TU WIEN Informatics

Communication and State Management for Micro Frontend Architectures

Challenges and Solution Patterns

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Simon Hayden, BSc
Matrikelnummer 01426127

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.-Doz. Mag.rer.nat. Dipl.-Ing. Dr.techn. Rudolf Freund

Wien, 15. Juli 2023

Unterschrift Verfasser

Unterschrift Betreuung

TU WIEN Informatics

Communication and State Management for Micro Frontend Architectures

Challenges and Solution Patterns

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Simon Hayden, BSc
Registration Number 01426127

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.-Doz. Mag.rer.nat. Dipl.-Ing. Dr.techn. Rudolf Freund

Vienna, 15th July, 2023

Signature Author

Signature Advisor

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Simon Hayden, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. Juli 2023

Simon Hayden

Acknowledgements

First and foremost I would like to thank my better half Dipl.-Ing. Magdalena Steinböck. She helped me greatly by being a rubber duck during development, a scientific mentor for data presentation, and helped me find the words you read in this thesis. Secondly, Dipl.-Ing. Mag.rer.soc.oec. Christoph Mayerhofer, Bakk.techn. supported me during the research process, guiding me in finding the next steps and discussing the latest developments. Last but not least I want to thank my parents who enabled me to study at TU Vienna. Without them, this thesis would not have been possible.

Kurzfassung

Die *Micro Frontend* Architektur ermöglicht es, Teams um vertikale Features, anstatt von horizontalen Applikationsschichten zu bilden. Um am Endgerät eine interaktive und reaktive Benutzererfahrung anbieten zu können, müssen isolierte *Micro Frontend* Instanzen jedoch kooperieren. In dieser Diplomarbeit untersuchen wir mögliche Kommunikationskanäle, mit denen Zustandssynchronisierung zwischen mehreren *Micro Frontends* ermöglicht werden sollen. Mithilfe der *Action Research* Methodologie erstellen wir drei Prototypen der gleichen Webapplikation: Zuerst wird eine monolithische Applikation entwickelt, die als Vergleichsbasis dient. Danach werden zwei *Micro Frontend*-basierte Applikationen mit `iframes` bzw. Web Components entwickelt. Beide Technologien haben dabei unterschiedliche Herausforderungen und Limitationen. Zuletzt vergleichen wir die Ergebnisse anhand des Ressourcenverbrauchs und Entwicklungsaufwands bei sich ändernden Anforderungen.

Keywords: *Micro Frontend Architektur*, *Micro Frontend Integration*, *Web Browser Kommunikationskanäle*, *Action Research*

Abstract

Micro frontends have enabled developers to think in feature vertices rather than application layers. But on the client, isolated *micro frontends* need to communicate with each other to provide rich and dynamic user experiences. This thesis explores possible communication channels in order to implement state synchronization across multiple *micro frontend* instances. Using the *Action Research* methodology, we test our theory by developing three prototypes of the same web application: First, a monolithic baseline, followed by two *micro frontend* based applications using `iframes` and Web Components – each posing different challenges and limitations. Finally, we compare the prototypes' resource consumption and development overhead of changes.

Keywords: *micro frontend architecture*, *micro frontend integration*, *web browser communication channels*, *action research*

Contents

CHAPTER 1

Introduction

1.1 Motivation

Over the last few years, the microservice architecture has gained traction in web development [57, 87], as seen by Netflix and Amazon[1] for instance. Parallel to this development client-side rendering technologies such as Angular [2, 7], React [13], and Vue [34, 33] became well established in modern, highly interactive applications [77] such as Google's store or the Microsoft Office web application [21], parts of Netflix [47] as well as Facebook – the creator of the React library [13]. However, despite their popularity the two technologies appear to diverge in their goal: While microservices want to create smaller software fragments and teams, client-side rendering frameworks increase the frontend's responsibilities and therefore also its size. The resulting monolithic frontends share many issues with monolithic backends: Frozen technology stack, large and complex code base, difficulty to parallelize and isolate teams, deployment as a single artifact, etc. [23, 48, 87, 61].

To solve the issues that come with monolithic artifacts, the micro frontend architecture aims to split the frontend into multiple parts that are recombined at the frontend to provide a coherent web application – similar to the microservices in the backend. While some aspects of this rather new approach are well researched already [58, 77], there is still a lot to be done compared to more established architectures, e.g., the already mentioned microservices.

In particular, the interactions between micro frontends are yet to be scientifically analyzed. In many cases, micro frontends will not just coexist next to each other but are required to be interactive and integrated with their siblings, child component(s), and parent component(s). A simple example is a button inside a micro frontend to view news

[1] Netflix and Amazon provide many tools for developing microservices on their GitHub page: https://github.com/Netflix & https://github.com/amzn

articles. Clicking the button should bookmark the article, therefore sending a signal to the "bookmark" micro frontend.

Additionally, micro frontends might not only need to communicate actions but also state information. Continuing the example from above, assume the bookmarks are kept in a list. To tell the user whether an article has already been bookmarked, the micro frontend for displaying articles needs the list of bookmarked articles. An additional backend Application Programming Interface (API) call should be avoided, as it fetches redundant information. Therefore, the state of an article – bookmarked or not – should be shared between micro frontends. This concept can be extended to not only share the User Interface (UI) state but also datasets as a shared cache, which are accumulated during the current session.

1.2 Aim of this Work

With this master thesis, we aim to answer the question of how cross-communication between micro frontends could be implemented as well as their advantages and drawbacks. In particular, following research questions should be answered:

RQ1 What kind of messaging interfaces for communication between micro frontends are provided by modern browsers?

RQ2 How can communication patterns or other technologies be used to synchronize the state between micro frontends?

RQ3 What is the overhead on performance as well as architecturally of a specific communication and synchronization pattern?

To answer these questions the evaluation process will be split into four steps:

1. **Research existing projects and architectural patterns (RQ1).** Within this step, an as complete as possible overview of current applications of micro frontends should be gained via literature research. Due to the young age of the micro frontend pattern, some backend technologies like Server Side Includes (SSI) or approaches to similar problems (e.g., cross-communication of microservices) will be investigated as well.

2. **Analyze ways of communication between micro frontends (RQ1, RQ2).** With the results of the first step, the different architectural patterns have to be further analyzed regarding the communication channels they allow and how to use them. The goal of the second step is to formulate different communication patterns that apply to the different architectural styles.

3. **Analyze options for state synchronization (RQ2).** Based on the communication patterns, the next step is to synchronize micro frontend instances. The goal is to find ways for lateral communication to share common data between micro frontend instances.

4. **Qualitative analysis for each communicational and architectural pattern (RQ3).** In the final stage, all evaluated patterns will be quantifiably compared using performance and architectural benchmarks.

1.3 Methodology

In the first step, we use literature research to determine the current state of the art. To analyze communication and state synchronization patterns in steps two and three, we apply the Action Research methodology. This is a cyclical process consisting of problem identification and finding solutions. In our case, we iteratively improve upon solutions for micro frontend communication and state synchronization by implementing three different prototypes. Finally, in step four we use benchmarking to determine performance and architectural characteristics of our solutions. For the performance metrics, we repeatedly open each prototype in a browser and measure the application's load time and memory consumption. We analyse the architectural properties by introducing change requests to the application, which we realize in each prototype. We then analyze the number of changed files as well as categorizing the types of changes.

1.4 Limitations

Since the topic of micro frontends can be rather broad and vague, this master thesis will restrict the investigated aspects. First of all, embedding legacy applications into another (micro) frontend will not be considered. While they exhibit some properties shared with micro frontends (e.g., combining smaller parts to form a complete user interface), they lack meaningful integration with the host application.

Secondly, only Single Page Applications (SPAs) are considered. The reason is that traditional web pages are usually rendered on the backend, mitigating the need for communication and dynamic state synchronization on the client. SPAs, on the other hand, move most of the representation layer to the browser. Hence, they also need most of the logic for integrating the individual frontend fragments, requiring means for communication and state synchronization.

Thirdly, Angular [2] will be used for all frontend code. While not necessarily a restriction on the topic, it must be mentioned and considered while evaluating patterns found and described for implementing solution patterns. For this reason, the described guidelines must ensure that they apply not only to Angular applications but are generally valid. The reason for choosing this framework is the author's experience with Angular.

Finally, this work is exploratory – meaning that little previous work has been done in this field – and cannot discuss every possible solution in detail. Rather, the Action Research investigates some of the many paths software development may lead to. Basing our research on known grounds and iterating upon it, however, should prevent developing "paperware" – software solutions that solely exist in academic papers with no real-world use case.

CHAPTER 2

State of the Art

In this chapter, we explore the state of the art of micro frontends and their origin. It forms the basis for our research regarding communication patterns and state synchronization between micro frontends.

2.1 The Software Monolith

In software engineering, a monolith is a piece of software that is developed, deployed, and executed as a single fragment [71, 53, 65]. In the early days of software engineering, this architecture type was the most prevalent, as it lends itself well to desktop-native applications. For example, a desktop word-processing application might be developed via a single code base and built into a single executable (usually an installer). The deployment is done by uploading the executable to a website for users to download and install locally. In the execution phase, the user runs the program on the local machine.

This scenario shows the main advantage of monolithic architecture: its simplicity. The development life cycle has low complexity, which means less likelihood of unexpected errors. The locality of code can also be used as an advantage. For example, refactoring cross-cutting concerns are guaranteed to be updated for all consumers. Static analysis of the code can help in finding bugs early, e.g., compiling errors after updating a type.

So naturally, early web applications used the same monolithic structure. Instead of rendering native windows and frames, the representation layer of the application creates Hypertext Markup Language (HTML) strings, which are then rendered by the browser.

As applications grow, however, the disadvantages of monoliths grow too [71, 70, 57, 53, 65]. Most importantly, managing a monolithic code base does not scale well with application complexity and size. Due to the lack of isolation of code fragments from each other, it is harder to parallelize the development of individual features. Due to the coupling of code,

it is easily possible to introduce subtle bugs via side effects of changes. Hence, the code base becomes stiff and legacy code might not be able to change or be replaced easily.

Additionally, the monolith is usually developed using a single programming language and a fixed framework for cross-cutting concerns like routing incoming Hypertext Transfer Protocol (HTTP) requests to their handlers, execute them, and handle errors. As the application ages, however, modern languages, features, or libraries might emerge which could greatly improve the application's code base or performance. However, the monolith might be too large for undergoing large refactors. The development team is therefore locked into the technology they originally chose [70, 73, 71].

2.2 Microservices in the Backend

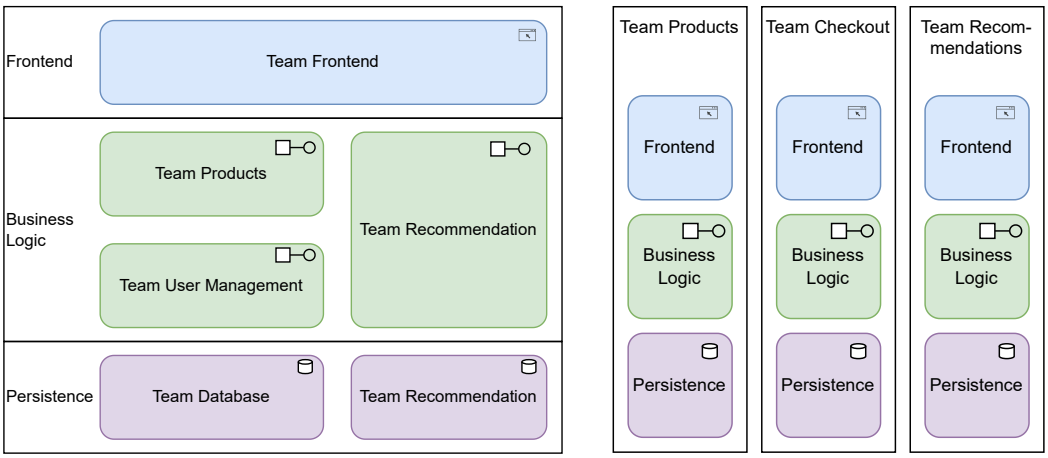
As websites become more interactive, their complexity rises as well, showing the disadvantages mentioned above. However, in comparison to desktop applications web applications are not limited to deploying everything as a single fragment. For desktop applications, downloading and running multiple executables simultaneously is impractical. But since only the final HTML is sent to the user, the origin which rendered the page is transparent to the user. So the backend of an application can be split into multiple fragments, each providing a different function.

From this idea, the microservice architecture emerged [71, 73, 70, 89]. A microservice is an isolated, small, and independently deployed and executed piece of software which provides a well-defined interface – usually a Representational State Transfer (REST) API [53]. If a microservice needs the features provided by another, it talks to the exposed API, without relying on its implementation.

This high degree of isolation solves many issues found in monolithic applications. First, since each microservice can be deployed in isolation, the application can update gradually. Deploying new features can be done individually without touching or shutting down other services. The high isolation of microservices allows teams to be split into smaller, independent teams. Therefore, they can act faster by focusing on their responsibilities only [70, 57, 53, 90, 89].

Secondly, since microservices only expose a single API, the technology stack for each of them is independent. This allows an application to be a polyglot of programming languages and frameworks [53, 70]. Hence, the application is not locked into using a single technology over the entire lifecycle. Furthermore, granular updates to the application can be done iteratively, preventing large refactors at once.

Parallel to the rise in the popularity of microservices, SPAs move the representation layer to the browser. This decouples the views from the backend. Hence, the backend is kept as a headless API. For this reason, SPAs are frequently used in combination with microservices, as the frontend is just another consumer of the outwards facing APIs.



(a) (b)

Figure 2.1: Comparison of vertical and horizontal teams. In a horizontal team structure (a), teams are divided based on responsibility. There are five teams, one team to implement the frontend monolith, four teams on the server side. Three teams develop micro services while one (Team Recommendation) also manages their own database. The other two teams rely on Team Database for their persistence layer. In the vertical team structure (b), each team is formed around features. They have complete control over the full stack.

2.3 Micro Frontend Architecture

The micro frontend architecture was first mentioned in thoughtworks' Technology Radar in 2016 [24]. They describe the architecture pattern as the frontend counterpart of the microservice architecture. Therefore, the frontend monolith is split into smaller, self-contained, modular frontend components or fragments, which are later recombined to form a complete web application [24]. Similar to microservices, micro frontends should therefore be developed, deployed, and executed in isolation from the rest of the application [78, 58, 77, 90].

One of the ways team separation is achieved is technology agnosticism [90, 58, 78]. Compared to the frontend monolith, different teams should be able to use different frameworks and libraries in different versions. That way incremental updates of dependencies in a larger application are easier. Additionally, teams may be encouraged to experiment with technologies and keep the application near the state of the art of modern web applications. Refactoring applications to new technologies is possible as well and can be done in increments. [58, 78]

However, beyond the code structure of micro frontends, the architecture pattern allows structuring teams around features rather than responsibilities. Usually, using microservices, one team is responsible for the monolithic frontend. Even if the backend uses microservices with small teams, the frontend monolith cannot scale its team size as easily. Splitting the frontend into micro components means that teams can be arranged

"vertically". Each vertical team is responsible for developing, maintaining, and deploying its feature, as seen in the example for developing a web shop in Figure 2.1 [24, 58, 77].

The goal of this team structure is to reduce the communication overhead of teams between each other and therefore improve development speed. For example, requesting a single feature no longer requires requesting changes from the database team, the backend API team, as well as the frontend team. Hence, vertical teams should naturally scale with the number of features or feature groups. [58, 77]

But micro frontends are no silver bullet either. There are many drawbacks of the architecture that have to be considered as well. The most prevalent is the added complexity of the system. Splitting the code base into multiple micro frontends means that ways for composition and communication between the parts have to be added to the system. Additionally, the infrastructure must support serving multiple parts of a web application, potentially across multiple servers. Another risk is that application-layer-based tasks span multiple, if not all, vertical teams (e.g., Application Performance Management (APM), security constraints, etc.). This could lead to even more communication and coordination overhead, compared to horizontal teams. [78, 58, 77]

Therefore, the first step in choosing micro frontends should follow the YAGNI-principle ("you ain't gonna need it" [59]). In other words, applying the micro frontends architecture pattern should be a necessity and there must be a real use-case for them (e.g., many or large teams, strict isolation of code, the requirement for different technologies, etc.). Singleton [85] mentions that microservices should not be used if less than 60 team members are working on a project. The same rule may also apply to micro frontends [60, 72].

2.3.1 Micro Frontend Composition

Given that an application should be developed in vertical teams, there is a need for re-combining the different artifacts back together to form a coherent application [58, 90, 89]. This is known as micro frontend composition. Geers [58] and Yang et al. [90] describe several ways of composing micro frontends, which we discuss in the following.

Route-based Composition

The simplest approach is hosting micro frontends on different paths or domains and only referencing them via regular anchor elements [58, 90]. However, Geers [58] concludes that this is not enough for most use cases. Only linking to other micro frontends prevents integrating other teams' fragments into different pages. For example, if we want to provide recommendations for a given product in a web shop, we have to link to the page of the recommendation team. According to Geers [58] and Yang et al. [90], this greatly reduces the user experience, since links are less interactive and may be overlooked.

```
<!-- product-page template -->
<div class="recommendations-container">
  <!--#include virtual="http://recommendations-domain/for-product/123" -->
</div>

```

Listing 2.1: Example showing SSI. The `#include` instruction tells the parser to fetch a resource referenced by the `virtual` parameter. Here, a dummy URL references the recommendations for the product with ID “123”.

Server Side Includes

The second solution described by Geers [58] is SSI. The idea is to add semantic comments to the page's markup to reference additional resources to load and include. A server then parses those comments, fetches the resource, and replaces the comment with the response's content before responding to the client's request [88]. The example Listing 2.1 shows a section of a product page to display recommendations for the current product. The `virtual` parameter points to the resource to load. In the given example, a dummy URL is passed to fetch the recommendations for the product with ID "123". As a result, according to Geers [58], developers can now integrate the UI of different teams into their applications. SSI also has a very low overhead in terms of client performance. Only a single request and response are needed for the composition. Since the server's response is a regular HTML file, there is no additional logic required on the client either.

However, SSI also has its drawbacks [58]. First and foremost, SSI does not provide means for isolating micro frontends. Since they are all part of the same Document Object Model (DOM) tree, they share the same styles and context, meaning teams need to carefully prefix their code with their namespace. Secondly, purely server-side rendered pages usually are less interactive than client-side rendered pages. For example, navigating to a product requires the client to reload the whole page. The server needs to re-render the entire page before sending it to the client. Hence, Geers [58] recommends combining SSI with client-side rendering techniques for optimal performance and responsiveness.

To counteract some of the disadvantages of server-side rendering, micro frontends can be composed on the client side [58]. This means that micro frontends are not rendered on the server, but rather generated via JavaScript on the client. As a result, pages can have more advanced features like dynamically showing and hiding elements, lazy loading content, and interacting with an API instead of re-rendering the full page, etc. [58].

The first client-side composition pattern Geers [58] describes is using HTTP calls from JavaScript to fetch HTML text and render it into the DOM. However, this approach still has many of the same drawbacks of SSI – most importantly lacking isolation. Moreover, the additional requests needed to fetch a page's content could lead to worse performance compared to SSI. [58]

```
<!-- product-page template -->
<div class="recommendations-container">
  <iframe src="http://recommendations-domain/for-product/123">
</div>

```

Listing 2.2: Example showing composition using `iframes`. By referencing the recommendation team's website, the product team can integrate product related recommendations.

`iframes`

To improve isolation between micro frontends, Geers [58] and Yang et al. [90] propose `iframes`. The largest benefit of `iframes` is their high degree of encapsulation. An application inside an `iframe` only has limited access to its parent and vice versa. As a result, styles and JavaScript code cannot affect and interfere with each other. However, Geers [58] mentions that `iframes` are hard to use in practice due to their lack of proper layouting. An `iframe` element will not adapt its height or width to its content. Hence, other means for resizing have to be found – usually via messaging, which will be described in detail in section 3.3. Additionally, every context the browser has to create costs performance [58, 90]. Finally, search engines will not treat a page containing `iframes` as a single page, but rather as individual pages [88, 58]. For those reasons Geers [58] sees the use case for `iframes` is limited to non-searchable sites that prefer the higher isolation to a higher integration. Geers [58] names the Spotify desktop app as an example where `iframes` make sense.

A simple example for client-side composition using `iframes` is shown in Listing 2.2. The code snippet shows the same context as the SSI example in Listing 2.1. But instead of server-side resolution of the micro frontend, the client will now render the recommendation team's website inside an `iframe`. For demonstration purposes, the URL of the `iframe` source is identical to the one of the SSI example. However, instead of only returning part of an HTML page, the recommendation team now needs to host and maintain a fully functional web page – including headers, styles, and code. Compared to SSI, this introduces an additional burden on the frontend teams.

Web Components

For a more modern approach, Web Components are a popular choice [58, 89, 88, 90, 75]. The Web Component specification consists of three parts: custom elements, shadow DOM, and HTML templates [82]. The first browser API allows declaring a custom element tag and its corresponding implementation. When the browser encounters the custom element's tag in the DOM, the implementation will be instantiated, which allows it to render the custom tag's content dynamically [82]. Hence, for the most part, it behaves similarly to built-in browser tags like the `video` element. The shadow DOM isolates part of the DOM from external styles and optionally even JavaScript access [82]. Using the `video` element as an example again, clients are not able to access or manipulate the

```
<!-- product-page template -->
<div class="recommendations-container">
  <script src="http://recommendations-domain/web-components">
  <recommendations product-id="123">
</div>

```

Listing 2.3: Example showing composition using Web Components. The product team references the recommendation team's micro frontend by using their Web Component in their template. However, the custom element needs to be registered first, which is done by executing the registration script.

browser-specific elements that are rendered as children of the `video` element. Finally, HTML templates allow reusing part of the DOM by cloning a template [82, 58].

Together, the Web Component specification allows the creation of reusable, isolated components that can be used like regular HTML elements [82, 88, 89, 90, 75]. Hence, they are technology-agnostic and can be used with any framework that allows the creation of elements via tag name (e.g., by calling `document.createElement`). This makes them a good candidate for client-side composition and a viable alternative to `iframes`.

The composition step can be found in Listing 2.3. We modified the example of the product page to use the recommendation Web Component. Since Web Components are defined using JavaScript, their consumers need to ensure that it is loaded before they can use them. Therefore, the code snippet uses a `script` element, which references the recommendation team's code. Next, the product page simply uses the Web Component like any other HTML element. Here, the Web Component receives the product's ID to show recommendations via an attribute.

2.3.2 Micro Frontend Communication

Geers [58] identifies three different communication directions a rich client application can use:

Parent to fragment communication (downwards). Downwards communication [58] is mostly used for passing parameters from the host to the hosted micro frontend. Using Web Components as an example, downward communication is possible by setting the custom element's attributes.

Fragment to parent communication (upwards). Upwards communication [58] is needed to notify a micro frontend's host about an event. For example, if a user clicked a button inside a micro frontend, the host may need to react to it.

Fragment to fragment communication (lateral). Lateral communication [58] is the most complex communication direction. It allows different micro frontend fragments

to communicate directly with each other. For example, if a product is added to the shopping cart, the "Add to cart"-button may become a "Remove from cart"-button. Since there might be more than one way of adding a product to the cart, all other buttons need to be notified about the change.

Implementations of Lateral Communication

To implement lateral communication, Geers [58] sees three possibilities. Firstly, direct communication can be achieved by looking for micro frontends in the current DOM and calling methods on them. However, this creates a very tight coupling between the sender and the receiver of the message [58]. Also, this approach is only possible, if all micro frontends have access to the full DOM. Hence, this solution is not viable when using `iframes`.

Secondly, lateral communication can be handled by routing messages via the parent(s) of a micro frontend. The message is emitted to the parent using the regular upwards communication channel [58]. The parent then needs to forward the message either to the receiver or pass it on to the next closest node. As a result, lateral communication becomes a path-finding problem, where each parent node must be capable of forwarding a message to either a sibling node of the sender or the parent if the receiver is no descendant of the current node. The complexity of the solution is therefore rather high.

Thirdly, Geers [58] recommends using a message bus that is agnostic to the micro frontend context. The global message bus can be accessed via all micro frontends and therefore lateral communication is as simple as emitting a new value to the message bus.

Signals

Nishizu et al. [75] introduced the concept of signals. A signal holds a value, which might change in the future. When the value changes, consumers of the signal are notified so that they can react to the changed value. This is therefore similar, how Svelte's change detection[1] and Angular signals[2] work. Therefore, those concepts can also be used in order to achieve lateral communication between micro frontends.

[1] https://svelte.dev/tutorial/reactive-assignments
[2] https://angular.io/guide/signals

CHAPTER 3

Evaluation Using Action Research

An important goal of this thesis is to found its results on a known basis. From there, new findings should build upon it, extending the boundaries of what is known. We thus avoid producing results, which are isolated to the specific field of research. Using the prototype methodology, for instance, an implementation phase is followed by extracting conclusions. Since the implementation is done in one phase, it may have blind spots due to the lack of reflection, degrading the quality of the results. Therefore, we apply the Action Research methodology. Action Research is a cyclical, iterative process of planning, acting and observing, and reflecting on the action taken as well as their effects [46, 66, 44, 1, 68, 51, 74, 52, 81, 86, 50, 62, 42]. Baskerville and Wood-Harper [46] identify three key characteristics of Action Research:

"(1) The researcher is actively involved, with expected benefit for both researcher and organization." [46] With the first point, Baskerville acknowledges that the researcher is part of the action research process and thus does not treat the researcher as an outside observer. By consciously reflecting on the researchers' actions the results do not claim to have a "false objectivity". Instead the path that lead to the results are an integral part of the research [46, 1, 51, 74, 52, 86, 50].

"(2) The knowledge obtained can be immediately applied. There is not the sense of the detached observer, but that of an active participant wishing to utilize any new knowledge based on an explicit, clear conceptual framework." [46] Action Research is an iterative process. Each iteration builds upon the results of the previous one, detecting new and existing weaknesses and additional goals. Hence, the research process is highly interactive. In comparison to a prototype where only the result is analysed, Action Research focuses on the process of creating the results too. [46, 1, 51, 74, 55, 52, 50]

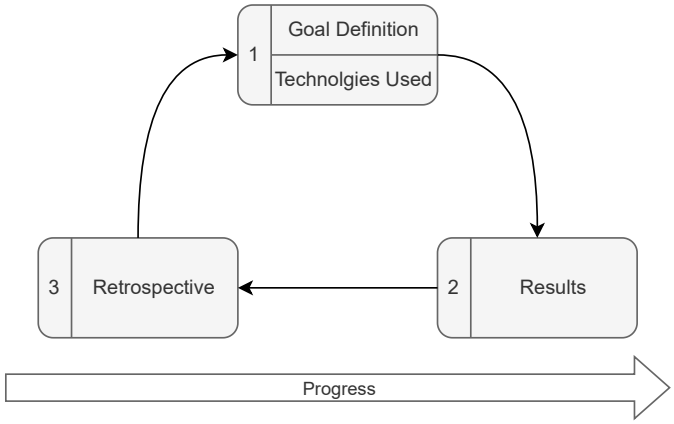


Figure 3.1: The Action Research Cycles of this thesis. Each iteration describes its goals and which technology should enable those goals, then describes the implemented results, and finally reviews the implementations. The last step will produce new issues and drawbacks, which will form the basis for the next iteration.

"(3) The research is a cyclical process linking theory and practice." [46] Each Action Research iteration follows a predefined cycle [69]. Usually, five phases describe the iteration process: *Diagnosing*, *action planning*, *action taking*, *evaluating*, and *specify learning* [46, 55, 52, 62]. However, a simplified version may only focus on the steps *problem diagnosis*, *action intervention*, and *reflective learning* [1, 42] – thus combining *diagnosing* and *action planning*, and *evaluating* and *specify learning*.

In this thesis, we will follow the shortened version, as it leads to less redundancy in the cycle step description. Each iteration, therefore, has the phases *Goal Definition* to describe the problem and its potential solutions, *Results* to describe the taken actions as well as the produced artifacts, and finally *Retrospective* to evaluate and reflect on the implemented solutions. Since this thesis focuses on the technologies that enable micro frontend communication and state synchronization, each cycle will include a chapter named *Technologies Used* to briefly describe the enabling APIs. Figure 3.1 visualizes this process.

Critics of the Action Research methodology mention that through the inclusion of the research the results may be biased [46, 81, 45, 62]. While some biases like the experience of the author are inevitable, we try to mitigate this issue by critically reflecting on the chosen implementations. However, due to the exploratory nature of this thesis, it is not possible to cover every possible solution available. Instead, we aim to provide a direction micro frontend communication and state synchronization can go. Future researchers may have the opportunity to compare more options than we can in order to gain a more holistic view and thus purely objective results.

