdomain vs. Subdirectory**. https://blog.cloudflare.com/subdomains-vs-subdirectories-best-practices-workers-part-1/. [Online; accessed 12-May-2020].

[59] **ESI Language Specification 1.0**. https://www.w3.org/TR/esi-lang. [Online; accessed 15-February-2020].

[60] **Measure Performance with the RAIL Model architectures**. https://developers.google.com/web/fundamentals/performance/rail. [Online; accessed 2-May-2020].