Another issue of Action Research is it's context sensitivity [46, 81, 86, 62]. As the results are developed within a specific setting, they may not be reproducible by other researchers.

In order to circumvent this issue, we use a widely applicable domain without external factors or stakeholders. This means that our experiments are reproducible and verifiable. The full domain is specified in section 3.1.

3.1 Domain Definition

Before starting the Action Research, we define as well as argue the researched domain. First of all, the main requirement for the domain has to be its reusability, as the goal of this research is to find patterns and approaches that are universally valid. In order to achieve this, a generic setting with as few domain-specific characteristics must be used.

However, it is also important to choose a domain that applies to the real world, as the Action Research tries to find problems by investigating resolutions to issues that arise when working within the domain. Therefore the domain should not be purely academic. Instead of solving specific, predefined issues, the domain should be an open field so that found problems are as complete as possible. This avoids having blind spots in our research.

To fulfill those criteria, we chose the domain of a simple web shop for the following reasons:

- The domain is often met in the real world, meaning that its validity can be more easily verified.
- A web shop has many tasks that can be easily translated into other fields, e.g., displaying products on a full page, sending items to another component like the shopping cart, sharing state information about an item, etc.
- The domain has few domain-specifics like complex logic or interaction paths.
- The domain is complex enough to generate a wide range of issues when implementing micro frontends, e.g., cross-communication, managing navigation, communication with a backend, etc.
- The functionality of a web shop can be split into multiple parts which can then be encapsulated into micro frontends.

3.1.1 Requirements Analysis

With the domain in place, a rudimentary requirements analysis has to be done in order to define features and their scope. However, those features will focus on their usefulness for the Action Research. Therefore, it is more important to have technical variety instead of covering a wide range of user journeys or having good usability.

Following Leffingwell [67], we decided on using their “user-voice” format. User stories are thereby defined in the form “As a <role> I can <activity> so that <business value>” [67]. The first part of the user story focuses on the persona. This avoids defining features without having a use case for them. The second part describes what kind of feature is needed. This is a high-level description of an action that should be done by the persona. Finally, the business value must be stated to avoid having features that might be wanted by the users without having any meaningful value to the application [67].

We decided on this method due to its popularity in describing high-level requirements of an application. However, we do recognize that due to the technical nature of this thesis, the user-focused aspect of the user-voice format may not be leveraged to its full potential. Nevertheless, using a standardized way of finding requirements helps in comparing this study to similar works.

We then use the user stories to create basic wireframes of the application. In the implementation phases, we will use them as guidelines for the general structure of the different views the web shop must provide. Finally, we extract concrete functional and non-functional requirements from the user stories and designs. While those requirements may not cover every aspect of a real-world shop, they define and confine the scope of the Action Research.

3.1.2 User Stories

This chapter enlists all user stories we defined for the web shop application. Each user story is written following the user-voice format by Leffingwell [67]. Additionally, we provide background information on why each user story matters in the context of this research.

As a consumer
I can visit the home page to quickly get an overview of relevant or recommended products so that I don't have to manually search through product catalogs.

On the landing page of the web shop, we have some sort of recommendation system. The idea is to mimic the functionality of real-world shops. Usually, when visiting the root page of a shop, the user is provided with highlighted products and products related to the user's preference or past shopping behavior.

As a consumer
I can navigate to any product displayed in the recommendations so that I can read more information about that product.

While obvious from a consumer standpoint, from a technical perspective this user story describes the need to trigger navigation from inside the recommendation tile. Thinking of micro frontends, this requires the ability for "upward" communication. I.e., the component needs to notify its host that a product was clicked and the host needs to understand what to do next.

As a consumer
I can copy a link to a product's details so that I can easily share or bookmark it.

This user story describes a crucial part of micro frontends: The ability to share the current route ("state") across the full application. We want to investigate possible solutions for host-owned states (in this case the route) with child components.

As a consumer
I can complete part of the checkout process and come back to it later
so that I don't have to complete the full checkout immediately.

In order to achieve this, the application must be able to retain its state and re-use it later. It is therefore similar to the user story above, in that users can restore the page's state by navigating to said page.

As a consumer
I can view recommendations for a product on the product's detail view
so that I can view alternatives to the current product.

We chose this user story to highlight that it must be possible to re-use components (e.g., the recommendations) across multiple views and micro frontends.

As a consumer
I can add products to the shopping from the details page or the recommendations
so that I can buy it later when I am done with the current session.

Here, the user story describes a cross-cutting concern. Namely, adding products to the shopping cart at any time. This requires us to find communication paths that might not only be hierarchical, but lateral as well.

As a consumer
I can click on a button to view and manage the current content of my shopping cart so that I know which items I have already added and what they cost.

The goal is to have the shopping cart managed and shared as a state inside the web application. So instead of fetching the shopping cart from the backend with each click, the frontend should know what products have been added so far.

As a consumer
I can open the website on a different device and still view the same shopping cart
so that I'm not bound to do everything on one device.

Complementing the previous user story, we want to persist the shopping cart of a user into a database. This means that the shopping cart state of the frontend must be synchronized with the backend.

Figure 3.2: Example for recommendations on `https://web.archive.org/web/20230101002323/https://www.amazon.com/`. Visited on 2021-12-18.

As a developer
I can develop a new or existing feature without much obstruction, no matter the technology, so that I can improve the development speed and work satisfaction.

While usually not written down as a user story, developers require a good developer experience. This includes being fast at developing small features, easy debugging and useful error messages, transparent behavior of the application, quick and responsive developer tools, etc. In general, we do not want to sacrifice the developer's experience when developing micro frontends, hence, we highlight it by adding this user story.

3.1.3 Designs

With the user stories defined, we created rough designs as guidelines for the implementation phase. Since this master's thesis has a technical focus, we will only create basic wireframes. There won't be user studies or other ways of verification of the user experience or usability. However, having the wireframes at hand helps us to think about the different entry points for micro frontends.

First, the landing page of the app must include some form of recommendations. We decided on displaying different categories of recommendations in horizontally scrolling containers. This design was inspired by Amazon's[1] layout for related products as seen in Figure 3.2. Similarly, we wanted to present more products on the front page, so the landing page should have three rows of recommendations. The wireframe for this page can be seen at Figure 3.2.

A single recommendation tile must show the product's title image. When a user hovers over the product, more details of it should be visible, e.g., the name, description, and price of the product. The tile must include a "shopping cart" button. When pressed, the product is immediately added to the user's shopping cart. If the user clicks anywhere else on the product, the product details page is opened, which we describe later on.

Also visible in the wireframe in Figure 3.3 is the top bar of the app. It should be always visible, even if the user scrolls down. Most prominently, at the center is a search bar.

[1] Amazon.com Inc, https://web.archive.org/web/20230101002323/https://www.amazon.com/

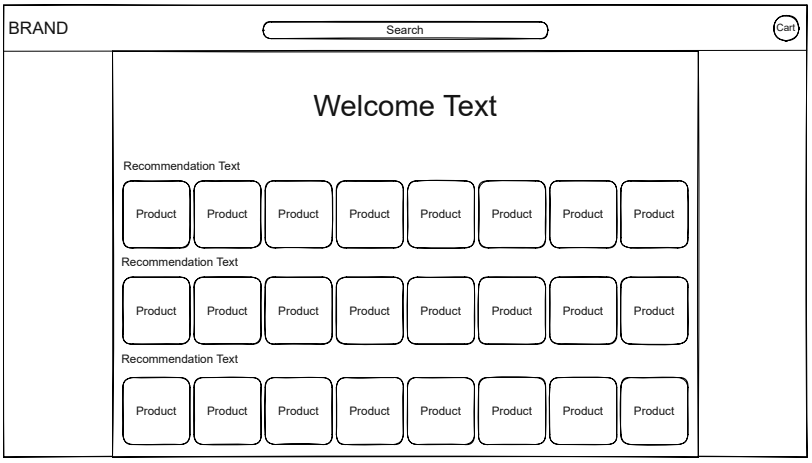


Figure 3.3: Wireframe for the front page of the web shop. The body of the page should list three rows of recommendations, which might need a horizontal scrollbar if too many products are shown. The top shows a header bar, containing a placeholder for the brand at the left, a search bar at the top, as well as a button to toggle the shopping cart at the right.

If the user enters text here, a dropdown shows products in a list with title images for each product. The user can click on a product to open the product's detail page. At the very right of the top bar is a button to toggle the shopping cart. The shopping cart should be located at the right of the screen as shown in Figure 3.4. The opened shopping cart displays the current shopping cart content as well as the total price. Each product is identified by its image and name and shows its price. Products must be visually separated in the shopping cart. While the shopping cart is open, the rest of the application should be de-emphasized. Clicking anywhere but inside the shopping cart should hide the shopping cart again – including the shopping cart toggle button.

If the user wants to view more details of a product, they can open a product details page – visualized in Figure 3.5. Here, all details of a product must be visible: the full description of the product, as well as all related images, and its price. We envisioned that the images should be displayed on the left in a sort of carousel. At the top, the user sees the currently highlighted image in a larger frame. Below are all images related to the product. Similar to the recommendations, if there are too many images, the container will scroll horizontally. Clicking on an image will show it in the larger view. Next to the image gallery is the product's description. In contrast to the recommendation tile, the description is not limited by size. There, a longer text with lists and custom formatted can be displayed here. Below each product, there should be a single recommendation row. This must use the same visual and logical recommendations as from the front page.

Finally, the checkout process in Figure 3.6 uses multiple steps shown on a single view. At the top of a card, a list of step numbers shows the current step, as well as the already taken and future steps. A step is displayed as a circle with its step number in it. Below

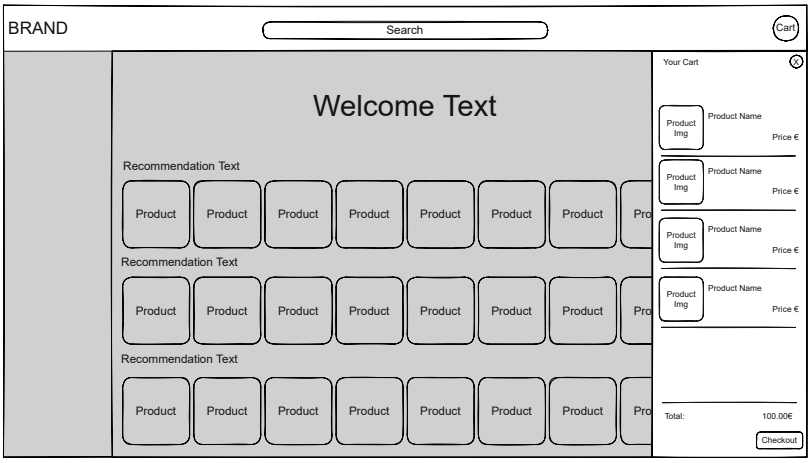


Figure 3.4: Wireframe for the opened shopping cart. All added products are shown here, including the total price for all products. The rest of the application is de-emphasized by being greyed out.

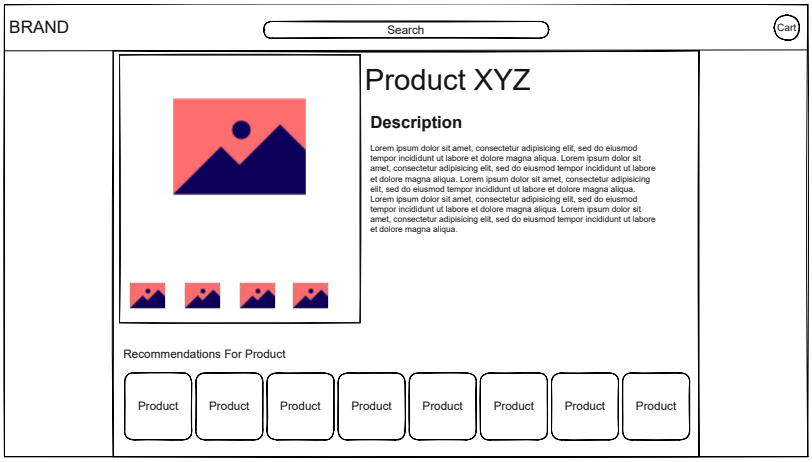


Figure 3.5: Wireframe for the detail page of a product. On the left, all images related to the product should be displayed. On the right, the product's full description is shown.

the circle is a short description telling what the user has to do to complete the current step.

Each checkout step has three states:

1. **The step is inactive.** This is the case if previous steps have not been done already and therefore the next steps cannot be performed yet. In Figure 3.6 this is the case for steps 3, 4, and 5.

2. **The step is active.** This means that this step is the currently viewed one. Only a single step can be active. In Figure 3.6, step 2 is currently active.

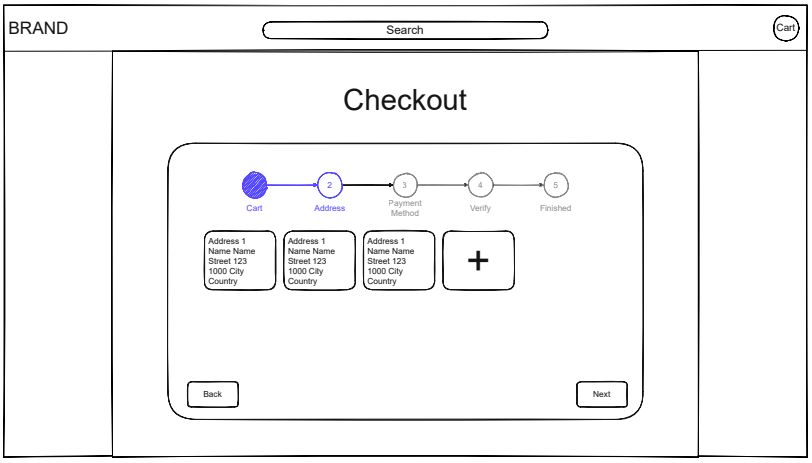


Figure 3.6: Wireframe for the checkout process displayed inside a card. Each step displays a different card content. Already completed steps (step 1), the currently active step (step 2), as well as not yet available steps (steps 3, 4, and 5) have to be highlighted in meaningful ways.

3. **The step is completed.** When the user moved on to the next step, the previously active step will be marked as completed. Completed steps can be re-visited to change selections and values. In the Figure 3.6, step 1 is already done.

To visualize the state of a step, clear highlighting has to be used. All inactive steps must be greyed out to visualize that they cannot be used yet. The active step should be highlighted in a light color. Already completed steps should be displayed using darker tones.

Every step will change the content of the card's body. The first step will display the shopping cart's content, while the second step displays the user's addresses, and so on. The steps content is defined as follows:

1. **Cart.** This step shows the shopping cart's content in a similar way the shopping cart does. It allows the user to review the shopping cart content before committing to the checkout process.

2. **Address.** The user can select or create an address to use for this transaction.

3. **Payment method.** The user's registered payment methods are listed as well as providing means for adding a new payment method.

4. **Verify.** A summary of the transaction and all items is shown.

5. **Finished.** The user is notified of the success or failure of the transaction.

3.1.4 Requirements & Limitations

In order to focus on the development process, a strict scope must limit the features provided by the web shop frontend and backend. We derive our requirements from the user stories and wireframes, extended with non-functional and technical topics.

The web shop must meet the following requirements:

- A web shop should be implemented to buy products, the kind of products is irrelevant. For prototyping, the action of buying a product and the transaction must be simulated.
- The web shop must have a landing page to display some products with some but not all information. The selection of products can be random.
- The navigation bar at the top of the page must be visible on all pages except for the checkout page.
- Clicking the shopping cart in the navigation bar opens a sidebar that overlays the content.
- A product's detail page should contain a description, a gallery of images, and related products. All of them can contain dummy values. However, the related products must be different for each product. I.e., the recommendations on the product details page must be parameterized.
- The user's ID can be generated client-side. This simulates having a logged-in session. The user ID should therefore persist across reloads.
- Every product, both in recommendations or on product details, can be added to and removed from the shopping cart.
- A shopping cart is attached to a user's ID.
- Adding and removing a product from the shopping cart must trigger a request to the backend. The cart must therefore not be stored purely in the browser.
- The shopping cart must be globally stored on the frontend. Meaning that all components must be notified when a product is added or removed from the cart.
- There must be a checkout process for a shopping cart.
- The checkout process must include multiple steps:
  - Display the current shopping cart content using the global state.
  - Display a list of addresses as well as the possibility to add more.
  - Display a list of payment options as well as the possibility to add more.

  - A final step to confirm the user's selection.
- The backend must provide APIs for managing a user's addresses and payment options.

As for the limitations, the web shop should not include any of the following:

- The recommendations can be mocked by fetching a random assortment of products from the database. No actual recommendation engine or user tracking should be implemented.
- The checkout process must not include any actual payments.
- After the checkout process has been completed successfully, no further interaction or features may be provided. The scope of the user journey is limited to a single visit.
- Server Side Rendering (SSR) of SPAs is not covered by this research.
- Basic styling of the web shop application is sufficient. Responsiveness of the application is not required.
- No usability study or other user-focused research will be performed. The evaluation is done solely on a technical basis.

While this scope definition is insufficient for a real-world web shop application, it serves as a boundary for this research. That way we can focus on relevant parts of the application rather than starting to research beyond the intended scope of this thesis.

3.2 Iteration 1: Implementation as Monolith

In order to build iteratively upon an application, we first need to implement a baseline for future iterations of the Action Research. This chapter, therefore, focuses on implementing a monolithic SPA and a backend infrastructure.

The frontend monolith is a simple web shop. Users can see recommendations, open details for individual products, add products to a shopping cart, and finally, complete the order in a "checkout" process.

To mock the functionality of a real shop, a backend is implemented providing a basic API for fetching recommendations, individual products, managing products to a shopping cart, as well as faked user data for the checkout process. For this purpose, product data is randomly generated and persisted in a database. This ensures that even while developing, usable, reproducible data is available.

3.2.1 Goal Definition

The first iteration has the ambitious but necessary goal of implementing the full frontend monolith. Therefore, the goal definition overlaps heavily with the domain definition and following requirements analysis of subsection 3.1.1 and subsection 3.1.4.

While the implementation of the monolith could have been split into multiple iterations of the Action Research, we argue that the monolith bears little information and gained knowledge concerning micro frontends. For this reason, we decided to describe the monolithic implementation in a single iteration.

Technologies Used

There are many web frameworks for building monolithic SPAs. For example, Angular, React, Vue, and Svelte are commonly used today. Since the focus of this research lies on communication and state synchronization, choosing one web framework over another must not matter. The found patterns must be technology agnostic and not use framework-specific features (such as dependency injection using Angular or the context API in React applications).

Due to our personal experience with Angular, we decided to implement the monolith – and hence future iterations – with an Angular base. Angular is a web framework for building modularized monoliths. This means that different, reusable frontend components are grouped into modules. Modules can be imported by other modules to reuse their defined components and services. This modularization of the monolith already helps us to define slices that can be extracted into micro frontends, e.g., it is possible to have a module for the checkout process in the monolith that can be extracted to a separate micro frontend.

Furthermore, we also need to decide on the technology for the backend. Since it is purely needed for implementing the features the web shop provides and is of little relevance to

the results of this research, its technology does not affect the results. Nevertheless, there are some considerations we had while deciding on the concrete implementation. Firstly, we wanted the backend to be extensible such that it is easily possible to add features needed by future micro frontend implementations (e.g., serving JavaScript files, adding a micro frontend registry, etc.). Secondly, adding features should have little to no impact on existing code. That allows us to remove features, once they are not needed for one of the following iterations, without extensive refactoring or regressions. Thirdly, the setup should be simple enough, such that it is understandable by an interested, educated reader. In other words, we do not want to rely on heavy, complicated frameworks. They might help in creating apps but are hard to understand for people unfamiliar with them. Since the goal of this master thesis is to research micro frontends, we hence saw no benefit in obscuring our backend implementation.

For those reasons, we decided to use NodeJS with the Express library for managing the REST endpoints. Moreover, to achieve isolation of the different features, we implement the backend using a microservice architecture. This allows us to write further applications and add them to the system. Additionally, they can also just as easily be removed from the system without any traces left behind.[2] For communication between the microservices, simple HTTP requests are used. While there are solutions available to simplify this process, we prefer understandability over ease of development here.

Finally, to simplify styling the frontend, we decided to use Bootstrap. Once again, this does not affect the research, as we only use the style sheet of Bootstrap, not its JavaScript components.

3.2.2 Results

Implementing the monolith required us to complete many different aspects of a SPA. As such, this iteration is the largest in terms of code written and features implemented. To provide an overview of the important aspects of the Angular monolith, we provide a graphic showing the most important Angular services and components in Figure 3.7.

In the diagram, the backend is treated as a closed box – its implementation is irrelevant. Instead, only the different APIs and how the frontend interacts with them are important. Below the backend, the Angular monolith is shown. We implemented the monolith using two layers: One providing the logic and one rendering the views. This allows us to have a clear separation of concerns. Therefore, splitting the monolith into multiple micro frontends will be easier later on.

The logic layer is composed of multiple Angular services, shown in blue in the diagram. Services with little cylinders at the top right corner represent “stateful services”. They contain data that must be shared across components.

[2] As it turned out, it was not necessary to adapt the backend for the different micro frontend implementations. So future iterations will always use the backend as described here.

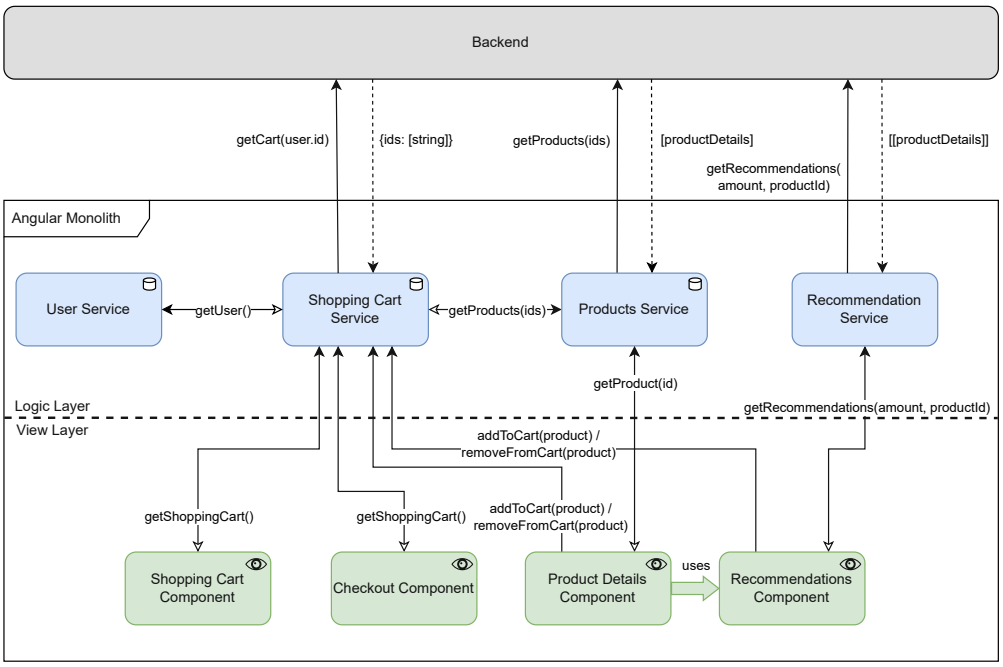


Figure 3.7: Architecture of the Angular monolith. The application is split into a two-tier system: The view layer and the logic layer. The view layer mainly handles rendering data and reacting to user input. The logic layer handles data management and backend interaction. Arrows show the communication between the different parts. Lines with single arrows show requests, and dotted lines the responses. Lines with two arrows show data streams. The filled arrow shows the direction of the request, hollow arrows show where the data is pushed to.

In the monolith, there are three stateful services:

1. `UserService`, providing the current user as a data stream (a user might log out and back in again).
2. `ShoppingCartService`, handling shopping cart interactions.
3. `ProductService`, which holds products that have already been fetched from the backend. It serves as a client-side cache.

As shown in the diagram, those services have to work together to provide the data needed by the views. For example, the `ShoppingCartService` needs to combine multiple data streams into one. First, it needs to get the current user. This is shown via the filled arrow: The `ShoppingCartService` requests the data from the `UserService`. The `UserService` returns a data stream in case the current user changes during the session. Returned data streams are marked via hollow arrows in the diagram. This visualization was chosen because data streams "push" data toward the caller. However, they need to

be differentiated from sending a request, which is why we decided on using hollow arrows instead.

The `ShoppingCartService` then sends the current user's ID to the backend in order to fetch the shopping cart. However, the shopping cart from the backend only includes product IDs. It, therefore, has to forward the product IDs to the `ProductService` to fetch the corresponding products. When one or more products have already been fetched, they are not loaded again. Only the missing products are fetched from the backend. Regardless, the corresponding products for the passed product IDs are provided as a data stream. Now the `ShoppingCartService` can provide the full shopping cart data – for the current user, containing the product details – to the views.

Apart from the stateful services, there are also stateless services, e.g., the `RecommendationsService` only returns a set of recommendations for a given number of recommendation groups ("amount" in the diagram) and product ID. No data is held beyond that interaction. Both kinds of services have to work as expected across all micro frontend implementations. In particular, stateful services have to share their state across all micro frontend instances. Hence, those services play a crucial role in our further research and are the focus when finding ways for state synchronization.

Following the services, we highlighted some components in the architecture diagram in Figure 3.7. These components are views – marked with an eye symbol at the top right corner. A view's task is to take data as input, and present it to the user via rendering HTML and Cascading Style Sheet (CSS), as well as handling user input – usually by calling another service, e.g., the shopping cart component receives the shopping cart as data. It then renders HTML to show the shopping cart's content. If a user wants to remove an item from the shopping cart, a button may be clicked. The view calls the `ShoppingCartService` to remove the clicked product. This triggers a data change, which is emitted to the shopping cart component again.

Apart from rendering, there is as little logic as possible inside views. As already mentioned, the separation of business- and rendering logic allows us to easily refactor the views without having to change the data aggregation. In our case, we will be able to easily replace components with micro frontends in following iterations.

While there are more services (e.g., for the checkout, the search, etc.) and components (e.g., footer, header, search), they play a smaller role in this research. For this reason, they have been left out to improve the clarity of the diagram.

Since the monolithic application is reused in future iterations, it is important to understand the current pages in detail. In particular, we discuss the following:

1. The landing page of the application. It is the main entry point of a user and displays recommendations.
2. A product details page to display further details of a single product. At the bottom, a row of recommendations is included.

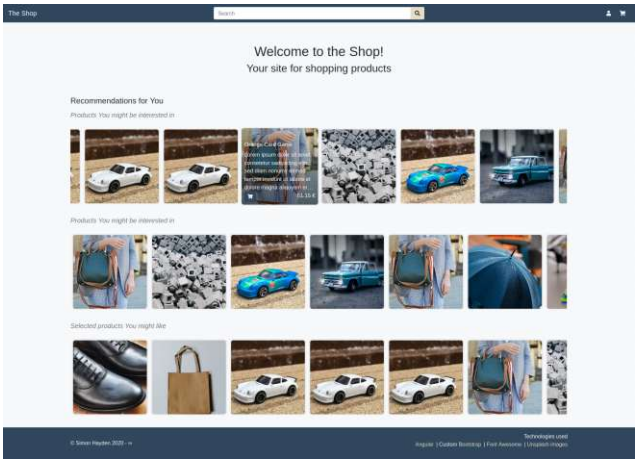


Figure 3.8: Screenshot of the monolithic landing page. It shows a welcome text at the top, followed by three groups of recommendations, each showing a heading provided by the backend. The individual groups can be scrolled horizontally as shown in the top group. Here, the middle product is hovered over by the user. Therefore, its name "Orange Card Game", a dummy description, as well as its price and button to add it to the shopping cart are visible.

3. The checkout process is a stepped process to complete the purchase.

Landing Page

Starting with the landing page, we had to write the recommendations component. This component takes the number of groups to load and display as well as a product ID as parameters. Each recommendation group displays its recommendations in a horizontally scrolling list of products. Each product is visualized using its title image. When a user hovers over a product, the product's name, a dummy description, its price, as well as a shopping cart button is shown. Clicking the shopping cart button adds the corresponding product to the shopping cart. Clicking anywhere else on the product card opens the product details page. A screenshot of the full landing page can be seen in Figure 3.8.

Looking at the component tree in Figure 3.9, the landing page is composed of the `app-landing-page` component, which nests the `app-recommendations` component. The outer `app-landing-page` component's task is to pass the proper parameters to the nested component in order to properly display three rows of recommendations. The `app-recommendations` parses the parameters and requests the recommendations from the `RecommendationsService`. The component then listens to the returned data stream and renders the emitted data.

With regards to micro frontends, this view should cover the "base" case: A hosting application integrates a micro frontend. There is no state to be shared between the host or the micro frontend, since we implemented the recommendation service stateless – as

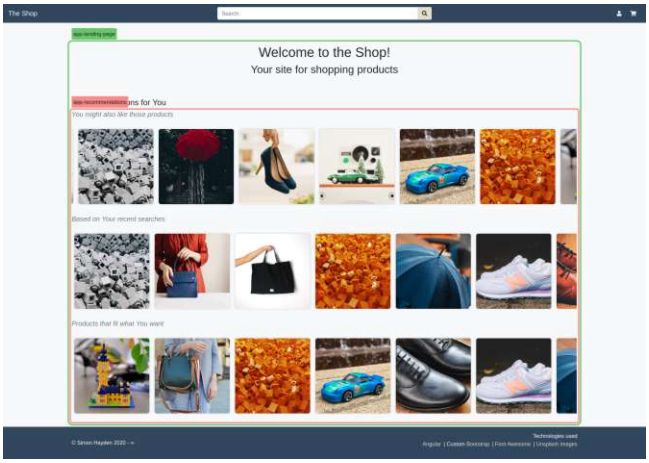


Figure 3.9: Angular components of the monolithic landing page. The view's content is the `app-landing-page` hosting the `app-recommendations` component.

mentioned above. Therefore, the hosting application only needs to pass the parameters (e.g., number of recommendation groups) to the micro frontend, and the micro frontend only has to emit events (e.g., a recommendation is clicked; a product should be added to the shopping cart).

Product Details

The next view to discuss is the product details page. Its task is to display all information related to a product. In addition to its name, price, and shopping cart button, the product's description and image gallery are shown. We also wanted to show recommendations for a product. This means that a single row at the bottom of the page must be displayed. For this use case, we re-used the recommendations component of the landing page.

As seen in Figure 3.10, we decided to display a dummy description, as to not have a mostly blank page. This description is hard-coded and does not change depending on the product. For simplicity, no actual data is provided by the backend for the description of a product. Regarding the component structure, we have highlighted the important Angular components in Figure 3.11. The root component is now `app-product-details`, which hosts the same `app-recommendations` component we have already used for the landing page.

With the outlook towards refactoring the monolith to using micro frontends, this page helps us in researching how to handle nested micro frontends. Recommendations must not only work as part of the hosting application but also inside another micro frontend. Therefore, the recommendations must not be coupled to the hosting context.

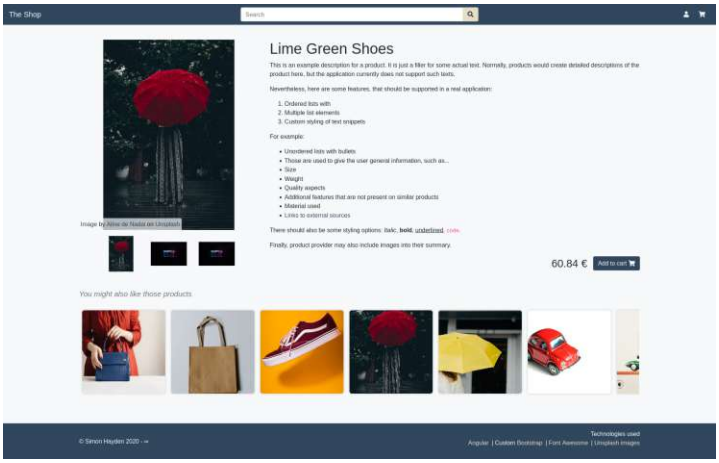


Figure 3.10: Screenshot of the monolithic product details page. On the left, an image gallery shows all images related to a product. Below is a list of recommendations for the selected product. At the center of the page are the product's name and a dummy description.

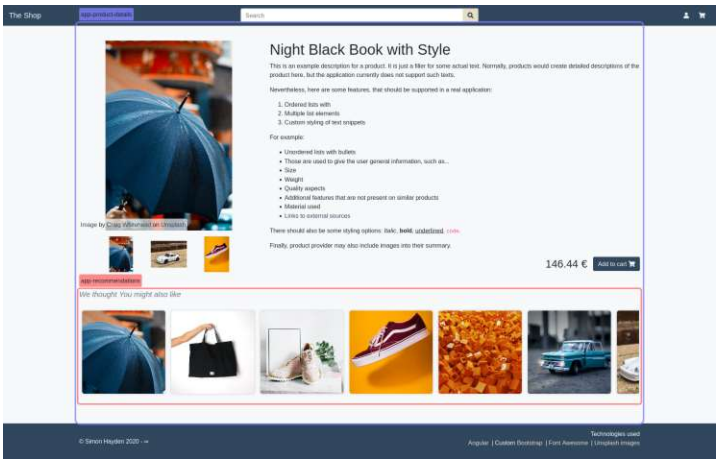


Figure 3.11: Angular components of the monolithic product details page. The `app-product-details` component is used as the root component to render product details. The `app-recommendations` component is re-used to display the recommendations for a product.

Checkout

The final page is the checkout process. It is implemented using a single component, containing multiple tabs. Each tab is displayed via a circle with a step number inside. The user can click arbitrary steps, as long as they are active. A step is active if all previous steps have been completed. A screenshot of our implementation is provided in Figure 3.12. Additionally, each step is individually shown in Figure 3.13.

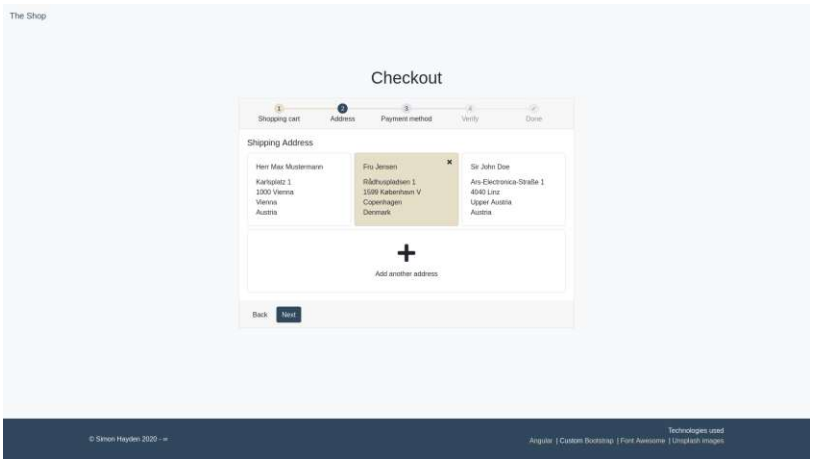


Figure 3.12: Screenshot of the monolithic checkout page. In the center of the page is a card, whose content changes based on the current step. Currently, the step "Address" is active.

Since the previous views all retain their state via services or components, we decided to use the browser's URL as state storage. For example, the currently active step is not stored via a variable inside the component, but rather by reading and updating a query parameter of the current URL. This also holds for the selected shipping address and payment method. This means that the checkout component is completely stateless. It emits events when the form data changes and receives updated parameters afterward. This should provide a unique challenge for the micro frontend implementations. We need to solve the problem of how micro frontends access and manipulate the URL.

Finally, the component structure of the checkout page – seen in Figure 3.14 – is one of the simplest yet. As mentioned above, we implemented the checkout process as a single component. Hence only one Angular component is shown in the screenshot. Once again, with this view we want to focus on URL synchronization, rather than creating complex nested components, which might be extracted to web components later.

Backend

Since our backend bears little informational value towards the research of communication and state synchronization of micro frontends, it will not be discussed in detail here. However, we have added a chapter to the appendix describing the architecture.

Regarding this thesis, the backend can be viewed as a closed box. It provides the APIs needed to fetch products, manage shopping carts, as well as addresses and payment methods for the checkout process.

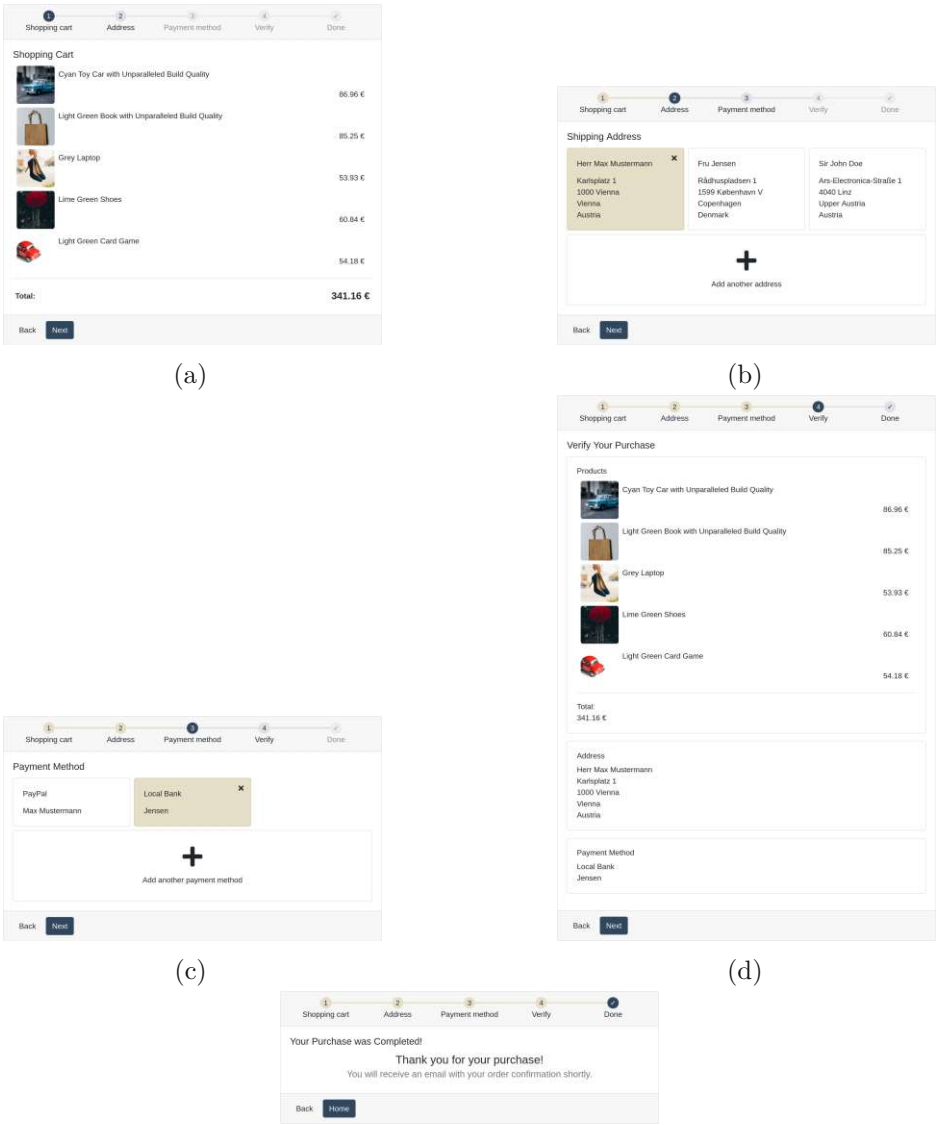


Figure 3.13: Screenshots of the individual steps of the checkout page. The process of the checkout process can be viewed via the stepper on the top of the card. The next step is only available if the current step has been completed (visualized by fading the step as long as it cannot be selected).

3.2.3 Retrospective

Implementing the baseline for the following iterations meant developing a wide range of aspects of the application. Especially the infrastructure was a challenge since we had to provide useful workflows for both the frontend and the backend. For instance, the

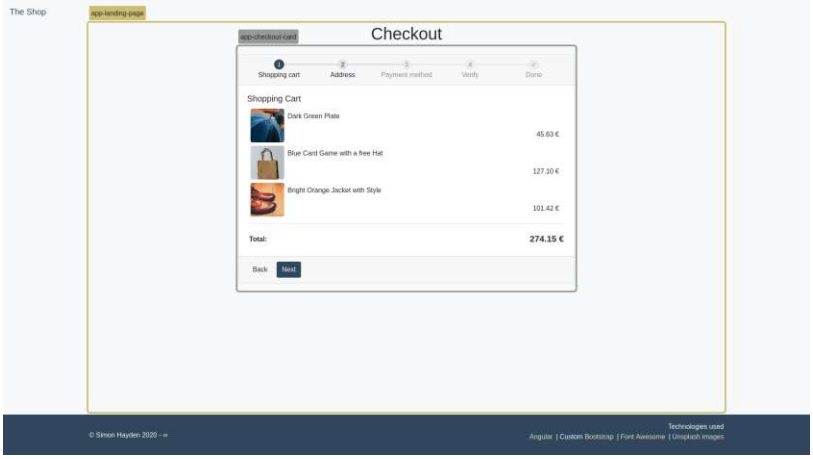


Figure 3.14: Angular components of the monolithic checkout page. The `app-landing-page` should not be confused with the component mentioned in Figure 3.9. The unfortunate naming came to be, because this component represents the landing page of the checkout process.

backend had to be usable while being developed, as well as when it is deployed as Docker containers. This meant having workflows in place on how to start a microservice locally, then building and deploying it into the Docker network once the changes have been applied. However, our established backend infrastructure is a robust foundation that the other iterations can build upon.

As for the implementation of the frontend, we implemented a generic Angular monolith (i.e., without extensive library usage, complex state management, etc.). This was done intentionally, as to not generate library- or framework-specific results. Consequently, the development process for the frontend mostly followed the official Angular workflow: Using the `ng` command to generate a new application, scaffold components and modules, serve the application with a development server, and build the application. Therefore, there have been no surprises during the development of the different components and modules.

Most noteworthy, we implemented the state management via global Angular services and data streams in this iteration. Those data streams need to be provided in micro frontends as well. However, for micro frontends we cannot use Angular's dependency injection of service ecosystem. Hence we anticipate that converting the current services to framework-agonistic classes is one of the main focuses in the next iterations.

3.3 Iteration 2: Migration to iframes and Implementing Direct communication

In the first micro frontend iteration of this Action Research, we split the application into micro frontends using iframes. We chose this approach first, because iframes are a well established and understood technology. They have been supported since 1997 and were standardized in HTML 4.0 [15]. Even though, iframes have not been intended to be used as micro frontends, however, they fit their requirements quite well. In particular, iframes provide a high degree of isolation, which means that applications can be developed and deployed independently. Implementing a micro frontend approach with iframes first provides the opportunity to find shortcomings early and compare how more modern technologies overcome them.

3.3.1 Goal Definition

In this iteration, we implement a micro frontend based application via iframes. In contrast to the previous iteration – the monolith – we are not focusing on developing working software yet. Instead, we want to achieve basic integration of the micro frontends into a hosting application. This also means that we will be adding one-to-one communication to the iframes.

Therefore, the goal definition for the second Action Research iteration is:

1. Decide on feasible cut-off points of the monolith.
2. Migrate the monolith's modules to separate Angular applications.
3. Integrate the newly created Angular application into hosting applications.
4. Implement direct, one-to-one communication between the host and the micro frontend.

Hence, the expected result of this iteration is a web application, which allows navigating between views, passing parameters toward micro frontend as well as receiving messages from the micro frontend at the host.

Technologies Used

The advantage of high isolation provided by the iframes from the hosting application is a disadvantage regarding the possible communication channels and patterns. Since iframes do not share the same context with their parent, there is only a limited amount of APIs the host is allowed to call on the inner content (and vice versa). There are some relaxations when the application inside the iframe is on the same origin (e.g., accessing the same localstorage, sharing SharedWorkers as service layer, using BroadcastChannels, etc.), but since this assumption could also highly hinder the

```
// Code on the receiving end.
window.addEventListener("message", (event) => {
  // The event.data contains the payload of the sender
  console.log(event.data);
});

```

(a)

```
// Code for the sender.
// In this example the sender is an iframe, therefore it sends messages to
// the hosting application. But any window reference (e.g., from pop-ups)
// can be used to send and receive messages to and from a foreign browser
// context
window.parent.postMessage("hi from iframe");

```

(b)

Listing 3.1: Example usage of the `postMessage` API. The receiver (e.g., the hosting window) must first register an event listener for the `message` event on its own window object. The sender can then get a reference to the receiver's window object e.g., via the parent field – it references the hosting window of an `iframe`. When the `postMessage` method is called, a new "message" event will be triggered in the receiver. Hence, the registered event listener is called.

usefulness of implemented patterns (e.g., not being able to embed external micro frontends into the host; allowing different teams to be on different domains), they have not been taken into consideration.

The main intended communication method between `iframes` and their hosts provided by browsers is `postMessage`[3]. It enables different browser contexts to communicate with each other via an event based, asynchronous communication channel. Arbitrary, serializable data can be sent via this function [16]. In the short example shown in Listing 3.1, the receiving end has to register an event listener for the `message` event on the global window reference. Whenever another instance wants to send a message to the receiver, it needs to get a reference to the receiver's window. Here it is assumed that the sender is inside an `iframe` of the receiver, therefore using the global `window.parent` to get a (potentially restricted) reference to the hosting application's `window` object. However, any other way of getting a window reference is valid (e.g, by opening a pop-up window or new browser tab).

The next step is sending a message. Calling `postMessage` on a window will generate a new event on the referenced window object – the `iframe` host in the example in Listing 3.1. This sent data can be arbitrary, as long as it is serializable. In particular,

[3] `https://web.archive.org/web/20230406194208/https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage`

the sender cannot pass function references (and by extension also class references). This means that senders cannot provide callback functions to a sent message directly. In the example above (Listing 3.1), only a string is passed. As a result, the receiving browser context generates a new "message" event and the registered listeners are called. The passed event will contain the sent custom payload in the `event.data` field. We will use this communication channel for pushing updates for parameters (inputs) to a micro frontend, as well as receiving events (outputs) from them.

Another way of passing arguments towards an `iframe` is via its source URL. Just like for any web application, query parameters can be used to send arbitrary, serializable data. However, updating the query parameters from outside requires changing the source attribute, hence reloading the application inside. Therefore, the `iframe`'s source does not fit our requirements. Nevertheless, this approach is still used to pass the initial parameters to an `iframe`. Further updates are sent via the `postMessage` API described above.

3.3.2 Results

One of the challenges of this iteration was deciding on how to split the monolith. In theory, every component of the Angular monolith could be split into its application run inside an `iframe`. However, the organizational overhead, as well as the performance overhead, could have slowed the development process by a lot. For this reason, we decided to use larger applications around business cases instead. The following `iframe` based micro frontends were developed:

- Micro frontend for showing recommendations.
- Micro frontend for showing product details.
- Micro frontend for handling the checkout process.

The resulting architecture can be seen in Figure 3.15. The views for the checkout process, product details, and recommendations have been rewritten to embed the corresponding Angular applications. As a result, the Angular app shell no longer needs to contain services for those views. Instead, the embedded applications have to provide those services themselves. For example, in the monolithic Angular application, the product details view had to fetch the product details from the product service. With the micro frontend architecture, however, the logic for getting data is transferred to the micro frontend itself. The product details component of the app shell only passes the arguments required by the micro frontend – in this case, it forwards the product ID that should be displayed.

Since each Angular application runs in its own browser context, each micro frontend hosts its Angular services. As a result, the current implementation does not provide a shared state – as mentioned in the goal definition.

In the diagram visible at Figure 3.15, this means the product services cannot share the same cache of products. Even though they share the same code, they cannot access the

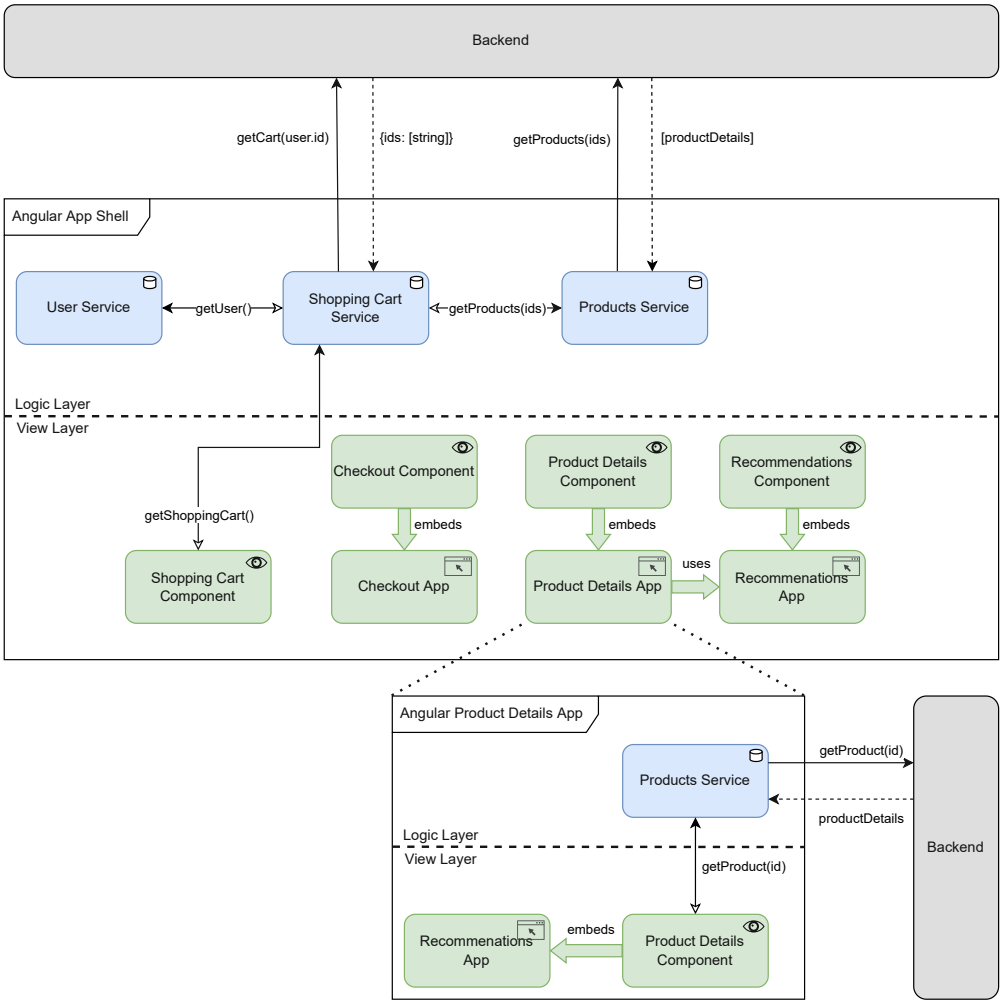


Figure 3.15: The micro frontend architecture using `iframes`. The shopping cart component was kept as a regular Angular component. The other views embed the corresponding applications (highlighted via the browser symbol on the top right corner). Each application consists of the component and all required services required for it. Note that the two backends may not be different instances, but are separated to improve the layout.

runtime data from each other. Therefore, each micro fronted needs to fetch the product details for recommendations, the shopping cart, or the product details page on their own.

The monolithic implementation for the user service relies on generating a random ID and storing it in the local storage of the browser. This simulates a unique, logged-in user for the application. Since the local storage is bound to the host address of the web application, it cannot be shared across iframes. For this reason, the product details app in diagram Figure 3.15 does not host its own instance of the shopping cart service. If it was, adding a product to the shopping cart would only add the product to the local user. Other micro frontends, as well as the app shell, have different users and hence do not see the added product. As a result, whenever the shopping cart button is clicked, the message must be forwarded to the app shell in order to toggle a product in the shopping cart. Nested micro frontends complicate this even further, as they need to "bubble" messages from their child micro frontend to their own host. Hence, the amount of different messages a micro frontend emits is always a superset of the messages that all child micro frontends emit. This creates a very strong coupling between the micro frontend and its children.

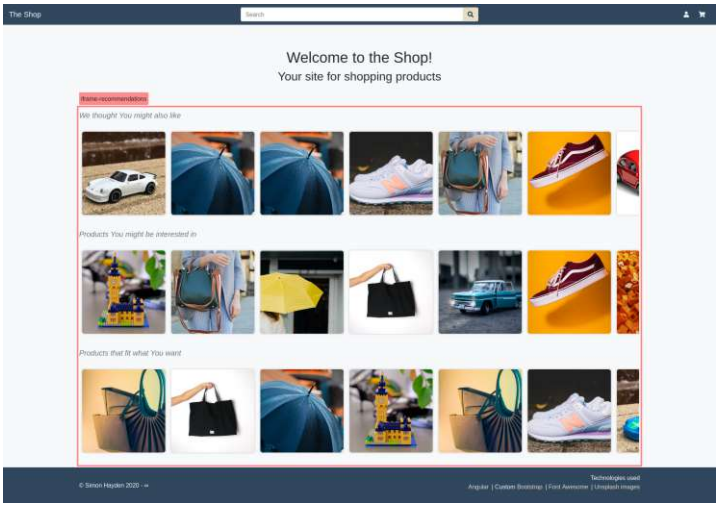
Because the checkout process highly relies on state management, we decided against implementing it via one-to-one communication alone. Instead, it will be implemented with the next iteration. More details regarding the communication between micro frontends will be discussed in section 3.3.2. First, however, we will demonstrate how the migration to iframes affected the UI. This also helps to understand which fragments need to communicate with each other and their hierarchy.

Views

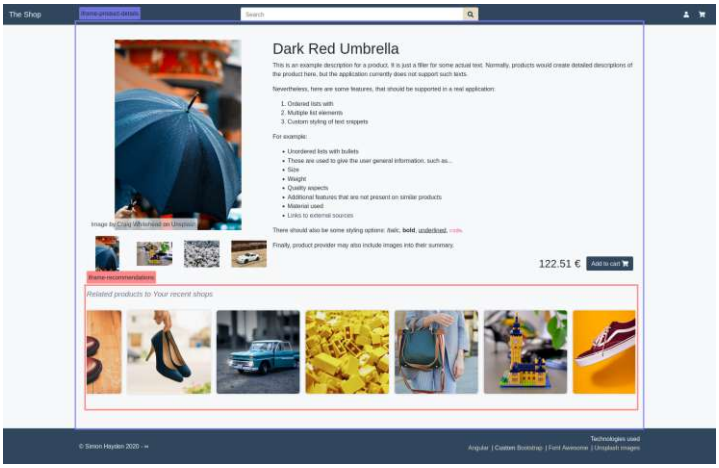
As already mentioned, for this implementation we decided to form coarse micro frontends. Due to the development overhead for each iframe, one micro frontend for sharing buttons or image components would have been too complex without yielding additional information regarding this research – the communication and state synchronization patterns for micro frontends.

To highlight the different micro frontends in each view, we displayed borders around each iframe containing a micro frontend in Figure 3.16. Each view – the landing page, the product details page, as well as the checkout page – has been migrated to micro frontends. The landing and checkout page only host a single micro frontend each – the recommendations and checkout micro frontend respectively. Their only parent is the app shell. In contrast, the product page uses nested micro frontends. As seen in Figure 3.16b, the innermost micro frontend is the recommendations. Next up in the hierarchy is the product detail micro frontend, which itself is hosted in the app shell.

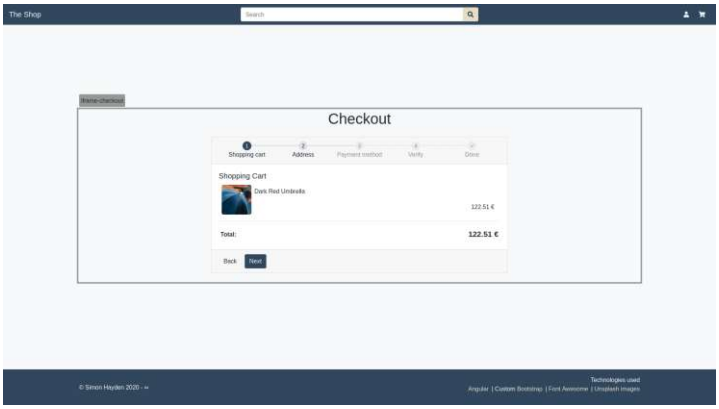
However, since iframes do not grow with their content, the dimensions of each iframe must either be defined by its host or emitted from the iframe and manually set. For our purposes, each micro frontend will send its current content height via the postMessage-channel to the micro frontend host, which sets the iframe's height to the emitted



(a)



(b)



(c)

Figure 3.16: Screenshots of the three micro frontends inside `iframes`. The landing page (a) hosts the recommendations micro frontend. In (b) the product details micro frontend nests recommendations. In (c), the checkout micro frontend is centered inside the host.

value. Figure 3.16 shows that, with this inconvenience in mind, the migrated application looks very similar to the monolithic implementation. However, this may not apply to all applications using iframes as micro frontends. For instance, some layouts use an HTML element's initial size as a baseline but will shrink it down to fit the container. This is how the CSS flexbox works [14]. The micro frontends, therefore, need a sensitive balance between emitting their intrinsic size as well as using the actual iframe's size. If the balance is violated, the iframe may begin to scroll or the layout of the webpage keeps changing based on the last emitted event – either the iframe's actual size or the iframe content's size.

Communication Between Micro Frontends

After splitting the monolith into iframes, we had to re-implement the communication channels of the application. For example, without knowing which product to display, the product details micro frontend cannot work properly. Therefore, the communication patterns mentioned in section 3.3.1 were applied to pass inputs and outputs between parent and child. The implementation is inspired by Angular's `@Input()` and `@Output()`. Parameters are passed from parent to child (one-way binding) and events are emitted from the child to the parent. There is no two-way binding (changes to input parameters are also reflected to the component host) or data sharing between parents and children. For our communication channel, we also want to push updated parameters towards a micro fronted, while emitting updates to the micro frontend host.

The first step in achieving this was the creation of the intercom Angular library, which abstracts the lower level details of message passing. Instead of dealing with sending and receiving messages via postMessage and message-events directly, consumers of the intercom library can pass and process message objects. The library then handles pushing the message to the correct receivers as well as creating data streams for incoming messages. The data streams are created by wrapping native JavaScript events of the message channel. While the payload of the postMessage API is untyped, we wrote some interfaces that all messages must follow. Most importantly, each message sent has to declare a `"type"` property (e.g., `"input-changed"`, `"output"`). It can be used by the type system to infer what kind of properties exist on the message. However, the dynamic payload, specific to each micro frontend, could not be typed statically. The library must still allow arbitrary data to be sent.

To have a better separation of concerns, we decided to provide two Angular services for communication. One service was created for micro frontend hosts (ParentIntercom Service) and one for hosted child micro frontends (ChildIntercomService). The first one allows callers to send messages directly to an iframe (and therefore a micro frontend) and listen for messages from them (e.g., output events). The second one simplifies receiving messages from the hosting micro frontend (e.g., updated inputs) and sending messages to it.

```
<app-iframe-host
  [app]="'mf-name'"
  [input]="{productId: 3}"
  (output)="onOutput($event)"
></app-iframe-host>

```

Listing 3.2: Example usage of the `iframe`-wrapper Angular component. The hosted micro frontend is passed via the "app" parameter. To pass parameters to the hosted micro frontend, the "input" parameter is used. Events from the `iframe` are re-emitted using the "output" event.

In order to further simplify handling `iframes` for hosting micro frontends, we wrote the Angular component `IframeHostComponent`. It enables using micro frontends as closely to regular Angular components as possible. It defines an Angular input, simply called `"input"` to send and update parameters to the micro frontend and an Angular output aptly called `output` to receive messages from the `iframe` as native Angular outputs. However, herein also lies the first drawback of this setup. Since the `IframeHostComponent` component cannot know what kind of parameters or events the inner `iframe` may or may not consume or emit (especially their structure), all inputs and outputs of the component are untyped. As a result, type checks have to be done at runtime by both the micro frontend and the micro frontend host. The `input` of the `IframeHostComponent` works in two ways. When the iframe is not loaded yet, the input must be visible to the `iframe` immediately. This means that the input is passed to the `iframe` via its source's query parameters. All values are `JSON`-encoded to enable more complex data to be passed. Afterward, any further updates to the `input` cannot re-write the `iframe`'s source attribute, as this force-reloads the content. This would introduce a larger performance overhead than is necessary. Instead, the `IframeHostComponent` will send `InputMessages` via the `ParentIntercomService` to the `iframe`. The `output` event of the `IframeHostComponent` is simpler, as it just listens for `OutputMessages` sent by the inner `iframe` to its host and then emits the payload as Angular output event. An example of how to use the `IframeHostComponent` is shown in Listing 3.2. Here, the parent wants to render a micro frontend called `mf-name`, passes the parameter `productId` with value `3`, and calls the method `onOutput` every time the micro frontend emits events. The implementation of `onOutput` (not shown) must then use the `$event.outputName` to differentiate what kind of output the child micro frontend emitted, so that `$event.outputValue` can be used correctly. For example, the child micro frontend could emit an output named `cartClicked` with a product ID to add to the cart as output value.

On the child micro frontend side, receiving initial and updated inputs was simplified via the `ChildIntercomService`. It provides a method called `getInput()` which returns a data stream that does two things. Initially, the query parameters of the child's `iframe`

are parsed. This will contain all initial values set by the `IframeHostComponent`. Afterward, whenever a `InputChangedMessage` is being received, the data stream will emit the updated value.

3.3.3 Retrospective

The primary goal of this iteration was to migrate the monolith towards `iframe` based micro frontends. While we did succeed in doing so, the path towards separated micro frontends had its challenges. The largest of which was the compiler setup. We are sharing libraries between different micro frontend instances by introducing aliases that point to the library distribution folders. For example, by writing an import of `"intercom"`, the compiler will load the files from `"dist/intercom"`. However, whenever a library is compiled, the dist folder will first be cleared. As a result, any compiler running in watch mode might crash, since some crucial files are now missing. This meant that every time we changed a library, we had to restart all Angular development servers for the changes to be picked up. While there might be solutions to this problem, it highlights the issues that can arise when working with micro frontends. Hence, developing micro frontends with `iframes` may not provide the same level of developer experience that monoliths have. One reason thereof is the increased complexity. For example, while the monolith has roughly 4769 lines of code, the `iframe` solution uses 7535 (increase of 58%). While many of those added lines are boilerplate code (e.g., each Angular application needs its own entry file to bootstrap the Angular application), those are still lines that must be maintained.

Finally, having multiple micro frontends could also mean having multiple versions of libraries at the same time. In our current monorepo setup, we only use a single `package.json` to define dependencies and their versions. However, it is also possible to use different versions for different micro frontends. As a result, developers potentially have to support different versions of libraries, depending on which environment the micro frontend provides. For example, the `"intercom"` library must be compatible with all micro frontends. However, some micro frontends may use older versions of Angular. So the library has to be compatible with a wide range of Angular versions. Since there are breaking changes with each major version of Angular, developers could be forced to publish different versions of their library to have feature parity between all micro frontends.

Regarding the communication channel `iframes` provide, `postMessage` is a good solution for one-to-one communication. For our use case, pushing messages towards and from micro frontends was simple enough to be a viable solution. The restriction of only sending serializable data has not been an issue for us, yet. However, the effort to send updates via a messaging port is clearly larger than a simple function call in the monolith.

Our biggest pain point is the lack of type information within our implementation. Since the library for communication should be message-agnostic, we do not wish to add type information directly into the intercom library. Otherwise, it is too tightly coupled to

our specific use case. Simultaneously, type information has to be shared across multiple micro frontends. Defining types of sent or received messages locally is possible but bears little use since other micro frontends won't be able to re-use them. A solution could be to export the type information of each micro frontend of received and sent messages, such that only such messages are allowed to be sent or received. Exposing this information, however, will again increase the complexity of the setup.

Lastly, we want to highlight the great isolation of `iframes`. This degree of independence allows micro frontends to be very stable. Therefore, micro frontends deployed today will still work later on. Since all libraries, styles, and the context of the micro frontends are self-contained, the only point of failure are the message interfaces the micro frontend understands and sends. Hence, accidental side effects when updating the app shell are very unlikely.

3.4 Iteration 3: State Management for iframes

3.4.1 Goal Definition

In our last iteration, we implemented micro frontends based on iframes, but still lacked sharing state between different instances. As a result, some features were still missing from the monolithic implementation (e.g., the shopping cart). Hence, in this iteration, we implement state management across all micro frontend instances in order to achieve feature parity with the frontend monolith.

Technologies Used

We use the same messaging channel we have already been utilizing in the previous iteration. Since iframes from different origins provide no other means for sending and receiving messages, postMessage has to be re-used for state synchronization as well. Therefore, deciding on how state synchronization is implemented becomes a question about the protocol to use on top of the messaging channel. We chose to implement publish-subscribe semantics via *data providers* and *data consumers*. The publish-subscribe messaging pattern as described in [49] allows *data providers* to send messages towards a data stream, without needing to know who the receivers will be. *Data consumers*, on the other hand, will subscribe to data streams, without having to know, where the data comes from. This degree of decoupling allows the pattern to be highly dynamic – producers and consumers can come and go as needed.

In the micro frontend context, *data consumers* are our micro frontend instances, which might need to consume some data in the shared store. Data producers are services that hold some shared state. In particular, we provide the shopping cart and product services as *data providers*, while, e.g., the checkout micro frontend is a *data consumer*. The other micro frontends interact with the checkout service in order to manipulate the current state. The diagram Figure 3.17 shows the general message flow of the publish-subscribe pattern as we implemented it. The consumer cannot access the data stream directly, or send a callback function to the *data provider*, due to the limitations of serialization of postMessage. As a result, the provider has to proxy the subscription of the consumer.

As seen in the diagram Figure 3.17, there are three types of messages. The first one is the *subscribe* message. The first parameter is a unique ID of the subscription, which allows us to connect all future messages to this initial request. The second parameter is the name of the data stream from which the consumer wants to receive updates. The second message is a *dataUpdate* event and is sent by the provider to the consumer every time the data stream emits a new value. To send these messages, however, the provider itself has to subscribe to the data streams. Because the data stream cannot leave the context of the provider, we have to proxy the subscription for the consumer. Hence, every time the source emits a new data event, its value is forwarded to the consumer. Finally, the consumer can send an *unsubscribe* message to the provider. It clears the provider's subscription to the data stream to free all resources. Therefore, this message

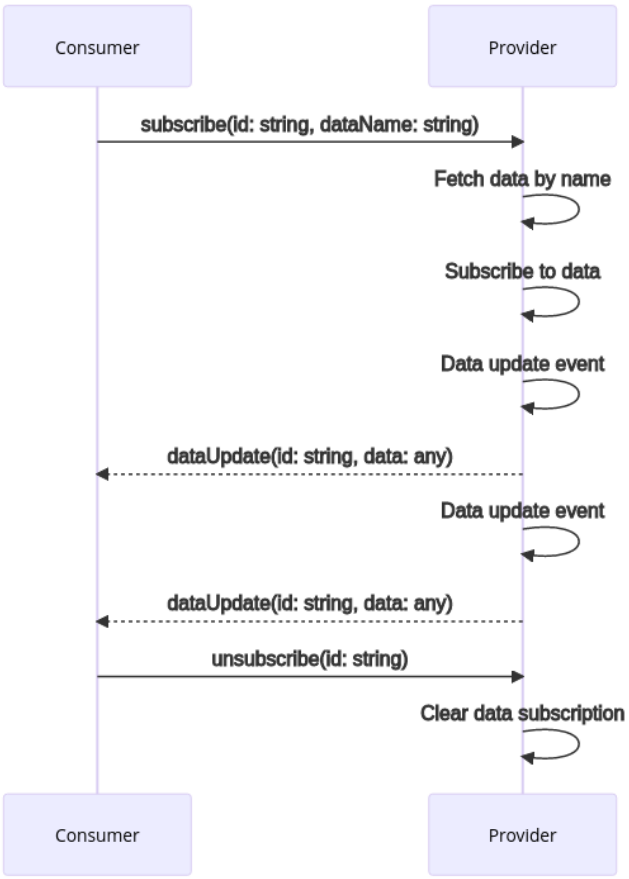


Figure 3.17: Publish-subscribe pattern shown via a consumer and provider. In this case, the consumer subscribes to the provider via some subscription ID and the name of the data stream to subscribe to. The provider resolves the data stream by name and subscribes to it. Every time the data stream pushes a new update, the provider sends a message to the consumer with the subscription ID and the emitted data. After some time, the consumer may unsubscribe with a subscription ID. This terminates the provider side subscription and no further messages will be sent.

must be called before a micro frontend is destroyed or the data is no longer needed. Otherwise, the provider will keep emitting data events, causing memory leakage and potential performance degradation.

Interestingly, those issues are very similar to those found in the backend. For example, suppose a website sells tickets for concerts. It uses a data stream that emits events when users have a concert's page open and when they buy a ticket. This allows the application to show live data on the frontend, i.e. how many people are currently on the page and how many tickets have already been sold. Since the data stream includes sensitive data, however, the frontend cannot access the stream directly, so the data and subscriptions

have to pass through a backend service first. In this scenario, the backend service will face the same issues as our micro frontend solution above.

3.4.2 Results

We noticed that the messaging pattern described in section 3.4.1 is reminiscent of the proxy pattern. A proxy provides functions, but instead of implementing the functions directly, the function call is forwarded to another service. This allows the proxy to forward requests to a remote service while behaving as if it was executed locally. We used this pattern to convert the existing stateful services to two separate implementations: The *data provider* in the app shell and the *data consumer* in the micro frontends. In Figure 3.18 the `ShoppingCartService` has been converted to an interface. Since the interface still exposes the same methods with the same signature, the Angular components do not need to be updated. The original implementation was then moved into the `ShoppingCartProviderService`. This service must be executed as a singleton across the full application – not just within each Angular context – to avoid data duplication and redundant API calls. For micro frontends to access the data streams returned by the `ShoppingCartProviderService`, we implemented the `ShoppingCartConsumerService`. When its methods are called, it sends a message to the `ShoppingCartProviderService` via the `postMessage` channel. The `ShoppingCartProviderService` then subscribes to the data stream on behalf of the micro frontend. Each time the data stream emits an event, the event is forwarded to the original subscriber. To reiterate, all micro frontends must use the `ShoppingCartConsumerService` and only the app shell must host the `ShoppingCartProviderService`. All Angular components still consume the `ShoppingCartService` as before, but the implementation is different depending on the context of the component.

In order to simplify the service implementation and avoid repetitive code, we have added a translation layer, which converts calls to the consumer service to calls of the provider service. The sequence diagram at Figure 3.19 reveals the full traversal of messages sent between the services for a subscription request and data response. The sequence diagram starts on the provider side. First, the translation layer `ConsumerProxy` [4] registers a listener for any store-related messages. Then the `ShoppingCartProviderService` is registered as a *data provider* at the `ConsumerProxy`. The `ConsumerProxy` now knows that whenever it receives messages targeting the `ShoppingCartService`, this provider needs to be called. Next, we assume that a component wants to interact with the `ShoppingCartService`. As the consumer is hosted in a micro frontend, it calls methods of the `ShoppingCartConsumerService`. In this case, `getShoppingCart()`

[4] We chose the name `ConsumerProxy`, as it receives requests from a consumer and forwards ("proxies") them to the provider.

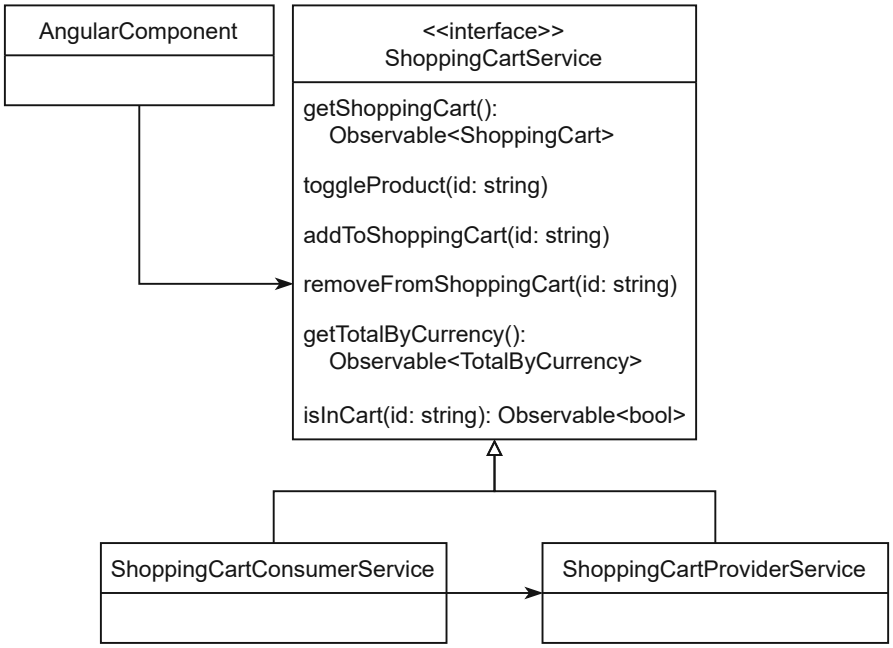


Figure 3.18: Class diagram showing the provider-consumer hierarchy. The shopping cart service was converted to an interface. It exposes the original method signatures. The shopping cart provider uses the same implementation as in the monolith. The shopping cart consumer, however, forwards all requests to the corresponding provider service. Components and services can still inject the `ShoppingCartService`. Depending on the context, the consumer or provider implementation is used.

is called. The `ShoppingCartConsumerService` sends a message to the app shell containing the following information:

- A `"type"` field with value `"subscribe"`. This message should trigger a subscription on the *data provider*.
- A randomly generated subscription ID. That way, both the consumer and provider can identify which messages belong together.
- The provider to address. In this case, we address the `ShoppingCartService`, so the value is set to `"ShoppingCartService"`.
- The action to call. Here, we forward the name of the method that was called on the `ShoppingCartConsumerService`. In this case `"getShoppingCart"`.
- Optionally, an array of arguments can be passed. However, the arguments are restricted by the serialization done by the browser when calling `postMessage`. Hence, no functions or other complex objects can be passed.

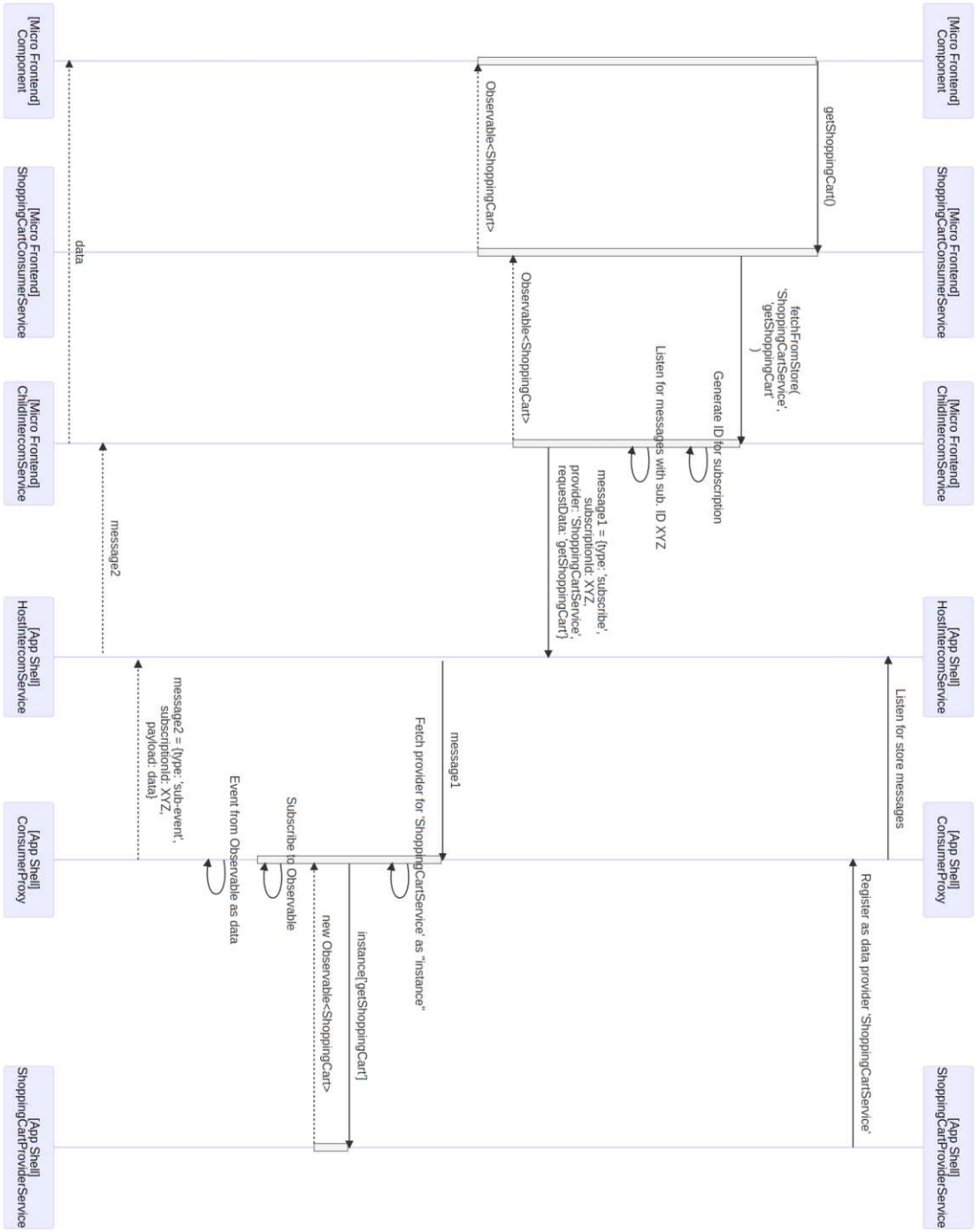


Figure 3.19: The complete process of communication between consumer and provider for subscribing to data and returning a data stream.

For better reusability, we implemented the message generation in the `ChildIntercomService`. It handles generating a unique ID, building the message, as well as listening for responses containing the generated subscription ID. This enables us to return a data stream to the caller. Every time a new message in response to the generated subscription ID is received, its payload is pushed to the returned data stream.

After the payload was sent to the app shell, the `ConsumerProxy` receives and handles the request. Using the value of the "provider" field of the message (in the example above `"ShoppingCartService"`), the `ConsumerProxy` resolves the corresponding *data provider*. If an instance is found, the method call will be forwarded to the provider. For the current example, this means calling the `getShoppingCart` method with no parameters. Because this method returns a data stream, the `ConsumerProxy` will subscribe to it. Every time the data stream emits a new value, the `ConsumerProxy` will be notified. With the received subscription ID, the `ConsumerProxy` can build a new message containing said ID, the emitted data, and the type `"sub-event"` to the `ChildIntercomService`. Since the subscription ID matches the previously generated ID, the `ChildIntercomService` can forward the received payload to the original caller of `getShoppingCart`.

Finally, the `ChildIntercomService` can parse the received message, understand which data stream the message is related to – based on the subscription ID – and push the message's payload to all subscribers – in this scenario the component. While there are more scenarios, they follow the same pattern and do not bear any more information than is already described above. Therefore, we will only shortly mention them:

- Data streams might complete, meaning that no further values will be emitted by it. In this case, all consumers in micro frontends must be notified by the completion as well. It, therefore, follows the same message flow as regular data.
- Data streams might throw an error. This will cause a data stream to stop emitting new values. So it behaves similarly to the completion, but the error object must be passed on as well.
- Data streams may never emit or complete. However, this case cannot be solved via the library and has to be checked if there is an issue with the data source.

3.4.3 Retrospective

We managed to extend the implementation of the previous iteration with an asynchronous protocol that enables state synchronization across multiple micro frontends. While it added a lot of complexity – as seen by the sequence diagram above – the proxy based approach means that it is easily possible to add more *data providers* as well as *data consumers*. Most of the complexity associated with the messaging layer between the provider and the consumer can be abstracted via proxy services. However, with the current implementation, the app shell and micro frontends still need to be configured

properly. For example, a micro frontend could provide the wrong implementation of the ShoppingCartService, therefore not using the shared state, but rather provide its own. Another issue is that the ShoppingCartConsumerService manually calls the ChildIntercomService to build the correct messages, leaving room for error. Both issues could be resolved by improving the library. However, such a solution will likely increase the complexity of the messaging, increasing the error surface and may worsen the traceability of the system.

Another weak point of our state synchronization is that sending messages had to follow the iframe hierarchy. While parent and child communication could be implemented as described above, we had to consider the communication between the parent and grand-children as well. Since micro frontends only send messages to their direct parent, we had to add more logic so that micro frontends are able to forward messages to their parent, in case they aren't the intended recipient. While it is possible to retrieve the top most parent with window.top, it may be possible for our application to be hosted inside an iframe as well. Micro frontends would therefore not get the app shell's window reference, but the hosting application that embedded our shop. This scenario is unlikely for our shop, but as mentioned we do not want to find solutions for our specific use-case. Instead, generally applicable patterns for communication and state synchronization are investigated.

A potential solution for this issue is creating another micro frontend, whose task is solely to provide the data streams required by micro frontends. Since its iframes will all be hosted at the same origin, more advanced browser features like MessagingChannel or BroadcastChannel as messaging channels could be used. They could also utilize SharedWorkers, which behave like background services that multiple application instances can interact with. Another advantage of SharedWorkers is the separation of the state synchronization logic from the app shell. Disadvantages, however, are the added complexity of the system and added artifacts to maintain. Loading additional iframes to the web application – even empty ones without any HTML being rendered – could also negatively affect the performance of the web application.

Finally, having the provider services centralized in one application creates the issue of instantiation: We do not want to create all possible services that any micro frontend could ever need at once. Rather, they should be loaded and executed lazily when they are needed. With the current implementation, the app shell cannot know which micro frontend requires which *data providers*, so it has to load and register all provider services at the start time. However, even if a micro frontend would be able to tell the app shell which services it requires, the app shell would still need to provide their implementation. In other words, the app shell has a high coupling to all provider services, since their implementation must be available inside it.

Alternatively, micro frontends may pass arbitrary URLs to the app shell in order to load the provider service code. While this allows us to remove the implementation from the app shell, it also forces us to execute micro frontend code inside the app shell, weakening isolation and potentially introducing an attack vector. Writing a secure management

service for external provider services, therefore, adds additional complexity to the code base, but also the infrastructure. Not only do we have to write and maintain a stable interface, but we also must provide means to register servers that serve the provider service's code.

To conclude, while we see the potential for our state synchronization, there are many pitfalls. Especially covering every edge case while still retaining good isolation of runtimes as well as code bases is no easy task. Having the bottleneck due to the `postMessage` limitations does not improve this situation either.

3.5 Iteration 4: Migration to Web Components and Implementing Direct Communication

During the development process, it became clear that `iframes` are great for isolation, but at the cost of complexity in regards to the integration of micro frontends. One reason is the lack of a shared environment between the micro frontends. Hence, sharing data and code between micro frontends inside `iframes` – while technically possible – requires complex protocols and patterns to be feasible. Therefore, instead of deepening the knowledge and possible implementations using `iframes`, we decided to investigate a different approach – one that better fits the requirements of micro frontends.

3.5.1 Goal Definition

For this iteration, where we are using Web Components instead of `iframes`, the focus lies on migrating the existing codebase. This includes creating a suitable development infrastructure. Features that should be provided by the infrastructure are hot reloading micro frontends, a process for developing and testing individual micro frontends (e.g., hosting a single micro frontend on a dedicated application), code sharing of libraries, exporting and importing type information, etc. Those goals go beyond the `iframe` based implementation because the decrease in isolation enables us to improve the development experience. At the end of this iteration, the SPA should provide basic functionality like displaying products from the recommendations. Hence, we want to achieve direct communication between micro frontends. More advanced features that require a global state, like the shopping cart, however, will not be implemented yet.

Technologies Used

Frequently, works describing micro frontends also explore the possibilities of Web Components [58, 89, 88, 90, 75]. Web Components are a rather new technology – they were first mentioned in 2011 by Russel [30] – that allows developers to reuse HTML elements. Firefox enabled the technology by default in 2018 [43]. The Web Component standard is a collection of multiple specifications related to sharing code between web applications [83]: shadow DOM, custom elements HTML templating, and ES Modules [63, 36]. The first one – Shadow DOM – is used to isolate part of the DOM (usually the inner HTML of a custom element) from the outside world – both from JavaScript and CSS access. Custom elements were designed to share code between multiple applications by defining new HTML elements. This is done by defining a JavaScript class, which is then added to the custom element registry with a new, custom HTML element tag. Whenever the browser detects the custom HTML tag in the DOM, an instance of custom element implementation will be created. The browser allows the custom element to react to lifecycle events (e.g., when the custom element is mounted into the DOM) or to attribute changes. The implementation can, for example, manipulate the element's inner HTML to render content based on attribute values. Finally, HTML templating enables creating a DOM tree once, any cloning it whenever needed, thus improving reusability and ES

```
import { createCustomElement } from "@angular/elements";

@NgModule({
  /** omitted for brevity **/
})
export class RecommendationsModule implements DoBootstrap {
  constructor(private injector: Injector) {}

  ngDoBootstrap() {
    // Wrap the RecommendationsComponent into a custom element
    const ce = createCustomElement(RecommendationsComponent, {
      injector: this.injector,
    });
    // Add the custom element to the browser's custom
    // elements registry
    customElements.define("mfe-recommendations", ce);
  }
}

```

Listing 3.3: Example of how *Angular Elements* can be used to create Web Components. The `createCustomElement` function wraps an Angular component into a custom element. The injector is necessary to enable dependency injection for the Web Component. To use the Web Component, it must be added to the custom elements registry with the browser's `customElements.define` function. The first argument constitutes the tag name of the custom element, the second parameter is the custom element class reference.

modules are a standardized way of modularizing JavaScript code (i.e., export and import from different files). However, the last two specifications play little role for this research, as they describe how code could be reused, rather than how micro frontends interact with their environment.

For this iteration, the existing Angular components must be converted to Web Components. Angular provides a library called *Angular Elements* [3], which wraps Angular components into custom elements. They can then be used like any other Web Component, including shadow DOM, custom events, and attribute binding. We converted Angular components to Web Components by passing the component's class reference to the `createCustomElement` function provided by the library `@angular/elements`. A code example thereof can be found in Listing 3.3. Here, we convert the existing code of the `RecommendationsComponent` to a Web Component. Note, that we do this in the `ngDoBootstrap` life-cycle method of the `RecommendationsModule`. This means, whenever this module is executed while bootstrapping the Angular application, instead of executing the default Angular logic for bootstrapping the application, we register the Web Components.

To implement direct communication, custom elements natively support listening to changes of attributes of the DOM element [58, 75]. This enables consumers of a Web

Component to pass arguments to a custom element. For upwards communication to the micro frontend host, custom events can be used [58, 75]. As the name suggests, those are DOM events that can have customizable payloads and properties.

As already mentioned, *Angular Elements* handles most of the heavy lifting. It automatically converts the outgoing messages marked with the `@Output` decorator to custom events and exposes all fields marked with `@Input` as bindable attributes of the DOM element. However, *Angular Elements* cannot convert the string values of the attributes to their correct expected types. For example, the `RecommendationsComponent` has an input called `groups` which modifies the number of recommendation groups fetched and displayed and, therefore, should be a number. However, since the runtime cannot know how to convert the string to a number correctly, the implementation of `RecommendationsComponent` also must be able to handle strings as input. In this case, the string can easily be converted to a number by casting the string. However, more complex parameters might need more complex parsing.

As for the infrastructure to handle hot reloading, building, and loading the Web Components, we use webpack's Module Federation. Usually, the webpack bundler assumes all dependencies to be local, meaning that during bundling, every dependency or JavaScript import must be present and will be added to the final bundle. With Module Federation enabled, webpack instead treats some dependencies of the JavaScript code as remote, so that they will be loaded at runtime from a specified server instead. For us, this meant that we could declare all micro frontends as remote code. To load a micro frontend, we could use regular JavaScript imports. webpack will detect them and fetch the remote code when it is needed.

This setup has several advantages. Firstly, since loading a micro frontend is an ordinary import, the TypeScript compiler as well as the IDE can resolve the type information for the micro frontend (e.g., by having the type information installed locally). This can have many advantages over the untyped approach we were forced to use in the `iframe` implementation (e.g., finding type errors at compile-time, getting better suggestions from the IDE while developing). Secondly, offloading the micro frontend orchestration to the infrastructure frees us from loading and managing micro frontends ourselves. Instead, webpack's Module Federation handles loading the code and re-using it when needed again. Thirdly, Module Federation also provides code sharing between multiple micro frontends, e.g., every micro frontend requires Angular as a runtime. Hence loading Angular only once and re-using it reduces loading redundant resources.

However, using Module Federation increases the complexity of the setup. Firstly, every micro frontend must declare Angular as a shared library. If it was missing, Angular would be included in the micro frontend's bundle. Secondly, all micro frontends should be able to agree on a shared Angular version. Module Federation uses semantic versioning to determine a globally compatible Angular version. If it fails to do so, Angular might be loaded several times. Luckily, there is a library called `@angular-architects/module-federation` [5], together with `@angular-architects/module-federation-tools` [6], which help in properly configuring webpack.

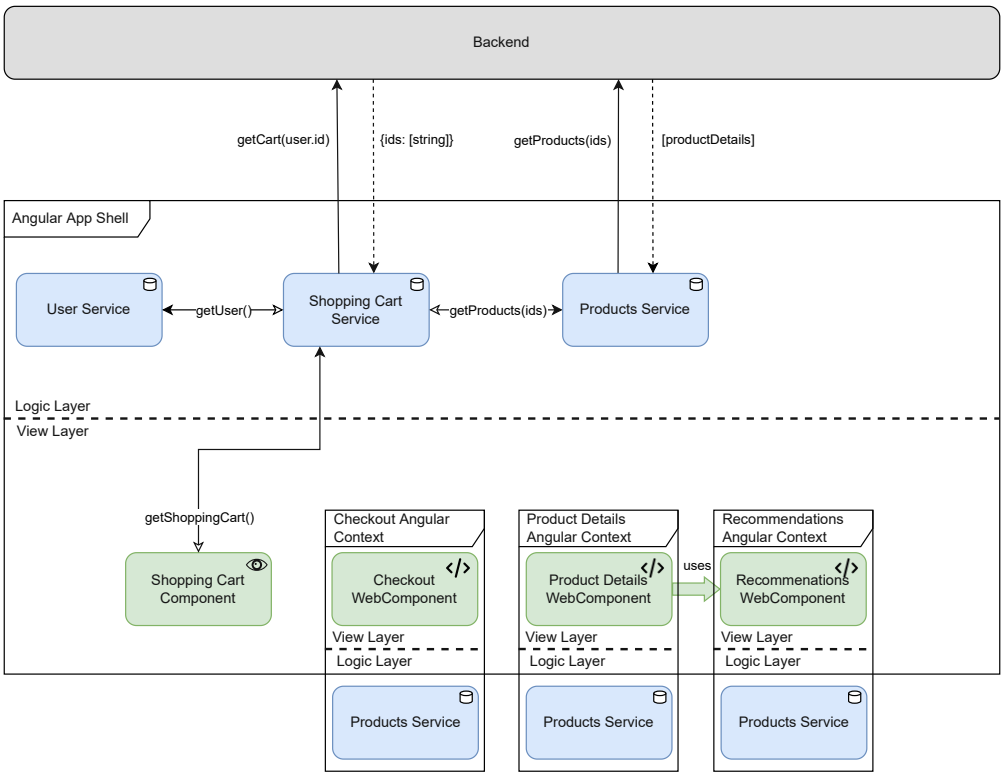


Figure 3.20: Architecture of the Web Component based micro frontend implementation. Each micro frontend has its own Angular context, containing instances of needed services (the backend communication has been removed for brevity). Each micro frontend exposes its components as a Web Component, which can be integrated into the app shell in the same way as regular HTML elements.

3.5.2 Results

We transformed the `iframe`-based web shop SPA to Web Components using *Angular Elements*. Since using *Angular Elements* together with webpack's Module Federation requires separate Angular applications for each micro frontend, the project setup of the previous iteration fitted the changed technologies. However, the code related to `iframe` based messaging was removed.

Since Web Components are running in the same context as the host application, they are not limited to using `postMessage`, thus widening the spectrum of possible communication channels. The final architecture of the Web Component based implementation can be found in Figure 3.20. The micro frontends are no longer separate web applications but are directly loaded into the app shell. Each micro frontend, however, retains its isolated Angular context. Therefore, they can create their own service instances.

In order to provide components to the app shell, each micro frontend exposes its Angular components as Web Components. Parameters are passed using HTML attributes and

messages are emitted via custom events. Apart from the already mentioned removal of code related to the `intercom` library, the change from `iframes` to Web Components had a great impact on the development process and environment. First, the Web Components may only use the minimum required features from Angular that are needed to create the custom elements. Therefore, during development, we created a separate Angular application for each micro frontend. The task of each application is to only host their respective micro frontend. This removes the need to run the full web shop app for developing individual micro frontends – hence achieving isolated development teams for different micro frontends.

Secondly, using the already mentioned Module Federation meant that each micro frontend had to be configured correctly to support the feature. To simplify the setup with monorepo projects, we used a helper library called `@angular-architects/module-federation`. It helps in configuring webpack in the right way, so that it uses the correct versions of shared libraries according to the `package.json` file as well as shared libraries added via TypeScript path aliases.

Nevertheless, the configuration for what dependencies should be shared and in which version still had to be done manually for each Angular application. For example, if one of the micro frontends was misconfigured and did not declare the `@angular/core` library as shared, then the library code would not be loaded at runtime dynamically. Instead, webpack bundled the library into the micro frontend's artifact. As a result, libraries no longer were provided as singletons. This easy oversight can not only lead to increased bundle sizes and hence load times, but also to more severe consequences for libraries that must be loaded as singletons.

An example of such a library is `Zone.js` [41]. This library instruments many global functions, so that they can be used by Angular for change detection. However, having multiple instances of `Zone.js` means that those patches are applied multiple times, which in turn can cause runtime errors or failed change detection [25]. For this reason, the library `@angular-architects/module-federation-tools` provides a bootstrapping function that can be used in place of Angular's default `bootstrapModule` function. It ensures that the singleton `Zone.js` instance is re-used across multiple Angular micro frontends.

Thirdly, the micro frontends, as well as the app shell, need to be integrated into each other. For the `iframe` implementation, we used a mapping of the micro frontend name and its URL. That way it was known at runtime which micro frontend is hosted on which server and how to load it into an `iframe`. For our webpack implementation using Module Federation, this mapping can be moved to webpack configurations. They can be configured at compile time or dynamically at runtime. We decided to use static configurations for the micro frontends, but a dynamic configuration at runtime for the app shell. The reasoning behind this is that the product-page micro frontend uses the recommendations as part of the product page. Therefore it is reasonable to define this information statically instead of having to configure it at runtime. To do so, we modified the `webpack.config.json` of the product-page micro frontend as seen in Listing 3.4.

This configuration uses the `ModuleFederationPlugin` to define `remotes`. This key-value map tells the webpack bundler, which imports (defined as keys) should be mapped to which server (defined as values). At runtime, this configuration is used to fetch the Module Federation metadata and the code of the micro frontend when it is first imported. The build process as well as the built artifacts are displayed in Figure 3.21. Each micro frontend has a separate build process with custom webpack configurations as described above. During bundling, webpack uses these configurations in order to bundle the micro frontend's source code. Additionally, as configured via the `filename` in Listing 3.4, a file called `removeEntry.js` is created. It contains meta data about which libraries this module depends on as well as which versions of those libraries are located on the server.

Finally, each shared library will be bundled into a separate file. This allows webpack to load a single library from the server, if it is required by one of the micro frontends. For instance, suppose webpack detects an import statement `import('recommendations/web-components')` at build time. Then, instead of bundling the code of said import into the final artifact, webpack loads the javascript file `"remoteEntry.js"` from the server `"http://localhost:4201/"` at runtime. This `"remoteEntry.js"` file contains metadata on which modules are exposed. In the case of a given import statement, the module `"web-components"` must be exposed by the `"recommendations"` namespace. Since the recommendations micro frontend is similarly configured as the product-page micro frontend shown in Listing 3.4, the `"web-components"` module points to a file that bootstraps the micro frontend as Web Component. As a result, after the example import was resolved, the product-page can use the Web Components defined by the recommendations micro frontend.

For the app shell, we do not want to statically configure every micro frontend's server location. Instead, it should be possible to fetch the list of available micro frontends from a server dynamically. We simulated this behavior by using promises that loaded the metadata from the other micro frontend servers. Later on, those promises could be easily replaced with an HTTP request or similar. The code can be found in Listing 3.5. There, the call to `loadRemoteEntry` fetchs the metadata from the given server and adds it to the webpack registry of different remotes. As a result, webpack knows which module has which dependencies and in which version. Only then, the bootstrapping file for the app shell itself is loaded (as declared via the `import('./bootstrap')` statement). As a result, webpack has time to negotiate versions of libraries before loading the actual application. For example, given that all micro frontends as well as the app shell share the Angular core library called `@angular/core`, each micro frontend might specify a slightly different version of the library. It could be that the app shell was built with Angular version 12.0.0, but the recommendations micro frontend was created a little later and uses Angular 12.1.0. However, since node libraries follow semantic versioning [79, 80], webpack knows that even though the app shell uses the APIs of Angular 12.0.0, Angular 12.1.0 must be backward compatible with Angular 12.0.0, as only the minor version changed – there can only be new features added, but no breaking changes. Therefore,

```
// Excerpt from
// web-components/frontend/projects/product-page/webpack.config.js
module.exports = {
  /** omitted for brevity */
  plugins: [
    new ModuleFederationPlugin({
      // Name of the micro frontend
      name: "productPage",
      // Generated file - must be loaded by the shell and other micro frontends
      filename: "remoteEntry.js",
      exposes: {
        // When importing the web-components, point to the micro frontend's
        // bootstrap.ts file
        "./web-components": "./projects/product-page/src/bootstrap.ts",
      },
      remotes: {
        // Statically define the web components this project depends on
        recommendations: "recommendations@http://localhost:4201/remoteEntry.js",
      },
      /** omitted for brevity */
    }),
  ],
};

```

Listing 3.4: Excerpt of the webpack configuration file of the product-page micro frontend. It defines exposed code as well as from which remotes a module and its metadata should be fetched.

webpack loads the highest compatible version of all modules. In this case, webpack will load the `@angular/core` with version 12.1.0 provided by the recommendations server.

To understand the full process of Module Federation, Figure 3.22 shows a sequence diagram to illustrate how the browser loads different parts of the code at different times. Initially, the browser loads and executes the JavaScript files for the app shell (marked with 1 and 2). As shown in Listing 3.5, in the first step the app shell loads the metadata of every micro frontend needed (steps 3 and 4). For demonstration purposes and to simplify the process, only a single micro frontend is shown in the sequence diagram. Furthermore, since the metadata of the app shell is already part of the app shell's code, it may not be loaded separately. In step 5, webpack's Module Federation evaluates the required versions of shared dependencies to find the optimal version to use. For demonstration purposes, the version negotiation allows loading all shared libraries from the app shell in step 6. It is possible, however that the version negotiation results in some libraries being loaded from different servers. After this step, the initialization process of the app shell is done. Later, the checkout micro frontend may be needed. For example, the user could navigate to the checkout page, which hosts the checkout micro frontend. At that time,

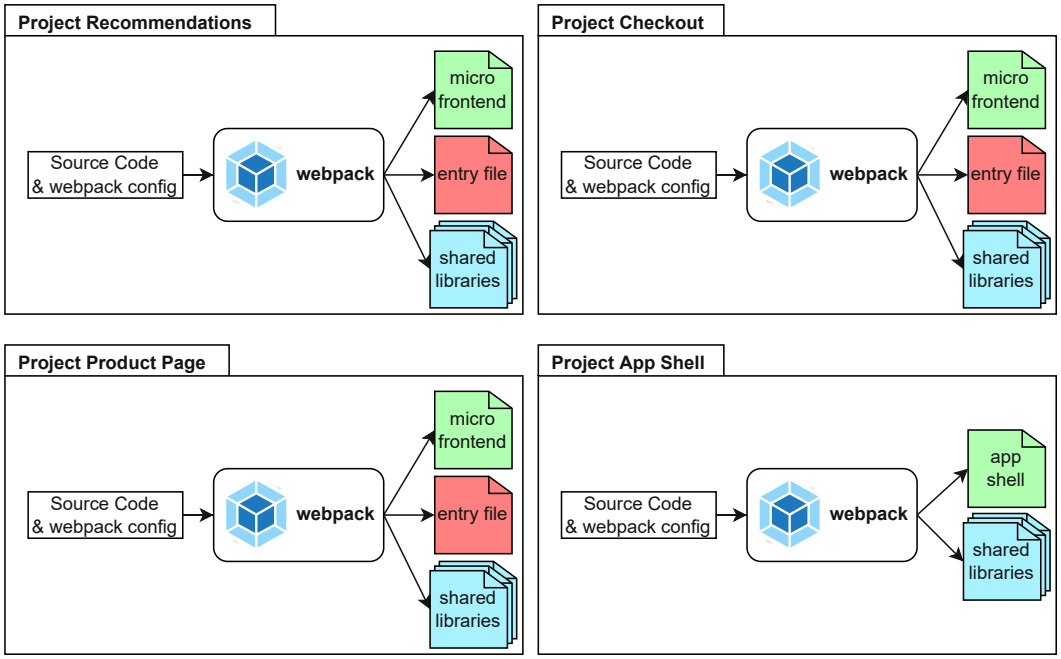


Figure 3.21: The created artifacts for all applications. All applications build the source code into executable JavaScript (marked in green). The micro frontends generate entry files (marked in red), which provide metadata for each micro frontend. Shared libraries are compiled into independent artifacts as well (marked blue).

the browser loads the micro frontend's code (steps 8 and 9). Since the checkout micro frontend might need additional dependencies, webpack has to determine which libraries have not been loaded at step 6. Usually, this is done when the checkout micro frontend tries to import a shared library. In steps 11 and 12, webpack found some libraries that have not yet been loaded and therefore loads them from the checkout server. Once again, libraries could have also been loaded from another server, if webpack needs to load a different version than the checkout server provides. This process may be repeated, once new micro frontends are being loaded.

In order to load and display a micro frontend inside the app shell, the already existing layout components – `LandingPageComponent` and `ProductDetailLayoutComponent` of the *products* Angular module and the `LandingPageComponent` of the *checkout* Angular module – load the Web Component code of each micro frontend. To illustrate this further, the TypeScript code of the `ProductDetailLayoutComponent` of the app shell is shown in Listing 3.6 and its HTML template in Listing 3.7. First, the component's constructor calls the `loadRemoteModule` function. This function is imported from `@angular-architects/module-federation`. Its purpose is to load a remote module – `"productPage"` in this case – and import an exposed module – in the given example `"./web-components"`. As discussed earlier, this loads the code of the product-page micro frontend that registers its Web Components into the browser's

```
// Excerpt from
// web-components/frontend/src/main.ts
import { loadRemoteEntry } from "@angular-architects/module-federation";

// Load all micro frontend metadata as well as the app shell
Promise.all([
  loadRemoteEntry(
    "http://localhost:4201/remoteEntry.js",
    "recommendations"
  ).catch((error) =>
    console.log("Failed to load MFE recommendations: ", error)
  ),
  loadRemoteEntry("http://localhost:4202/remoteEntry.js", "productPage").
    catch(
    (error) => console.log("Failed to load MFE productPage: ", error)
  ),
  loadRemoteEntry("http://localhost:4203/remoteEntry.js", "checkout").catch(
    (error) => console.log("Failed to load MFE checkout:", error)
  ),
])
  .then(() => import("./bootstrap"))
  .catch((err) => console.error(err));

```

Listing 3.5: Dynamic loading of metadata of the micro frontends. Calling `loadRemoteEntry` fetches the metadata of a specified micro frontend and adds it to the webpack runtime. The second parameter defines the module name to load.

custom elements registry. After the code of the product-page micro frontend was loaded, the `ProductDetailLayoutComponent` of the app shell can now use the product-page custom element as seen in Listing 3.7. Hence, the HTML template declares to render the element `mfe-product-page`. The tag name is declared by the micro frontend and must therefore be known by the consumer of the micro frontend.

Communication

Web Components mainly use two mechanisms for communicating with the outside world. Similar to Angular's binding with `Outputs` and `Inputs`, Web Components use custom events for upwards communication and attribute binding for passing parameters downwards. Custom events behave like native browser events, e.g., the click event or scroll event. Moreover, custom events can add a custom payload to the event instance. This allows the sender of an event to propagate additional information to all listeners of an event. Regarding Web Components, this means that they can tell their parent component that some special event occurred. For example, whenever a product is clicked in the recommendations micro frontend, it emits a new, custom event called `productClicked` with the clicked product as payload. The listener of the event can then decide what to do with this event. Attribute binding on the other hand is a technique added specifically

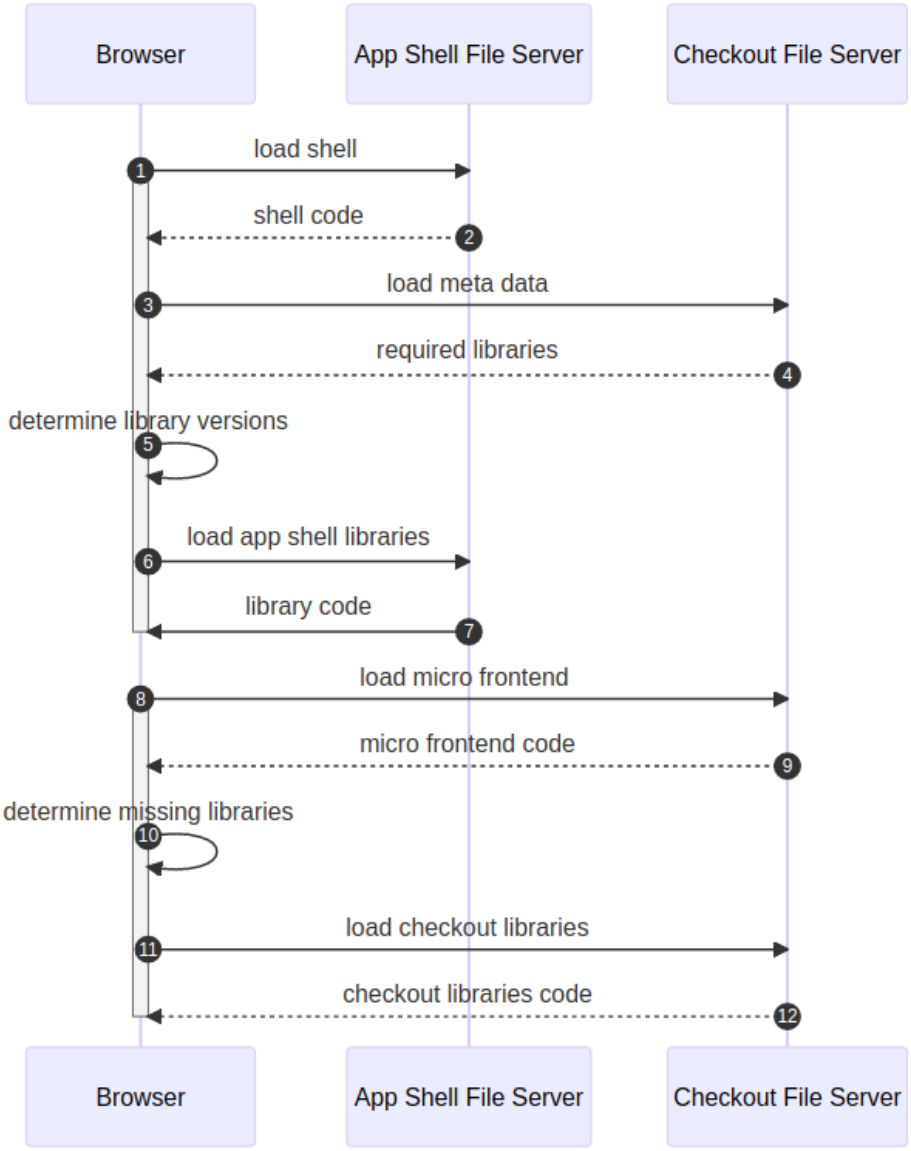


Figure 3.22: Sequence diagram for webpack's Module Federation. In this example, the app shell is loaded first, followed by the checkout micro frontend.

for custom elements. A custom element can declare the attributes it wants to observe via a static field called `observedAttributes`, which must be an array of strings with the names of the HTML attributes to observe. Then, the custom element instance can have a function called `attributeChangedCallback` with up to three parameters: The attribute name, the old value of the attribute, and the new value of the attribute. This function is called by the browser each time one of the observed attributes of the custom

```
// Excerpt from
// web-components/frontend/src/app/products/components/
//   product-detail-layout/product-detail-layout.component.ts
@Component({
  selector: "app-product-detail-layout",
  templateUrl: "./product-detail-layout.component.html",
  styleUrls: ["./product-detail-layout.component.scss"],
})
export class ProductDetailLayoutComponent {
  productId$ = this.route.params.pipe(map((params) => params["productId"]));

  constructor(
    private route: ActivatedRoute,
    private navigationService: NavigationService
  ) {
    // Load the micro frontend code asynchronously
    loadRemoteModule({
      remoteName: "productPage",
      exposedModule: "./web-components",
    });
  }

  // Called from the HTML template
  onRecommendationClicked(product: Product | null) {
    if (!product) {
      return;
    }
    this.navigationService.showProductDetails(product.id);
  }
}

```

Listing 3.6: Example code for loading a micro frontend and its Web Component. `loadRemoteModule` fetches the JavaScript of the `productPage` micro frontend and loads the `"web-components"` module. It then initializes the custom elements.

element changes. We have added an example of such a custom element in Listing 3.8. It observes value changes on the `product-id` attribute. The implementation of the `updateProduct` method is left out for brevity.

However, since HTML attributes are strictly string values only, passing more complex information downwards needs custom solutions. For example, if a micro frontend should only take and display data – without any logic on how to fetch it – the data must be encoded in some way. One solution is to use JSON encoding and decoding, but apart from the performance overhead this adds, it also couples the consumer of a micro frontend closer to the micro frontend implementation – namely the encoding and decoding of values. For this reason, attribute binding alone is not suitable for state synchronization across multiple micro frontends but can be used for direct communication. However, as

```
<!--
  Excerpt from
  web-components/frontend/src/app/products/components/
    product-detail-layout/product-detail-layout.component.html
-->
<mfe-product-page
  [productId]="productId$ | async"
  (recommendationClicked)="onRecommendationClicked($any($event).detail)"
></mfe-product-page>

```

Listing 3.7: Example usage of the product-page Web Component. The product ID to display is passed to the Web Component via the `productId` attribute. The custom event `recommendationClicked` calls the function `onRecommendationClicked` of the `ProductDetailLayoutComponent`.

```
class DummyProductComponent extends HTMLElement {
  static observedAttributes = ["product-id"];

  attributeChangedCallback(name, oldValue, newValue) {
    switch (name) {
      case "product-id":
        this.updateProduct(newValue);
      //...
    }
  }
}

```

Listing 3.8: Example code showing how custom elements can react to attribute changes. By implementing the `attributeChangedCallback` function, the browser will call it every time one of the attributes defined in the static `observedAttributes` array changes. The first argument of the function call is the updated attribute, followed by its old and new values.

mentioned in the goal definition, this aspect of Web Components is investigated in future iterations.

3.5.3 Retrospective

The goals of this iteration were to migrate the web application from `iframes` to Web Components and implement one-to-one communication from the micro frontend hosts to micro frontends and vice versa. Focusing on the migration process first, changing the codebase was little effort since we could reuse most parts of the project structure from the `iframe` implementation. Apart from the reimplementation of the communication channels, we did not have to rewrite large parts of the application to adopt Web Components. The migration was further supported by much better tooling surrounding

our use case. To implement the `iframe` based micro frontend, we had to find our own ways of managing remote micro frontends that should be loaded into the app shell. As a result, there is no support for loading type information for micro frontends or getting suggestions from the IDE on possible parameters and events of micro frontends.

For the Web Component based implementation, we used features of the webpack bundler to automate the process of wiring micro frontends together. Using Module Federation, we were able to map regular imports in our code to remote locations. As a benefit, we could delegate the management of those remotes to the tooling (i.e., loading remote code, keeping track of already loaded code, and sharing dependencies) instead of developing it ourselves. Hence, we only had to configure webpack properly. Additionally, since remote code is imported like regular libraries, the existing tooling for TypeScript can be re-used. Most importantly, it is possible to install type information of a micro frontend locally and the compiler and IDE will be able to resolve it correctly. As a result, there can be compile-time checks as well as advanced hints and recommendations by the IDE during the development process.

On the downside, using tooling to achieve our goals also means that the project is tighter coupled to that technology. For instance, if the tooling was to be abandoned or found to be insecure or flawed, the project might be stuck with the disadvantages the tooling has or requires extensive refactoring. In our case, webpack Module Federation is still a rather young concept – released with webpack 5 in 2020 [38]. Therefore, it is possible that webpack may change some features in the future. This could either to failing builds or runtime errors, because different projects were built with different webpack versions. Such impacts must be considered when choosing the tooling for Web Components.

Nevertheless, Web Components themselves are part of the DOM standard [11], meaning it is very unlikely that they change in a way that breaks the current Web Component specifications. Furthermore, having a shared standard (e.g., as compile target) across multiple tools also means that interoperability between them is very likely, freeing the developers to use whatever fits their requirements best.

Following the migration to Web Components, we implemented one-to-one communication between micro frontends and their hosts. Custom elements allow to listen for changes of specific attributes of the element in the DOM. Thus, whenever an HTML attribute's value is changed, the custom element can be notified via executing a method on the custom element's class. This feature enables passing arguments to a custom element as well as updating them when needed. For upwards communication from the Web Component to the host, custom events are used to attach an arbitrary payload to an event. Together, attribute binding and custom events allow us to send and receive messages to and from the micro frontend.

However, there are also drawbacks of using attribute bindings for downwards communication. Since DOM attributes can only be string values, their flexibility is limited. For simple arguments like numbers, booleans, or string values that might not be an issue. However, more complex parameters may require custom string parsing. For example, any

cyclic data cannot be converted to a string using the `JSON.stringify` function. Callback functions likewise cannot be serialized easily. Therefore, we conclude that attribute binding combined with custom events alone is not suitable for state synchronization.

3.6 Iteration 5: Shared Services for Web Components

3.6.1 Goal Definition

After using attribute bindings and custom events for communication, we want to implement lateral communication for state synchronization. Since all micro frontends are hosted in the same browser context, global services should be used. In order to prevent state duplication, all shared services must be provided as singletons so that no micro frontend creates new instances.

Technologies Used

Currently, it is possible to share library code between micro frontends, but each Angular context creates its own instances of the services, because each Angular application creates a new root Angular context. The micro frontends do not and should not share Angular context information. Otherwise, the isolation principle of micro frontends is broken. Nevertheless, Angular provides a special provider token called `platform`. According to the documentation, it uses a "special singleton platform injector shared by all applications on the page" [4]. In other words, instead of only looking for providers for services in the current context, Angularalso uses a global context. However, to avoid Angular-specific solutions for shared service instances, we decided against using platform providers. Instead, we apply a more general solution.

During the fourth iteration in section 3.5, where we used Module Federation for migrating the `iframe` based solution to Web Components, we recognized that the code of shared libraries was already provided as a singleton, i.e., it is only loaded once and reused for every new import. Thus, if the shared libraries not only provide the code, but instances of services instead, we can use Module Federation as a foundation to provide shared singleton services. That way our infrastructure handles and ensures service instantiation and uniqueness of them.

We recognize that using Module Federation for our research on state management may lead to technology-specific results. However, this setup can still be generalized to other implementations by having a loader for required services. Our approach is therefore reminiscent to the plugin pattern for developing frontend application [61]. Instead of referencing services directly, service consumers ask the service loader for a reference to the service instance. It is the service loader's task to use a global directory of already loaded services and only fetch and instantiate newly required services. In our case, we want to pass this task on to the tooling, instead of implementing it ourselves.

For micro frontend communication, we do not introduce a new messaging protocol or channel in this iteration. Instead, sending messages is performed via method calls of the singleton services. Nevertheless, calling a method is similar to sending a message to it. In Smalltalk [84], calling a function on an object is conceptionally described as sending a message to an object. The function name is the message type and the parameters are part

```
export class ShoppingCartService {
  private static instance: ShoppingCartService;

  static getInstance() {
    if (!ShoppingCartService.instance) {
      ShoppingCartService.instance = new ShoppingCartService(
        HttpService.getInstance(),
        ProductsService.getInstance()
      );
    }
    return ShoppingCartService.instance;
  }
}

```

Listing 3.9: Factory for the shopping cart service. If an instance has already been created, it is returned to the caller. If no instance is available, a new one is created. For all dependencies of the shopping cart service, factories are called. This procedure ensures that only a single instance is used across the full application.

of the message's payload. Thus, implementing a mechanism that allows communication via shared services is analogous to communication via shared memory.

3.6.2 Results

In order to remove Angular's dependency injection from the application, we created factory methods for each shared service as shown in Listing 3.9. The factories' task is to check if a global instance has already been created. If so, the cached instance is returned. Otherwise, the service is instantiated and cached. The factories also must handle creating dependencies of the service by calling other factories. Therefore, our implementation uses no dependency injection. Instead, our refactored services are plain JavaScript classes, so no external framework or library is needed to use them. This shows that our implementation of shared state can easily be integrated into any web application or framework.

In order to keep the component implementations as they are, we added Angular *providers* for the shared services. Angular *providers* are a mechanism that allows us to declare dependency injection tokens and a factory to return the value that should be used for the token. In our case, the dependency injection tokens are the shared service class references and the factories are the same as mentioned above. Having those providers means that the shared service instances are referenced in each Angular dependency injection context. In the architecture diagram in Figure 3.23, we symbolized this via a dashed border around the services in each Angular context. As a result, no components have to be altered, since the services are injected as before.

Consequently, the architecture is very reminiscent of a monolith. Instead of communicating via messaging channels, the different parts of the application now reference the

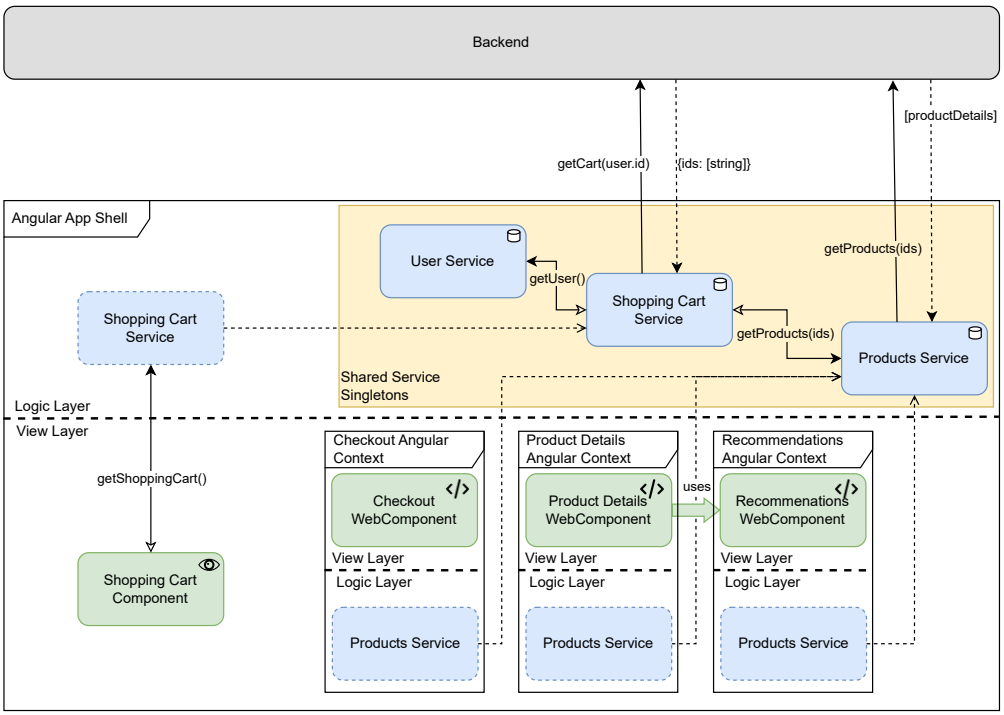


Figure 3.23: Architecture of the shared services of the Web Components. Instead of providing service instances in their own contexts, each micro frontend references the global, singleton instance.

same service instances. Therefore, our implementation no longer has issues passing and receiving complex objects like data streams. This enables a development experience, which is close to the familiar frontend monolith. Since the shared code can be imported like any other regular library, the type information is available to the IDE and compiler as well, thus reducing the danger of having to cast messages to their proper types manually.

3.6.3 Retrospective

Compared to the complexity of the state synchronization of the `iframe` based solution, using shared services was a much smaller and simpler change to the code base. The largest refactoring was replacing Angular's custom HTTP client service with browser native `fetch` calls. Since all shared services are no longer Angular service, but regular JavaScript classes, they can no longer access Angular specific features – such as the HTTP client. The logic of the services, as well as the consumers of them, did not change. Not having to worry about the communication between micro frontends and the stateful services removes the burden of maintenance. For `iframes`, we had to implement our own communication layer on top of the `postMessage` messaging channel. The additional code introduces a larger surface for errors, especially when the protocol has to

be expanded for new use cases. None of this is present with shared services. Developers can therefore focus on implementing the services.

In our opinion, having type information available is a great benefit of using shared services with Module Federation. As already mentioned, lacking types of messages requires casting them to educated guesses of their shape. Errors can therefore easily be introduced by changes to the messaging protocol. However, the backward compatibility of shared services still has to be guaranteed by developers.

However, using singleton services provided via webpack Module Federation has drawbacks as well. Foremost, misconfigurations of webpack can be very hard to trace and debug. In our monorepo example app, this is less of an issue, since the library developers also have control over the other applications. For micro frontends in different repositories, misconfigurations could easily occur due to the lack of proper static analysis across all projects. Hence, the services would no longer be provided as global singletons, breaking the state synchronization.

One solution to this issue could be creating a registry of services on the global `window` object. Instead of checking whether a local variable in the current module is set, the factories could check if a service instance is already available in the `window` object. Misconfigurations can still occur, but this approach is more resilient against them. In the best-case scenario, the only drawback is that the shared service code is loaded redundantly, but is still only executed once. In the worst case, due to the lack of webpack's version negotiation, the misconfigured micro frontend may require a newer version than was loaded by the application. Hence, some features may not be present, causing runtime errors. Those issues might be hard to trace, since a web application may work fine while it is being developed, but any library version update in any of the micro frontends could lead to broken micro frontends. There is no silver bullet to solve the issue, yet. Future work may develop mechanisms or libraries to tackle those issues. For now, having a strict configuration management system before deploying composed web applications is recommended.

Another disadvantage of our implementation is the use of build tools to handle a task that should be part of the application code. As a result, micro frontend developers are forced to use the same tooling. As discussed earlier, webpack is a common tool supported by most web frameworks. Nevertheless, by introducing a required dependency across all micro frontends, we break the goal of technology heterogeneity. In the future, there might be newer tools or frameworks, which build on tooling not compatible with webpack. If that was the case, a substantial refactoring of the web application would most likely be required. This is the reason why technology isolation is so important in the micro frontend architecture. As of this thesis, Module Federation is a promising development that may be adopted by other tools as well. For example, there are extensions for esbuild [54] and Vite [32] to enable similar features to or support for Module Federation. Both build tools are modern alternatives to webpack.

3.7 Iteration 6: Message Bus for Web Components

3.7.1 Goal Definition

In the previous iteration, we achieved singleton services across multiple micro frontend instances. In doing so, we have introduced a coupling of our micro frontends to the service implementations, i.e., micro frontends need to know which services have which responsibility and how to interact with them. For example, each micro frontend project must be configured correctly, otherwise, the services might not work properly.

On the other hand, shared service developers must consider, which services expose which interfaces and who the consumers are, e.g., some service method may have side effects or strict history constraints. Currently, we have the history constraint that the constructor must not be called anywhere outside the provided factory. Those constraints can be hard to communicate and easy to break. Instead, we want to decouple the consumers from the producers by implementing a messaging bus for our application – inspired by message brokers used in the microservice architecture [56, 64]. Data producers (e.g., the stateful services) publish messages to the message bus, while *data consumers* (e.g., Angular components) subscribe to the data streams. We have already discussed this publish-subscribe pattern in the context of our `iframe` implementation. However, now we have the benefit of sharing the same context between micro frontends. Hence we are not limited to using a hierarchical messaging bus to route data between micro frontends.

Technologies Used

In contrast to Nishizu et al. [75], we chose not to implement a custom messaging bus from scratch. Instead, we want to use browser-native features. Thereby our findings can more easily be applied to other projects, as our implementation is not application specific. For this reason, we have chosen the `BroadcastChannel` API, which is similar to named queues in RabbitMQ [26]. In the example in Listing 3.10a, a client wants to interact with the “cats” channel. Therefore, the client passes the channel’s name to the `BroadcastChannel` constructor. The client can then receive messages by adding a listener to the `"message"` event. The provider in Listing 3.10b also creates a new `BroadcastChannel` with the same name. When the provider sends a new message using `postMessage`, the client receives a new `"message"` event. Each `BroadcastChannel` therefore is a multi-directional port to access the named channel. In the context of the micro frontend architecture, `BroadcastChannels` are a good candidate for implementing the publish-subscribe pattern, which we have already used in previous iterations. In comparison to the shared service implementation in section 3.6, however, consumers no longer need to know the source of the data, but only the data stream’s name. This greatly decouples the *data consumers* from the data source.

In the previous iteration, in order to guarantee the uniqueness of service instances, we had to ensure that all projects are configured properly. If a project does not declare the service library as shared code, webpack treats it as a local library and bundles it

```
// Consumer Code
const channel = new BroadcastChannel("cats");
channel.onmessage = (msg: MessageEvent) => {
  console.log("Received cat:", msg.data);
};

```

(a)

```
// Provider Code
const channel = new BroadcastChannel("cats");
channel.postMessage({ color: "black", name: "Oscar" });

```

(b)

Listing 3.10: Code example for the `BroadcastChannel` API. Both consumer and provider can create an instance of the `BroadcastChannel`. Since they use the same channel name (`"cat"`), they reference the same data stream. When the provider sends a message to the channel, all current subscribers to the channel receive the message.

into the project. Hence the code is executed twice and duplicate instances are created. To avoid the reliance on the tooling and a proper configuration thereof, we decided to use a `SharedWorker`. The `SharedWorker` API allows us to spawn a new background thread, which multiple browser contexts can access – hence "shared". Similar to the `BroadcastChannel`, a `SharedWorker` is identified by the string that is passed to its constructor – the URL to the source code to execute. In other words, if multiple browser contexts (tabs, windows, `iframes`, etc.) call the constructor with the same URL, they can interact with the same `SharedWorker`. This allows us to use browser features in order to guarantee that only a single instance of our code is loaded and executed. Therefore, the chance for misconfiguration is reduced.

3.7.2 Results

We started by creating two `BroadcastChannels` per service. One for sending out commands to a service (analogous to calling methods of the service), and another for broadcasting the current state, e.g., the current shopping cart. This setup allows us to decouple the operational stream from the data stream. In order to prevent misuse of either stream, we established factory functions that either return `read-only` streams or `write-only` streams. For example, *data consumers* receive data, but send commands. Hence, the data `BroadcastChannel` is a `read-only` stream, while the command stream is a `write-only` stream. For *data providers*, however, it is the other way around – commands are read while state updates are sent. In order to share message channels between applications, we created a new library that is imported by each micro frontend. However, in contrast to the previous implementation, micro frontends no longer need to configure the library as "remote" using webpack's Module Federation. The library can

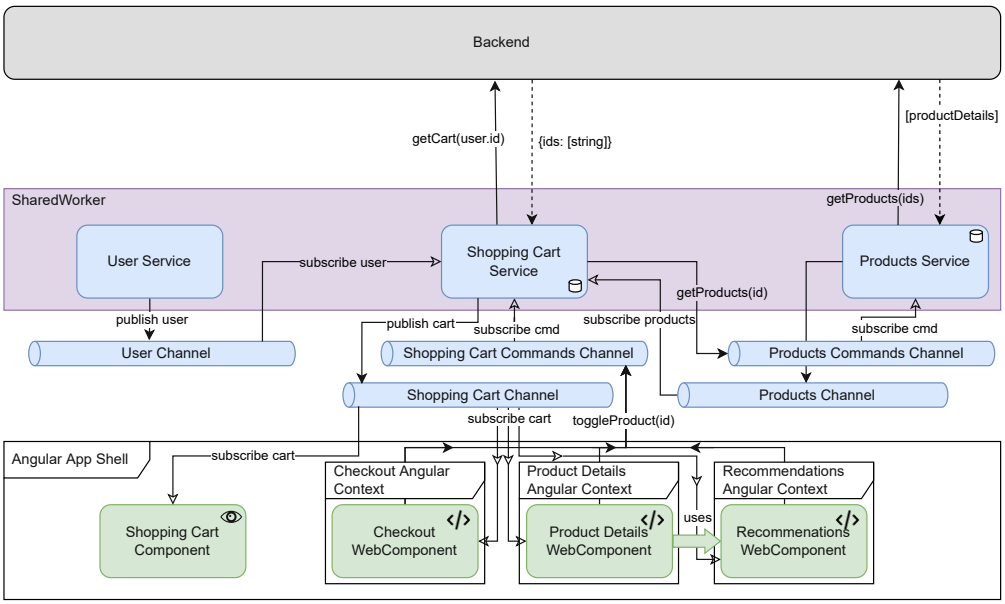


Figure 3.24: Architecture using a `SharedWorker` as a service layer. The micro frontends no longer use service instances directly, but rather subscribe to data streams and submit messages to command streams. For service instances running in the worker thread, it is the other way around.

be bundled into the micro frontend's bundle. Since all communication is now done via the message channels, micro frontends no longer need to import the service library.

Furthermore, some of the commands sent to a service require a direct response. For example, when adding a product to a shopping cart, we might want to display a loading spinner until the request is completed – either successfully or with an error. For those cases, a caller can send a (randomly generated) name of the response channel alongside the command. If such a response channel is given, the handling service will send the response to the `BroadcastChannel` with the given name. After the client has received the response, the channel is no longer used and can be destroyed. This pattern is also used for backend publish-subscribe services like RabbitMQ [22]. After the messaging was in place, we migrated the services to a `SharedWorker`. Then, we needed to create the bootstrapping code that starts our services as singletons. We created another messaging channel so that micro frontends or the app shell can notify the `SharedWorker` which services are currently needed. The `SharedWorker` then loads the required service via a dynamic import. Those lazy imports keep the initial `SharedWorker` code as small as possible.

The resulting architecture can be found in Figure 3.24. Instead of interacting with services directly, micro frontends subscribe to data streams. Commands can be sent to services using their respective command streams. Micro frontend components are hence completely decoupled from the service implementation. Vice versa, services no longer

need to know the different consumers of their functionality. The separation therefore allows us to decouple the service's context from the component's context. As a result, it was easily possible to execute services in the shared worker – a separated thread from the UI thread. In theory, this could improve the performance of the application, since the application logic can be executed asynchronously, non-blocking, and therefore doesn't block the web page from rendering.

3.7.3 Retrospective

Separating the representation layer from the logic layer had its unique challenges and interesting consequences. Compared to the previous iteration, this approach has advantages, but also disadvantages. Starting with the advantages, the largest benefit of using the message bus is the possibility of running the service layer in a different browser context. Being able to decouple the service layer strictly from the UI layer allows the creation of a new vertical team just for shared services. In the previous iteration, shared service developers were restricted by the different micro frontends, e.g., if a project is not able to leverage webpack 5 and its Module Federation, the shared services could not be used as easily. A micro frontend might also misuse a service by calling its constructor, disregarding the requirement to run it as a singleton. Using a `SharedWorker`, however, means that the shared service developers only need to ensure that at least one micro frontend – usually the app shell – loads and starts the service layer. This means much less communicational and organizational overhead between the teams.

The second advantage of our implementation is the context-spanning state. Since both the `SharedWorker` and `BroadcastChannels` can be used across multiple browser contexts (tabs, windows, workers, etc.), we can share the application state even beyond the current browser tab. Hence, adding a product to the shopping cart in one window automatically updates all other browser contexts as well. Especially caches benefit from this, as duplicate requests can be avoided even across browser tabs and windows.

A smaller, but still interesting advantage of `SharedWorkers` is that they do not block the rendering thread. Usually, JavaScript is executed inside a single thread. Hence, a single, computationally expensive task could block any other tasks from executing. As a result, the browser cannot execute any code required to render the webpage. This causes it to freeze until the code has terminated. Moving the service layer into a `SharedWorker` causes the JavaScript code to be executed in a separate thread. Hence, even computationally expensive tasks can be done independently of the rendering loop. However, the added overhead for cross-thread communication negatively impacts the performance. Thus, depending on the number of messages sent in comparison to the CPU time needed for the JavaScript code, our solution may improve or decrease a web application's performance.

Nevertheless, there are also disadvantages to our implementation. The largest disadvantage is that our approach lacks type safety. Using Module Federation, we were able to use the shared services' types directly in our code without needing to provide separate

type declarations. We tried to mitigate the issue by providing typed wrappers around the `BroadcastChannels`. However, a vertical team could easily misuse or not use the library. Hence, a micro frontend may inject data into channels, it isn't supposed to. Sadly, we cannot prevent this, therefore reducing the type safety of the system. Secondly, `BroadcastChannels` have the same restrictions we have already discussed for the `iframe`'s `postMessage` interface. Most importantly, it is not possible to send complex messages through channels. All data must be cloneable, which prevents callback functions from being passed along a message. As a result, we cannot send objects with methods over the channels. Hence we had to implement a different, more elaborate way of responding to messages via ephemeral response channels. Lastly, by moving the service layer code to a `SharedWorker`, we had to ensure that the service worker understands and provides all required services. This means that the `SharedWorker` becomes a centralized registry of all possible services and combinations. In the previous iteration, Module Federation handled the handshake and version resolution of libraries between all micro frontends. This means that Module Federation scales well with new features and libraries. For example, if a new micro frontend requires a new library, it will simply load it from its JavaScript repository. With a `SharedWorker`, new libraries must be understood by the centralized registry, hence most likely requiring code changes. With enough effort, it might be possible to implement a similar mechanism to webpack's Module Federation inside the `SharedWorker`. As of now, however, to the best of our knowledge such a system does not exist and we must therefore consider it a drawback of the `SharedWorker`.

For those reasons, it is hard to verify if all parts of the application use the API correctly at compile time and runtime. Bugs that might arise from incorrect usage of the `BroadcastChannels` can be very hard to trace, since messages don't have a trace attached to them. Combined with the side effects that messages in the channels have, a bug's cause might be hard to detect and fix.

CHAPTER 4

Comparing the Results

In the last few chapters, we developed a monolithic web application and split it into micro frontends using `iframes`, followed by Web Components. We investigated how the different micro frontends could communicate with each other – hierarchically as well as laterally. What we have found are multiple patterns for communication and state synchronization of micro frontends.

In this chapter, we qualitatively compare all the different solutions based on the last implementation of each micro frontend technology. For the monolith, we are using the results of the first iteration. The `iframe` based solution was completed after the third iteration. Finally, for the micro frontends using Web Components, we take the status after the sixth and last iteration for comparison.

The first analysis is the current performance of each implementation. Then, we measure the development effort via two change requests. The unit of measurement is the number and category of changed files. After each change request, we again measure the performance and compare the values against the previous measurements.

4.1 Baseline Performance

In this chapter, we measure the performance baseline for each implementation. It will demonstrate the overhead each micro frontend approach has over the monolith. The performance properties we evaluate are the consumed memory (JavaScript as well as Graphics Processing Unit (GPU) Random Access Memory (RAM)), the number of HTTP requests, the amount of data transferred, as well as the duration until all resources have been loaded. Computational utilization is not being evaluated, as this measurement is very volatile due to its dependency on the computer's state and background tasks. Furthermore, CPU utilization always stabilizes towards 0%. Thus the comparison between measurements is non-trivial and therefore out of scope of this thesis.

To ensure the comparability of each implementation, we decided on a fixed environment in which the applications are executed. First, we removed all randomness from the system. Thus, the endpoint for fetching recommendations always returns the same products to ensure the same images are loaded with every performance test. As a result, the same URLs will deterministically load the same products, including their images and texts.

Secondly, all applications are served with the same command line arguments. We decided to run the applications using the production configuration of Angular. It ensures that the source code is optimized and minimized. Furthermore, the flag `-no-live-reload` prohibits the application from creating a WebSocket. During development, the WebSocket is used to notify the client that the source code has changed. The client can then reload the application. For our performance test, we removed this overhead. The final command for running each application is as follows:

```
ng serve -configuration production -no-live-reload
```

It ensures that each application can be executed with as little overhead as possible. The comparison of the application performances is, therefore, more accurate and less noisy.

Thirdly, each performance measurement must be executed on the same hardware and software. We execute the tests on an AMD Ryzen 7 1700X 16-threads CPU with an NVIDIA GeForce GTX 970 GPU. The installed operating system is Manjaro Linux with kernel version 5.19.16. The used Nvidia GPU driver version is 520.56.06. For the browser, we decided on using Chromium version 106.0.5249.119 released in October 2022 due to its provided developer and performance measurement tools. As a windowing system, X11 is used.

Apart from the test environment, the test procedure needs to be identical across every measurement. First, each test must be run inside a private window. Private windows clear the user data and caches when they are closed. Therefore, they minimize the amount of persistent data between runs. To disable network caches, we opened the developer tools and ensured that network caches are disabled before each test. Finally, Chromium's task manager allows us to inspect the memory consumption of each open tab.

To extract the data during performance testing, we followed the same steps for every run. After the site under test loaded, we used the development tools to measure the time until the last HTTP call is finished, the amount of data transferred, and the number of HTTP requests. We differentiated between style sheets, JavaScript files, and API calls. Afterward, we closed the development tools, to avoid additional memory being consumed for them. Finally, we measured memory consumption using Chromium's task manager. Since memory consumption is usually higher after the page has loaded, we waited until the memory consumption stabilized. The data points we collected are "Memory footprint", "GPU memory", and "JavaScript memory" ("alive" only).

The sites we measured are the landing page and the product detail page. The first represents a page with a single micro frontend (the recommendations). The product detail page represents a page with multiple, nested micro-frontends. The checkout view

has not been tested, as – test-case-wise – it is identical to the landing page, i.e., a page hosting a single micro frontend.

4.1.1 Results

Starting with the web application's memory consumption on the landing page (Figure 4.1a and Figure 4.1b), the average total RAM footprint was 36KB for the monolith, 41.376KB for the `iframe` implementation, and 40.389KB for the Web Components. Therefore, the `iframes` use on average 13.12% more memory than the monolith. The Web Components have a 9.14% higher average memory footprint.

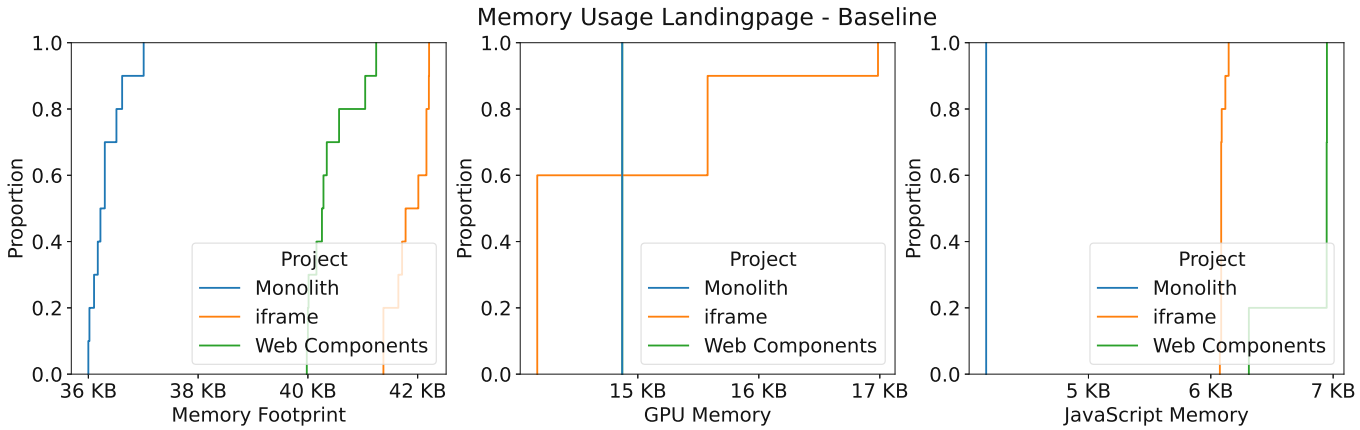
Interestingly, the Web Components have the highest memory consumption for allocated JavaScript objects (6.8204KB; +63.79% compared to the monolith). `iframes` are in the middle with 6.0938KB used (+46.34%). The monolith uses the least JavaScript memory with 4.164KB. We assume that the `iframes` use less JavaScript because most of the orchestration and isolation is handled via the browser's native code, thus not counting towards the JavaScript heap.

The GPU memory consumption is rather stable across all test runs. The monolith and the Web Components had the same GPU memory usage across all measurements (14.871KB and 14.872KB respectively). The `iframes` had a little more variation. Their average was 14.873KB, with a minimum of 14.169KB and a maximum of 16.985KB. We explain the lower minimum value due to the higher isolation of the frames, which might lead to possible optimization paths that are not possible if all styles are – potentially – applied globally. The higher variation could be explained by the higher total memory consumption. Therefore, the browser might get into a state that prevents further optimization.

The product detail page shows a similar behavior (Figure 4.1c and Figure 4.1d). The monolith and the Web Components use a little less total memory compared to the landing page with 33.924KB and 39.178KB on average. The reason lies in the less complex layout and fewer DOM elements to render. The `iframes`, however, increased their memory consumption by 0.72KB to 42.616KB. Even though the site is less complex, it has to render two `iframes` instead of one. Since providing the different browser contexts is a memory-intensive task, the total memory used is higher than before.

The GPU memory shows little difference compared to the landing page. Both the monolith and Web Components use a stable 14.08KB of memory. The `iframes` have higher variation with an average of 13.459KB used.

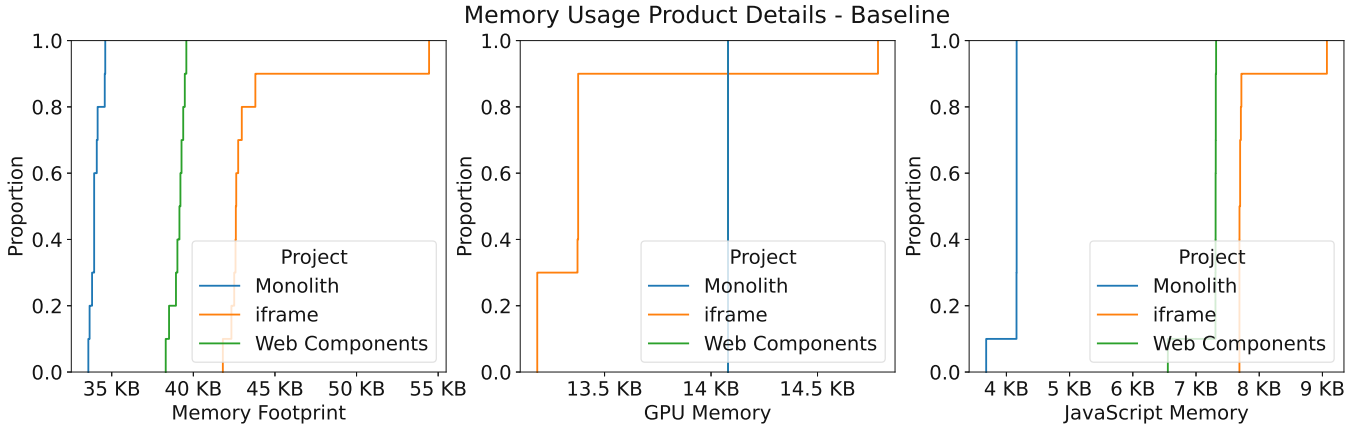
Finally, the JavaScript memory consumption shows that the monolith is mostly unaffected by the different pages. It uses on average 4.113KB, which is 0.05KB less than the landing page average. The Web Components and `iframe` had an increased JavaScript memory usage by 0.415KB to 7.236KB and by 1.739KB to 7.833KB respectively. The additional overhead comes from the nested micro frontends, which have to be kept in memory too.



(a) Baseline memory measurements of the landing page.

	Memory Footprint (KB)				GPU Memory (KB)				JavaScript Memory (KB)			
	min	max	mean	median	min	max	mean	median	min	max	mean	median
Project												
Monolith	36.00	37.01	36.33	36.26	14.87	14.87	14.87	14.87	4.16	4.16	4.16	4.16
Web Components	39.98	41.24	40.39	40.27	14.87	14.87	14.87	14.87	6.31	6.95	6.82	6.95
iframe	41.38	42.21	41.86	41.90	14.17	16.98	14.87	14.17	6.07	6.15	6.09	6.08

(b) Statistical analysis of the baseline memory usage of the landing page.



(c) Baseline memory measurements of the product detail page.

	Memory Footprint (KB)				GPU Memory (KB)				JavaScript Memory (KB)			
	min	max	mean	median	min	max	mean	median	min	max	mean	median
Project												
Monolith	33.57	34.60	34.02	33.92	14.08	14.08	14.08	14.08	3.68	4.16	4.11	4.16
Web Components	38.31	39.57	39.08	39.18	14.08	14.08	14.08	14.08	6.55	7.32	7.24	7.31
iframe	41.81	54.44	43.84	42.62	13.18	14.78	13.46	13.38	7.69	9.07	7.83	7.69

(d) Statistical analysis of the baseline memory usage of the product detail page.

Figure 4.1: Baseline memory measurements.

For the `iframe` approach, the overhead is much larger, because each `iframe` has to hold a separate JavaScript context and Angular application.

Next, regarding the network performance measurements, on the landing page (Figure 4.2a and Figure 4.1b) the monolith loaded the fastest with 515.5ms to finish loading. During this time, the monolith transferred 1.9MB of data. The `iframe` implementation was the slowest and needed 622.4ms (+28.5%) for 2.4MB. The Web Components managed second place with 598.8ms (+16.2%) on average and 2.3MB sent. Looking further into the data, the Web Components require the most data for JavaScript – 27KB more than the `iframe` and 339KB over the monolith. While this appears counter-intuitive since the `iframes` have to load parts of the JavaScript twice (e.g., the Angular framework), it shows that Module Federation does not allow for as much optimization as when the compiler can treat the code as local-only. Additionally, each JavaScript file now has to carry proper meta-data as well, thus increasing the total bundle size even further.

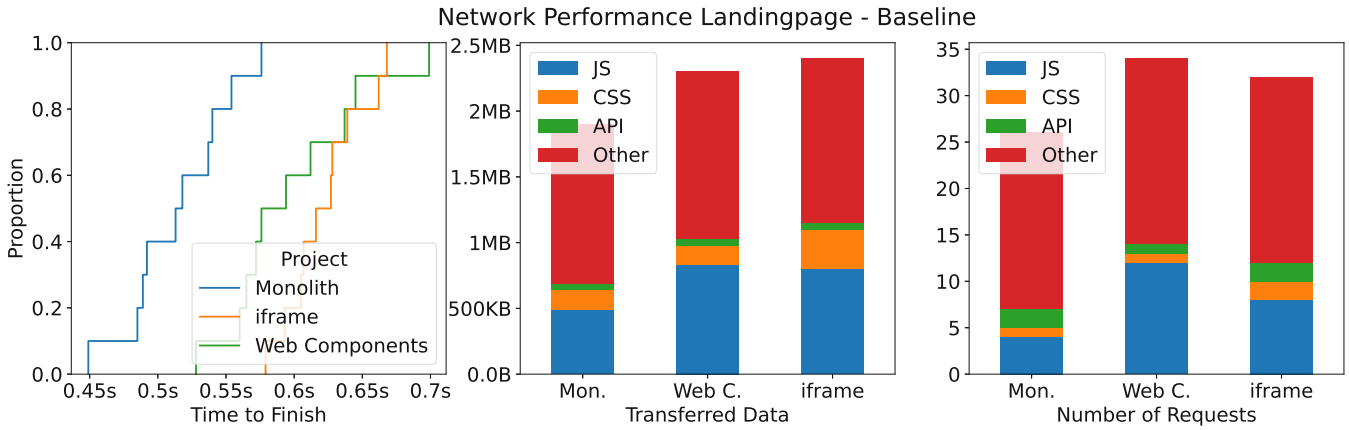
Using Module Federation for the Web Components also increased the number of requests needed to fetch the JavaScript files. While the monolith and `iframes` needed 4 and 8 calls respectively, the Web Components had to load JavaScript files 12 times. This also highlights the disadvantages of not knowing at compile-time, which code can be delivered as a single file and which cannot.

On the product details page, the monolith could finish a little faster with 512ms (-0.68%) on average. The `iframes` were slower with 805.5ms (+57.3% over the monolith, +29.4% over the landing page) on average. The reason – similar to the memory footprint – is the nested `iframes`. The Web Components loaded in 611.1ms on average (+19.6% compared to the monolith, +2.1% over the landing page).

In comparison to the landing page, the product page showed nearly the same performance for the monolith (512ms time to finish (-0.6%), 491KB JavaScript transferred (±0%)). The Web Components (611.1ms (+2.1%), 931KB (+12.2%)) and `iframes` (805.4ms (+29.4%), 1.1MB (+37%)), however, decreased their performance due two the nested micro frontends. The number of fetched JavaScript files stayed the same for the monolith as well but increased for the `iframe` and Web Components to 11 and 13 respectively.

4.2 Change Request 1: Adding a Micro Frontend as *Data Consumer*

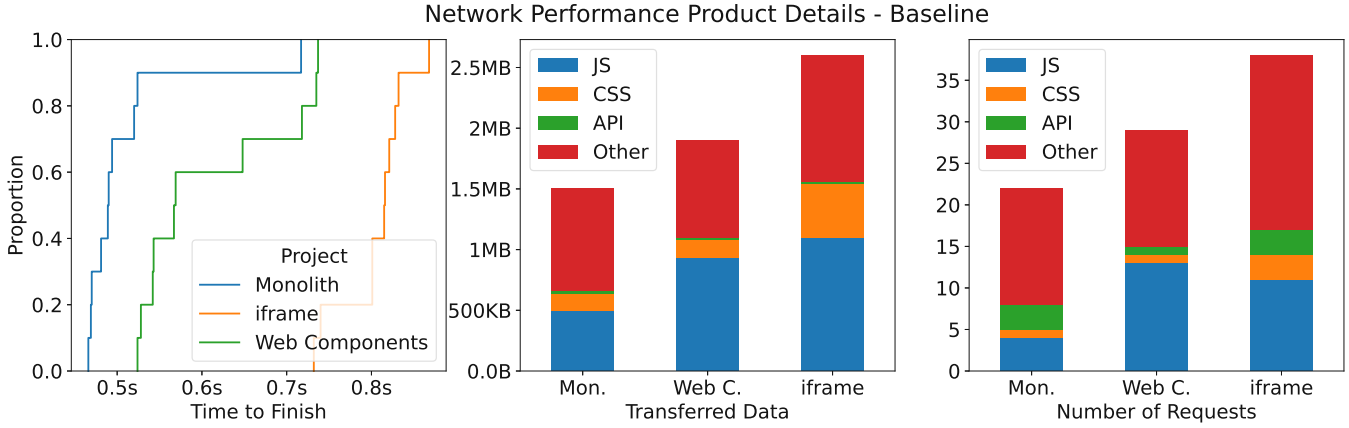
In this chapter, we test the extensibility of each prototype by adding a new *data consumer* to the application. Since communication patterns need to scale with new consumers, it is important to measure the impact a new message receiver has on our system. In this case, the *data consumer* is a new micro frontend to simplify the usage of the shopping cart



(a) Baseline network performance of the landing page.

	Time to Finish				Transferred Data				Number of Requests			
	min	max	mean	median	JS	CSS	API	Total	JS	CSS	API	Total
Project												
Monolith	449ms	576ms	515.3ms	515.5ms	491KB	148KB	50.2KB	1.9MB	4	1	2	26
iframe	579ms	668ms	622.4ms	621.5ms	803KB	294KB	50.2KB	2.4MB	8	2	2	32
Web Components	528ms	699ms	598.8ms	585.0ms	830KB	149KB	48.9KB	2.3MB	12	1	1	34

(b) Statistical analysis of the baseline network performance of the landing page.



(c) Baseline network performance of the product detail page.

	Time to Finish				Transferred Data				Number of Requests			
	min	max	mean	median	JS	CSS	API	Total	JS	CSS	API	Total
Project												
Monolith	466ms	717ms	512.0ms	489.5ms	491KB	148KB	19.9KB	1.5MB	4	1	3	22
iframe	732ms	868ms	805.4ms	815.5ms	1.1MB	442KB	19.9KB	2.6MB	11	3	3	38
Web Components	524ms	737ms	611.1ms	568.0ms	931KB	149KB	17.1KB	1.9MB	13	1	1	29

(d) Statistical analysis of the baseline network performance of the product detail page.

Figure 4.2: Baseline network performance.

button. Therefore, each existing shopping cart button needs to be replaced by the new micro frontend.

After we implemented the changes, we use the Git history in order to extract the changed files.[1] Then, we categorized each change into one of four different categories: *Feature*, *Config*, *Bugfix*, and *Other*. The first category, *Feature*, groups changed files, which are directly related to the new feature (e.g., new files for the new shopping cart button component, files modified to replace the old shopping cart button with the new component, etc.). Secondly, the *Config* category includes non-code changes that had to be done in order to add the new micro frontend (e.g. updating URLs for new micro frontends). This category indicates how easily new micro frontends can be integrated into an existing application stack. The *Bugfix* category highlights issues that we found during developing the new feature. Finally, *Other* contains all files, which have little to no effect on the change request (e.g., comment updates, style changes, formatting, etc.).

Each file is only categorized with one label. If a file would fit two labels, the one higher up takes precedence. For example, if a file got modified by integrating the new micro frontend, but also fixed a bug while doing so, it is categorized as *Feature*.

	Category	Total	New	Modified
0	Feature	10	3	7
1	Config	0	0	0
2	Bugfix	0	0	0
3	Other	1	1	0

(a) Monolith

	Category	Total	New	Modified
4	Feature	21	13	8
5	Config	10	6	4
6	Bugfix	1	0	1
7	Other	5	3	2

(b) `iframes`

	Category	Total	New	Modified
8	Feature	21	17	4
9	Config	14	9	5
10	Bugfix	0	0	0
11	Other	3	3	0

(c) Web Components

Figure 4.3: Changed files for each project after the first change request.

As seen in Figure 4.3, the monolith had the fewest files changed with 10 feature-related changes and one added test file. The `iframe` based implementation had to update 32 files, including 21 files related to features, 10 config updates and 1 bug fix. The bugfix was related to a minor bug of missing a break in a switch-case statement, causing a misleading log to be printed. Additionally, 5 files are categorized under *Other*, mostly related to testing files, updated shell scripts, and styles. For the Web Components, we had to update 35 files, 21 of which are *Feature* changes. The remaining 14 files are config updates. 3 additional files had to be changed to accommodate new test files and a `favicon.ico` file for the new Angular project.

The monolith clearly shows its strength here, as it has the lowest amount of changes required to implement the feature. We had to create a new component – the three new files in Figure 4.3a – and replace the old shopping cart in all existing components. The `iframes` and Web Components, however, required us to generate new projects. Hence, the higher number of new files. Additionally, for both micro frontend approaches, we

[1] We decided against using line-based changes for this evaluation since line changes are too noisy and unstable. They depend on the code style as well as the usage and length of comments, which are not related to the feature. Hence only the number of changed files will be taken into consideration.

have to update multiple configurations, at least one for each micro frontend consumer, to integrate the new remote component.

Most importantly, however, none of the projects required us to modify the existing code for the micro frontend communication. Meaning, our solutions for inter-app communication are stable regarding new *data consumers*.

Analyzing the memory performance (Figure 4.4), we noticed a small impact on the monolith (+6.58% average on the landing page, +5.73% for the product detail page). The `iframe` implementation, however, had a very high decrease in performance (+304.29% average memory consumed on the landing page, +112.84% on the product detail page). Finally, the Web Components has shown a medium increase in memory usage with a +17.9% increase on the landing page and +9.93% on the product detail page respectively.

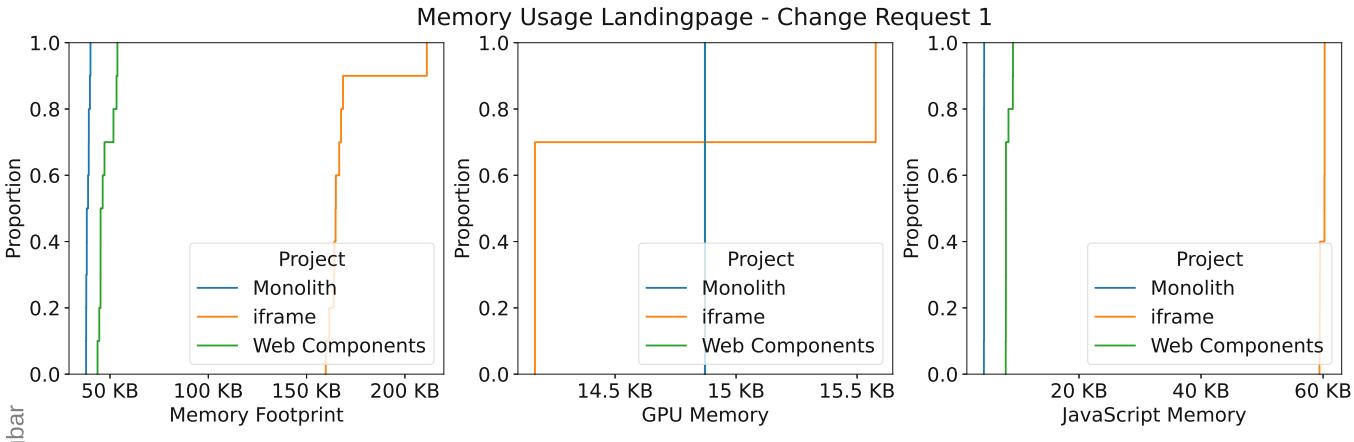
The network performance shown in Figure 4.5 follows a similar trend. The monolith showed an increased time to finish by +22.47% on the landing page by +14.26% on the product detail page. The Web Components had +22.23% higher load times on the landing page and +32.83% on the product detail page. The `iframes` struggled the most with an +236.76% increase on the landing page and +190.91% on the product detail page.

The transferred data also reflects the increase in resource consumption. The monolith transferred a total of 1.9MB for the landing page and 1.5MB for the product page. The overhead for the added component is therefore 0%. The Web Components resulted in a slight increase to 2.4MB (+4.34%) on the landing page and 2.2MB (+15.79%) on the product page. In contrast, the use of `iframes` resulted in significantly higher transferred data, with a total of 20.2MB (+741.66%) and 7.5MB (+188.46%) loaded by the landing and product pages, respectively. This is due to the duplication of requests for each micro frontend.

Our testing showed that the `iframes` had the poorest performance. As each shopping cart is an additional micro frontend instance, the increase in memory consumption and network traffic are most evident on the landing page. Web Components scaled more gracefully with the additional micro frontend. The monolith, however, had the lowest impact on performance, as well as developer effort to migrate to the shopping cart buttons.

4.3 Change Request 2: Adding Translations as *Data Provider*

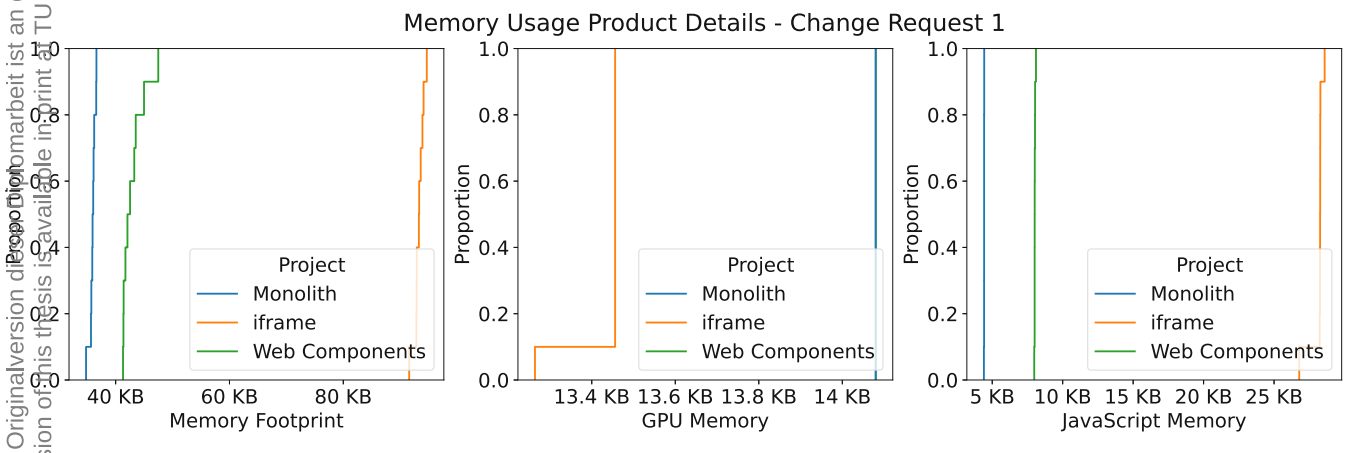
With this change request, we analyze the stability of the systems concerning new *data providers*. Similar to the change request before, our messaging patterns must scale with new message senders. If adding a new message source were a large undertaking, the system could not be updated or extended easily. For this research, a new service is added



(a) Memory measurements of the landing page after the first change request.

	Memory Footprint (KB)				GPU Memory (KB)				JavaScript Memory (KB)			
	min	max	mean	median	min	max	mean	median	min	max	mean	median
Project												
Monolith	37.83	40.06	38.72	38.53	14.87	14.87	14.87	14.87	4.41	4.46	4.44	4.44
Web Components	43.66	53.78	47.62	45.75	14.87	14.87	14.87	14.87	7.99	9.18	8.29	8.02
iframe	159.74	211.14	169.25	164.81	14.17	15.58	14.59	14.17	59.41	60.26	59.93	60.23

(b) Statistical analysis of the memory usage of the landing page after the first change request.

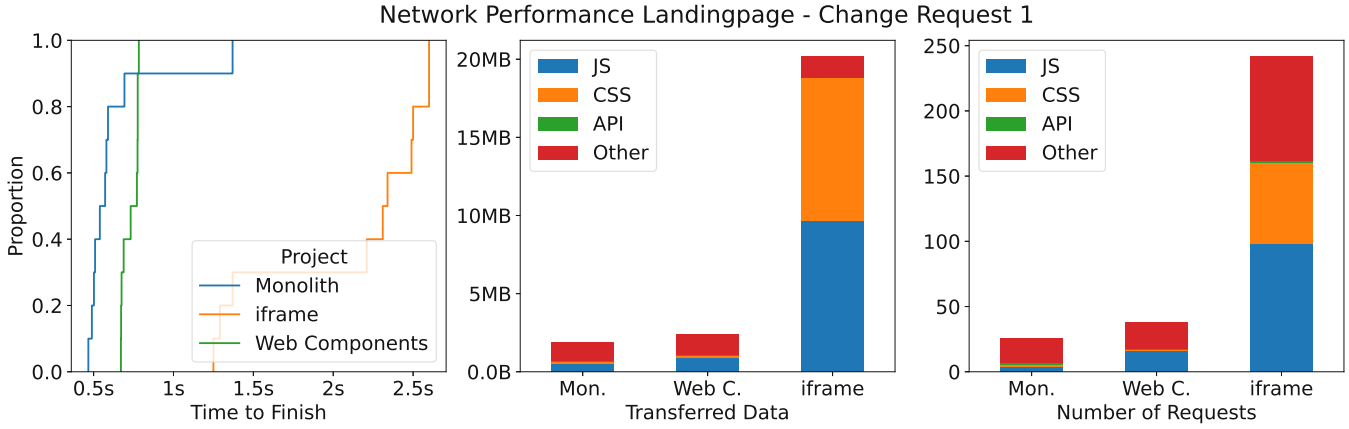


(c) Memory measurements after the product detail page after the first change request.

	Memory Footprint (KB)				GPU Memory (KB)				JavaScript Memory (KB)			
	min	max	mean	median	min	max	mean	median	min	max	mean	median
Project												
Monolith	34.80	36.62	35.97	36.00	14.08	14.08	14.08	14.08	4.41	4.43	4.42	4.42
Web Components	41.30	47.49	42.97	42.30	14.08	14.08	14.08	14.08	7.98	8.10	8.02	8.01
iframe	91.58	94.68	93.31	93.31	13.26	13.46	13.44	13.46	26.79	28.59	28.15	28.27

(d) Statistical analysis of the memory usage of the product detail page baseline after the first change request.

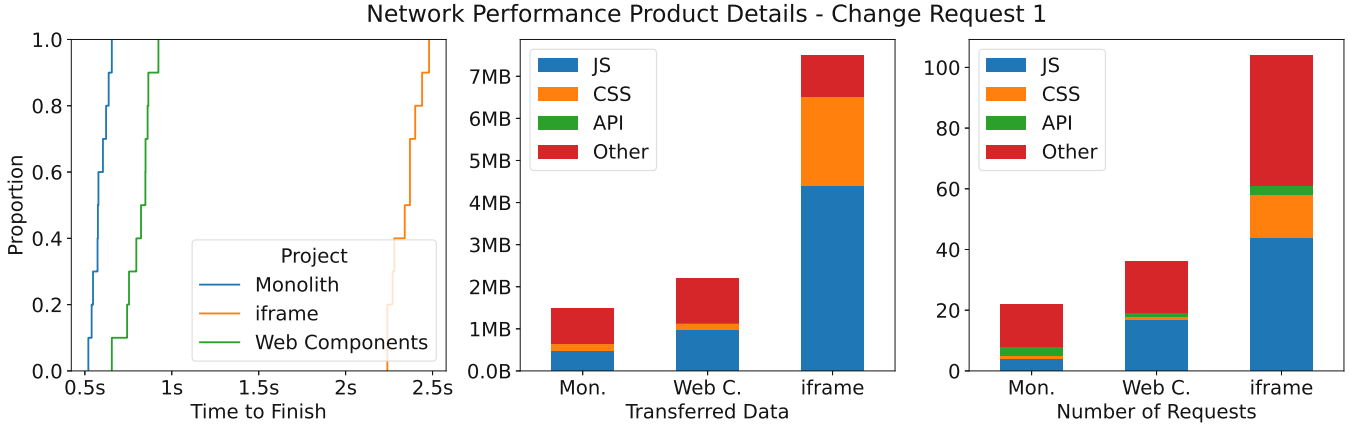
Figure 4.4: Memory measurements after the first change request.



(a) Network performance of the landing page after the first change request.

	Time to Finish				Transferred Data				Number of Requests			
	min	max	mean	median	JS	CSS	API	Total	JS	CSS	API	Total
Project												
Monolith	467ms	1370ms	631.1ms	555.5ms	491KB	148KB	49.8KB	1.9MB	4	1	2	26
iframe	1250ms	2600ms	2096.0ms	2325.0ms	9.7MB	9.1MB	49.6KB	20.2MB	98	62	2	242
Web Components	671ms	783ms	731.9ms	751.5ms	896KB	149KB	49.5KB	2.4MB	16	1	1	38

(b) Statistical analysis of the network performance of the landing page after the first change request.



(c) Network performance of the product detail page after the first change request.

	Time to Finish				Transferred Data				Number of Requests			
	min	max	mean	median	JS	CSS	API	Total	JS	CSS	API	Total
Project												
Monolith	520ms	655ms	585.0ms	576.5ms	490KB	148KB	19.1KB	1.5MB	4	1	3	22
iframe	2240ms	2480ms	2343.0ms	2355.0ms	4.4MB	2.1MB	19.1KB	7.5MB	44	14	3	104
Web Components	655ms	923ms	811.7ms	835.5ms	977KB	149KB	17.1KB	2.2MB	17	1	1	36

(d) Statistical analysis of the network performance of the product detail page after the first change request.

Figure 4.5: Network performance after the first change request.

to all three implementations to translate texts dynamically on the page. All existing micro frontends need to pass a translation key to the service and receive a stream of data containing the translated texts. Thus, the translation service is a new *data provider*, similar to the existing services for product data or the shopping cart.

	Category	Total	New	Modified
12	Feature	30	4	26
13	Config	0	0	0
14	Bugfix	0	0	0
15	Other	1	1	0

(a) Monolith

	Category	Total	New	Modified
16	Feature	30	8	22
17	Config	1	0	1
18	Bugfix	0	0	0
19	Other	2	2	0

(b) `iframes`

	Category	Total	New	Modified
20	Feature	27	5	22
21	Config	1	0	1
22	Bugfix	6	1	5
23	Other	14	1	13

(c) Web Components

Figure 4.6: Changed files for each project after the second change request.

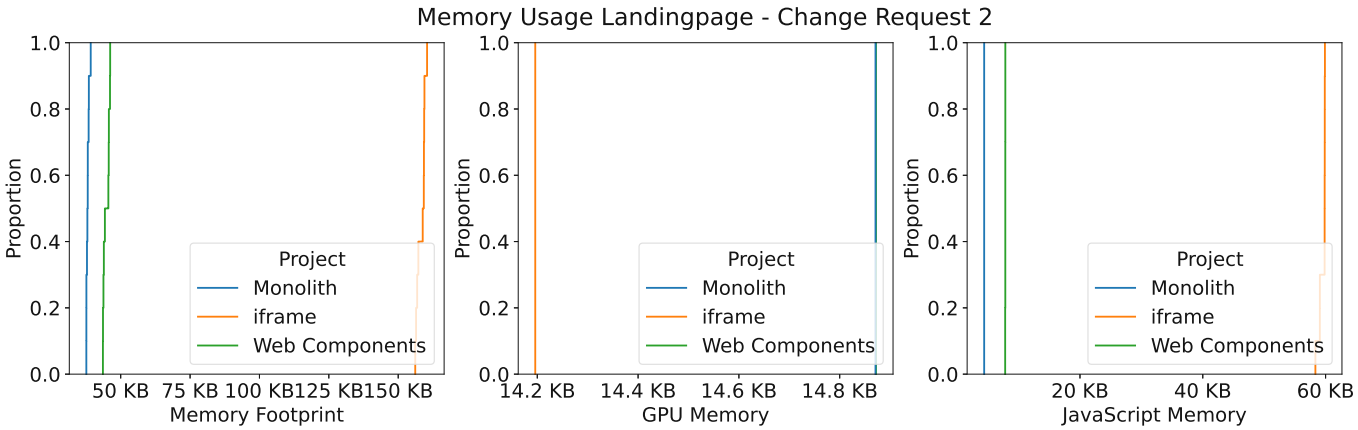
In this iteration, however, we have found a crucial bug in the Web Component implementation. It had a race condition between subscribing to a *data provider* and the first emitted value by the *data provider*. As seen in Figure 4.6, we had to change 6 files in order to fix this bug. Additionally, 13 files had to be updated due to a misspelling in one of the libraries. While unintentional, discovering this bug shows the difficulties communication between micro frontends can have. Due to their added complexity, the surface for bugs increases and the maintainability decreases.

The changes related to features, however, reveal that every implementation requires roughly the same changes made to files. Both the monolith and the `iframe` applications required 30 files to be changed. We changed 27 files for the Web Components. Additionally, both the `iframe` and Web Components had to register the new *data provider* in the system. Which means that each had to update one configuration file.

In terms of network performance, the monolith required an average of 38.11KB of memory on the landing page (-1.56%) and 35.75KB on the product page (-0.60%). The Web Components required 44.85KB (-5.80%) and 42.30KB (-1.54%) on the respective pages. Even the `iframes` showed a decrease in memory usage, with an average of 158.32KB (-6.46%) and 90.21KB (-3.32%) on the landing and product pages, respectively.

The network performance of the applications slightly decreased, however. The monolith transferred a total of 2.0MB (+5.26%) in 701.9ms (+11.21%) on the landing page, and 1.5MB (+0%) in 649.6ms (+11.04%) on the product page. The `iframe` transferred 20.5MB (+1.49%) in 2894ms (+38.07%) on the landing page and 8.1MB (+8.00%) in 2278ms (-2.77%) on the product page.

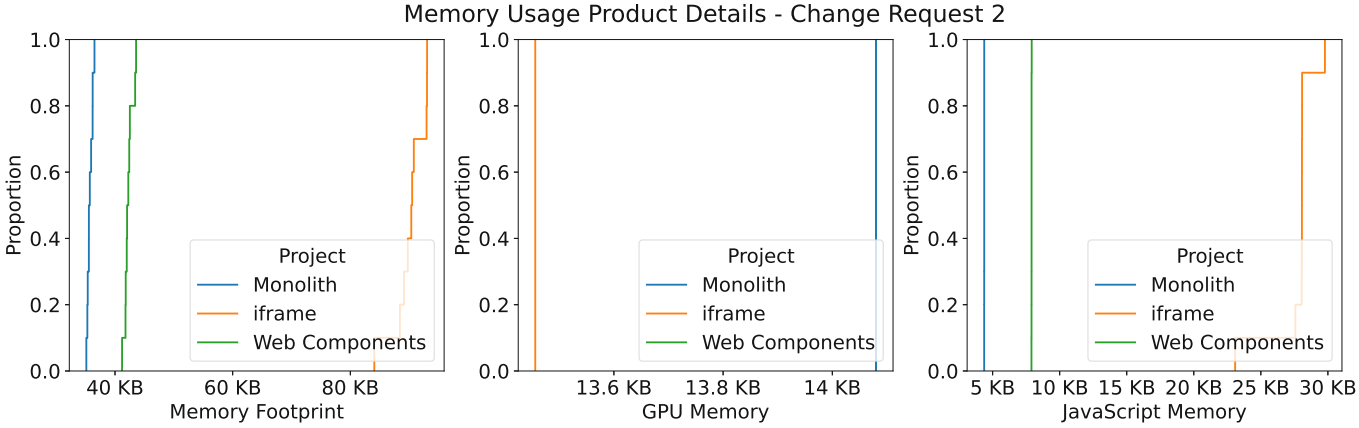
While the decrease in memory consumption is surprising, it could be caused by fluctuations in our measurements or the system's state. Alternatively, using dynamic texts in the components rather than hard-coded strings could allow for a more optimized code, which might allow the browser to free up more memory.



(a) Memory measurements of the landing page after the second change request.

	Memory Footprint (KB)				GPU Memory (KB)				JavaScript Memory (KB)			
	min	max	mean	median	min	max	mean	median	min	max	mean	median
Project												
Monolith	37.62	39.20	38.11	38.05	14.87	14.87	14.87	14.87	4.40	4.40	4.40	4.40
Web Components	43.59	46.20	44.85	44.93	14.87	14.87	14.87	14.87	7.82	7.85	7.84	7.84
iframe	156.14	160.41	158.32	159.03	14.20	14.20	14.20	14.20	58.34	59.89	59.55	59.84

(b) Statistical analysis of the memory usage of the landing page after the second change request.

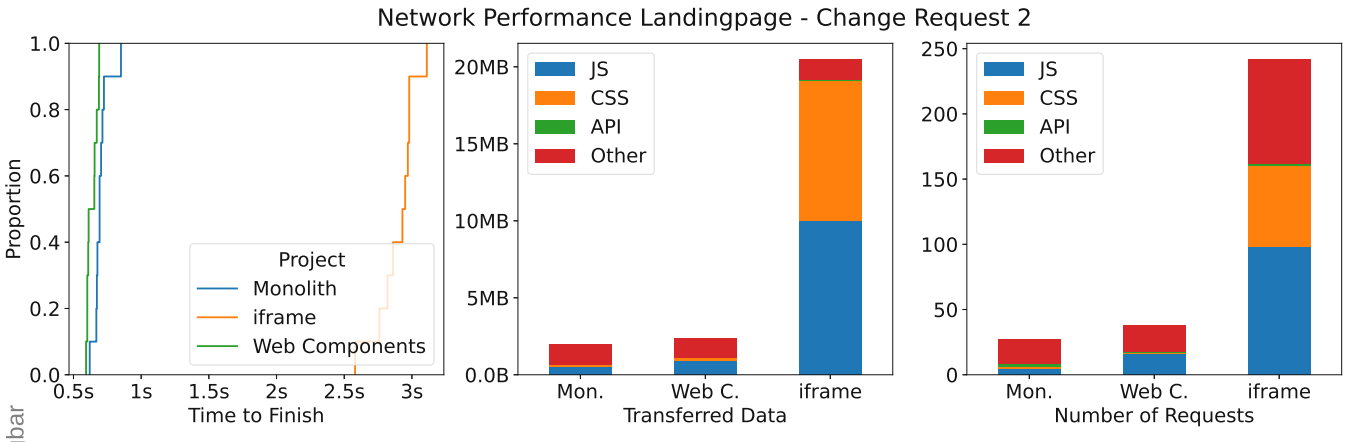


(c) Memory measurements of the product detail page after the second change request.

	Memory Footprint (KB)				GPU Memory (KB)				JavaScript Memory (KB)			
	min	max	mean	median	min	max	mean	median	min	max	mean	median
Project												
Monolith	35.13	36.51	35.75	35.64	14.08	14.08	14.08	14.08	4.37	4.37	4.37	4.37
Web Components	41.20	43.58	42.31	42.16	14.08	14.08	14.08	14.08	7.89	7.91	7.90	7.90
iframe	84.07	93.04	90.21	90.43	13.46	13.46	13.46	13.46	23.09	29.78	27.69	28.06

(d) Statistical analysis of the memory usage of the product detail page baseline after the second change request.

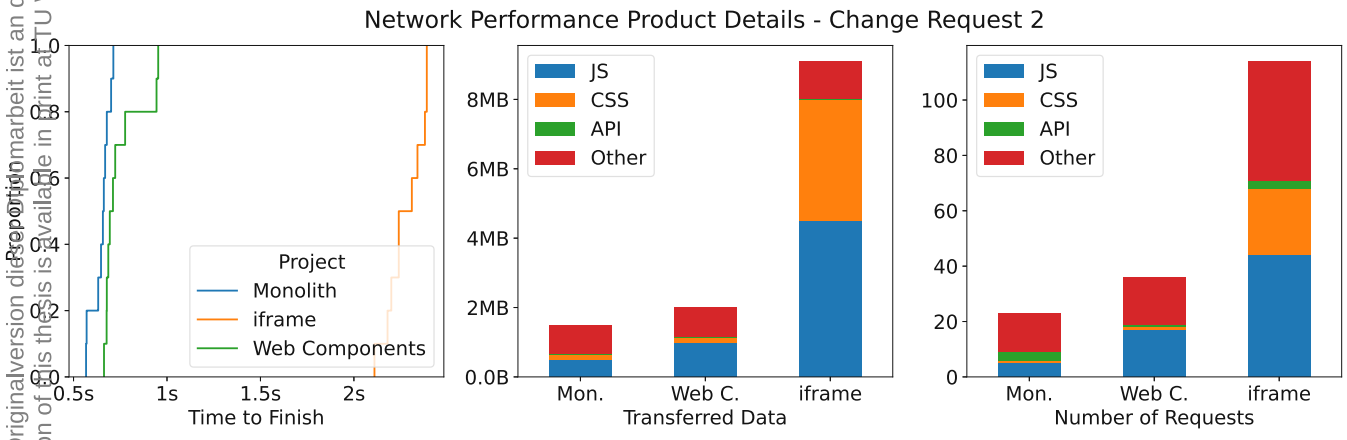
Figure 4.7: Memory measurements after the second change request.



(a) Network performance of the landing page after the second change request.

	Time to Finish				Transferred Data				Number of Requests			
	min	max	mean	median	JS	CSS	API	Total	JS	CSS	API	Total
Project												
Monolith	620ms	851ms	701.9ms	693.0ms	497KB	148KB	50.2KB	2MB	5	1	2	27
iframe	2580ms	3110ms	2894.0ms	2940.0ms	10MB	9.1MB	50.2KB	20.5MB	98	62	2	242
Web Components	593ms	690ms	637.8ms	633.0ms	915KB	149KB	49.8KB	2.4MB	16	1	1	38

(b) Statistical analysis of the network performance of the landing page after the second change request.



(c) Network performance of the product detail page after the second change request.

	Time to Finish				Transferred Data				Number of Requests			
	min	max	mean	median	JS	CSS	API	Total	JS	CSS	API	Total
Project												
Monolith	567ms	713ms	649.6ms	659.5ms	497KB	148KB	19.9KB	1.5MB	5	1	3	23
iframe	2110ms	2390ms	2278.0ms	2275.0ms	4.5MB	3.5MB	19.9KB	9.1MB	44	24	3	114
Web Components	663ms	953ms	750.5ms	702.5ms	988KB	149KB	17.1KB	2MB	17	1	1	36

(d) Statistical analysis of the network performance of the product detail page after the second change request.

Figure 4.8: Network performance after the second change request.

4.3.1 Summary

For the final evaluation of our implementations, we have measured the baseline performance followed by applying two different change requests – one for adding a new *data consumer* and the other to add a new *data provider*. After each change request, we measured their performance to understand the impact of changing requirements to the system.

We have discovered that both the Web Components and the `iframe` add similar properties concerning the implementation overhead of features. Both had to modify 21 files for the first change request. The second change request caused 30 changed files in the `iframe` application and 27 file changes for the Web Components. The monolith needed 10 file changes for the first change request and 30 for the second. Hence, the monolith can be easier to work with, but large refactoring can require about the same effort.

Looking into the performance of the different applications, the monolith clearly shows the lowest performance overhead across all tests – both for the memory and network resources. During our baseline measurements, the `iframes` showed slightly higher performance requirements compared to the Web Components. After the first change request, however, the `iframe` showed a much greater memory footprint and network traffic than both the monolith and Web Components. Since the newly added micro frontend is rendered very frequently, each micro frontend causes a performance-heavy `iframe` to be rendered on the page. After the second change request, however, we didn't observe a large increase in resource consumption in any of the implementations. This shows that the performance overhead of the `iframes` is related to the number of rendered micro frontends, not, however, caused by the communication patterns.

Concluding the performance evaluation, our communication patterns scaled well with new *data consumers* and *data providers*. However, Due to the performance-intensive nature of `iframes`, adding more micro frontends comes at a large cost. As a result, `iframes` are not as suited for micro frontends, but rather "macro" frontends: Larger parts of the application are isolated into separate frames. As seen in the baseline performance measurements, `iframes` can compete with Web Components in those use cases.

However, during the development process, we found a crucial race condition in the Web Component communication channel. It prevented late subscribers to data streams to receive the first event. This unintentional example shows the risks of adding more complexity to a system. Therefore, the "monolith first" principle, often found in the context of microservices, does also apply here [60, 72, 57].

CHAPTER 5

Future Work

Since this thesis is an exploratory work, there are many more ways of achieving cross-communication and state synchronization across micro frontends. For example, `iframes` could include a dedicated application for state management. Each micro frontend then includes this state management `iframe`. Since each state manager `iframe` has the same origin, they could use a similar implementation we developed for the Web Components – namely a `SharedWorker`. We have not explored persistent methods for sharing state information either. Especially in the context of a client-side cache, using persistent methods like the `IndexedDB` or `localstorage` could boost a web page's performance.

Further, we have limited ourselves from using existing micro frontend frameworks such as single-spa [29]. Thus we could research the differences, drawbacks, and advantages of existing solutions to our implementations. This could give insights into the requirements of large-scale frameworks that did not occur in our small-scale experiments.

Since we implemented our solutions in a framework-agnostic way, the next step could be creating a library to integrate the patterns into other applications. This requires us to further test the code, improve the traceability of messages and state, make the code reusable across multiple use cases, and document the usage of the library.

Which leads to integrating the solutions into existing applications. Usually, during the development process of an application new requirements are added and existing requirements might change. Robust communication and state synchronization solutions must withstand those changes in order to reduce developer burden and feature limitations. Thus, to test the viability of our solutions, the patterns should be applied to real-world applications.

Lastly, the web is an ever-evolving standard. Therefore, new APIs might be introduced, which improve the usability or performance of micro frontend communication. So in the future, this thesis may be extended to include updated technologies and patterns.

CHAPTER 6

Conclusion

In the age of distributed computing, backends have become micro instances extracted from their monolithic predecessors. Small units of computations provide single responsibilities and, hence, may need to be recombined in order to provide larger features. While this re-integration of units is common for backends, micro frontends – the frontend counterpart to microservices – are mostly considered for their high isolation properties. However, highly interactive web pages will require micro frontends to work together for more complex features.

In this thesis, we have researched ways for communicating between micro frontends to create richer applications. We applied the Action Research methodology in order to iteratively develop three web shop prototypes. First, we developed a monolith as our baseline. Next, we re-implemented the same application with micro frontends using `iframes` as isolation mechanism. Finally, for the third application we replaced `iframes` with Web Components as micro frontend technology.

We identified two different types of communication, i.e., hierarchical and lateral, which need different approaches to solve them. We use hierarchical communication for passing parameters to a micro frontend and emitting events from one. Both `iframes` and Web Components provide mechanisms to implement them. `iframes` can communicate with their host and vice-versa using the `postMessage` API. Web Components can get notified when attributes on their custom HTML element change and emit custom events for sending data to their host.

To share data between micro frontends, we use lateral communication. In contrast to hierarchical communication, we had to implement custom solutions. `iframes` cannot access any context outside their own. Hence, we had to use hierarchical communication for routing messages to the correct recipients. Web Components, however, are all hosted in the same browser context. This allowed us to add global objects to serve as shared services. To further decouple the components from the provided services, we

moved them to a shared web worker. Commands and data updates are sent using the `BroadcastChannel` API. This allowed us to implement the publish-subscribe pattern within the browser.

To analyze our solutions, we introduced change requests against our three implementations – the Monolith, the `iframes`, and the Web Components. The first change request added a new *data consumer* to the application and the second a new *data provider*. By comparing the changes committed to Git, we found that `iframes` and Web Components have a similar overhead for changes compared to the Monolith. The communications patterns did not need any changes and only had to be extended for the new features. The performance measurements, however, showed that `iframes` do not scale well with the number of micro frontend instances. While the resource overhead was similar during our baseline measurements, after adding a new micro frontend as *data consumer* to the application, the `iframes` required at least 190.9% more memory and had to transfer at least 188.5% more data. In comparison, the Monolith used at least 14.3% more memory and transferred roughly the same amount of data. The Web Components saw a minimum increase of 9.9% in memory consumption and at least a 4.3% increase in transferred data. The second change request did not add any micro frontends, hence the performance did not decrease as drastically.

In conclusion, we have demonstrated the viability of cross-communication between micro frontends using three different implementations of the same application. Our communication patterns using `postMessage` for `iframes` and `BroadcastChannels` for Web Components could be extended beyond the initial feature set with minimal changes required. `iframes`, however, showed that their lack of scalability limits their feasibility in real-world applications.

List of Figures

List of Listings

Glossary

`BroadcastChannel` A browser API for sending messages to multiple recipients. [8]. 35, 51, 71–75, 94, 96

`IndexedDB` A browser API for storing persistent data in a structured database. [18]. 91

`SharedWorker` A browser API for executing tasks in the background on a different thread. A `SharedWorker` is shared across multiple browser contexts (e.g. windows, tabs, `iframes`). A `SharedWorker` is identified via the path to the JavaScript file. Each worker is run as a singleton. [28]. 35, 51, 72–75, 91, 95

`iframe` An HTML element, which allows nesting a website inside another website. The browser creates a new browser context, in which the nested page will be loaded. The nested content is sandboxed and thus has limited access to the hosting page and vice versa. [17]. ix, xi, 10–12, 35–45, 51, 53, 55–57, 64, 65, 67, 69, 71, 72, 75, 77, 79, 81, 83, 84, 87, 90, 91, 93–97

`localstorage` A browser API for storing persistent key-value entries. Values can only be strings. [40]. 91

Action Research A cyclic, iterative, interactive research method. For more details see chapter 3. ix, xi, 3, 4, 16, 17, 25, 35, 93

Angular A JavaScript framework for developing SPAs. Initially released in 2016, it is mainly used for monolithic applications but provides some APIs for Web Component. [2]. 1, 3, 12, 25–27, 30–32, 34, 35, 37, 38, 41–43, 47, 54–58, 60, 61, 67–69, 71, 81, 83, 95, 96

Bootstrap A CSS library. [**?**]. 26

Docker Docker is a tool to build and run isolated containers. Containers are similar to virtual machines but lighter, as they don't try to abstract away the underlying operating system or CPU architecture. [9]. 34

Express A JavaScript library for building REST endpoints in NodeJS. [12]. 26

JavaScript An interpreted scripting language. In the context of web applications, JavaScript is executed in the browser, which allows the creation of dynamic content and interactive elements on the client. However, JavaScript can also be executed in different runtimes and is not limited to the browser. [19]. 9–11, 26, 41, 53–55, 59, 60, 63, 68, 69, 74, 75, 77–79, 81, 97, 98

NodeJS A JavaScript runtime for running code outside the browser. Usually used for server-side applications in the context of web development. [76]. 26, 97

React A JavaScript library for rendering reactive DOM elements from JavaScript. [27]. 1, 25

Svelte A JavaScript framework for building SPAs. [31]. 25

TypeScript A programming language which extends JavaScript with type annotations. [20]. 55, 57, 60, 65

Vue A JavaScript framework for building SPAs. [33]. 1, 25

Web Component An umbrella term for technologies to create reusable, isolated custom elements. [35]. ix, xi, 10, 11, 53–58, 60, 61, 63–65, 67, 69, 77, 79, 81, 83, 84, 87, 90, 91, 93–97

webpack A popular JavaScript bundler. A bundler combines multiple assets into one or more files in order to reduce network overhead for each file individually. Bundling is therefore usually the final step in building a web application for a production-ready build. [37]. 55–62, 65, 70–72, 74, 75, 95, 96

WebSocket A browser API that allows opening and keeping a connection to a server. They enable servers to push messages to the client. [39]. 78

Acronyms

API Application Programming Interface. 2, 6, 8–10, 14, 24–26, 32, 35–37, 41, 47, 58, 71, 72, 75, 78, 91, 93, 94, 96–99

APM Application Performance Management. 8

CSR Client Side Rendering. The client – usually a web browser – handles rendering data to a visual representation – usually HTML.. 99

CSS Cascading Style Sheet. 28, 41, 53, 97

DOM Document Object Model. API for accessing the rendered nodes of the HTML programmatically. [10]. 9–12, 53–55, 65, 79, 98

GPU Graphics Processing Unit. 77–79

HTML Hypertext Markup Language. 5, 6, 9–11, 28, 35, 41, 51, 53, 56, 60–63, 65, 93, 99

HTTP Hypertext Transfer Protocol. 6, 9, 26, 58, 69, 77, 78

RAM Random Access Memory. 77, 79

REST Representational Sate Transfer. A software pattern for defining API endpoints similar to a folder structure. Parameters can and should be passed via path fragments. E.g. `/people/123/pets` represents the endpoint to interact with the pets of the person with ID “123”.. 6, 26, 97

SPA Single Page Application. They are web applications that only consist of a single, static HTML page. The content of the page is generated dynamically by the client. Hence, rendering pages is not done on the server, but in the browser (Client Side Rendering (CSR)).. 3, 6, 24–26, 53, 56, 97, 98

SSI Server Side Includes. 2, 9, 10, 96

SSR Server Side Rendering. The server handles the creation of HTML for a request.. 24

UI User Interface. 2, 9, 39, 74

Bibliography

[1] *Action Research*. Routledge, (Accessed at 2023-01-05). [Online]. Available: https://www.taylorfrancis.com/chapters/edit/10.4324/9781315456539-22/action-research-louis-cohen-lawrence-manion-keith-morrison

[2] Angular. (Accessed at 2019-07-15). [Online]. Available: https://angular.io/

[3] Angular - Angular Elements Overview. (Accessed at 2019-07-21). [Online]. Available: https://angular.io/guide/elements

[4] Angular - Injectable. (Accessed at 2022-07-12). [Online]. Available: https://angular.io/api/core/Injectable#providedIn

[5] @angular-architects/module-federation. npm. (Accessed at 2023-02-15). [Online]. Available: https://www.npmjs.com/package/@angular-architects/module-federation

[6] @angular-architects/module-federation-tools. npm. (Accessed at 2023-02-15). [Online]. Available: https://www.npmjs.com/package/@angular-architects/module-federation-tools

[7] @angular/core. npm. (Accessed at 2019-07-15). [Online]. Available: https://www.npmjs.com/package/@angular/core

[8] Broadcast Channel API - Web APIs | MDN. (Accessed at 2023-01-14). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Worker

[9] Docker: Accelerated, Containerized Application Development. (Accessed at 2023-04-07). [Online]. Available: https://www.docker.com/

[10] Document Object Model (DOM) - Web APIs | MDN. (Accessed at 2023-01-04). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

[11] DOM Standard. (Accessed at 2019-12-17). [Online]. Available: https://dom.spec.whatwg.org/

[12] Express - Node.js web application framework. (Accessed at 2023-01-20). [Online]. Available: https://expressjs.com/

[13] “Facebook/react,” Facebook, (Accessed at 2019-12-03). [Online]. Available: https://github.com/facebook/react

[14] Flexbox - Learn web development | MDN. (Accessed at 2023-02-11). [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Flexbox

[15] HTML 4.0 Specification. (Accessed at 2019-12-17). [Online]. Available: https://www.w3.org/TR/1998/REC-html40-19980424/

[16] HTML Standard. (Accessed at 2019-12-17). [Online]. Available: https://html.spec.whatwg.org/multipage/custom-elements.html

[17] <iframe>: The Inline Frame element - HTML: HyperText Markup Language | MDN. (Accessed at 2023-01-04). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe

[18] IndexedDB API - Web APIs | MDN. (Accessed at 2023-01-15). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

[19] JavaScript | MDN. (Accessed at 2023-01-04). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

[20] JavaScript With Syntax For Types. (Accessed at 2023-01-04). [Online]. Available: https://www.typescriptlang.org/

[21] Made with Angular. Made with Angular. (Accessed at 2019-12-03). [Online]. Available: /

[22] Messaging that just works — RabbitMQ. (Accessed at 2022-07-12). [Online]. Available: https://www.rabbitmq.com/

[23] Micro Frontends. martinfowler.com. (Accessed at 2022-07-04). [Online]. Available: https://martinfowler.com/articles/micro-frontends.html

[24] Micro frontends | Technology Radar | Thoughtworks. (Accessed at 2022-07-04). [Online]. Available: https://www.thoughtworks.com/radar/techniques/micro-frontends

[25] Multi-Framework and -Version Micro Frontends with Module Federation: The Good, the Bad, the Ugly. ANGULARarchitects. (Accessed at 2022-07-12). [Online]. Available: https://www.angulararchitects.io/en/aktuelles/multi-framework-and-version-micro-frontends-with-module-federation-the-good-the-bad-the-ugly/

[26] Queues — RabbitMQ. (Accessed at 2022-07-12). [Online]. Available: https://www.rabbitmq.com/queues.html#names

[27] React – A JavaScript library for building user interfaces. (Accessed at 2023-01-04). [Online]. Available: https://reactjs.org/

[28] SharedWorker - Web APIs | MDN. (Accessed at 2023-01-14). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/SharedWorker

[29] Single-spa | single-spa. (Accessed at 2023-01-14). [Online]. Available: https://single-spa.js.org/

[30] The state of Web Components – Mozilla Hacks - the Web developer blog. Mozilla Hacks – the Web developer blog. (Accessed at 2019-12-17). [Online]. Available: https://hacks.mozilla.org/2015/06/the-state-of-web-components

[31] Svelte • Cybernetically enhanced web apps. (Accessed at 2023-01-04). [Online]. Available: https://svelte.dev/

[32] "Vite-plugin-federation," originjs, (Accessed at 2022-07-12). [Online]. Available: https://github.com/originjs/vite-plugin-federation

[33] Vue.js - The Progressive JavaScript Framework | Vue.js. (Accessed at 2022-07-07). [Online]. Available: https://vuejs.org/

[34] "Vuejs/vue," vuejs, (Accessed at 2022-07-07). [Online]. Available: https://github.com/vuejs/vue

[35] Web components. MDN-Web-Dokumentation. (Accessed at 2019-07-20). [Online]. Available: https://developer.mozilla.org/de/docs/Web/Web_Components

[36] Web Components | MDN. (Accessed at 2023-02-15). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Web_Components

[37] Webpack. webpack. (Accessed at 2023-02-15). [Online]. Available: https://webpack.js.org/

[38] Webpack 5 release (2020-10-10). webpack. (Accessed at 2022-07-12). [Online]. Available: https://webpack.js.org/blog/2020-10-10-webpack-5-release/

[39] WebSocket - Web APIs | MDN. (Accessed at 2023-02-11). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket

[40] Window.localStorage - Web APIs | MDN. (Accessed at 2023-01-15). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage

[41] Zone.js. npm. (Accessed at 2022-07-12). [Online]. Available: https://www.npmjs.com/package/zone.js

[42] A. Abdelhadi and T. Khreis, "The application of action research to enhance the ability of students to conduct project-based research," in *2015 IEEE 7th International Conference on Engineering Education (ICEED)*, pp. 39–42.

[43] B. Albeza. Developer Tools support for Web Components in Firefox 63. Firefox Nightly News. (Accessed at 2022-07-12). [Online]. Available: https://blog.nightly.mozilla.org/2018/09/06/developer-tools-support-for-web-components-in-firefox-63

[44] D. E. Avison, F. Lau, M. D. Myers, and P. A. Nielsen, "Action research," vol. 42, no. 1, pp. 94–97, (Accessed at 2023-01-05). [Online]. Available: https://doi.org/10.1145/291469.291479

[45] D. Banegas and L. Villacañas-de Castro, "A look at ethical issues in action research in education," vol. 3, pp. 58–67.

[46] R. Baskerville and T. Wood-Harper, "A Critical Perspective on Action Research as a Method for Information Systems Research," vol. 11, pp. 235–246.

[47] N. T. Blog. Netflix Likes React. Medium. (Accessed at 2019-12-03). [Online]. Available: https://medium.com/netflix-techblog/netflix-likes-react-509675426db

[48] K. Brown and B. Woolf, "Implementation Patterns for Microservices Architectures," in *Proceedings of the 23rd Conference on Pattern Languages of Programs*, ser. PLoP '16. The Hillside Group, pp. 7:1–7:35, (Accessed at 2019-12-13). [Online]. Available: http://dl.acm.org/citation.cfm?id=3158161.3158170

[49] C. Chen, Y. Tock, and S. Girdzijauskas, "BeaConvey: Co-Design of Overlay and Routing for Topic-based Publish/Subscribe on Small-World Networks," in *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, ser. DEBS '18. Association for Computing Machinery, pp. 64–75, (Accessed at 2023-04-07). [Online]. Available: https://doi.org/10.1145/3210284.3210287

[50] B. Christens, V. Faust, J. Gaddis, P. Inzeo, C. Sarmiento, and S. Sparks, "Action Research," pp. 243–251.

[51] H. B. Christensen, K. M. Hansen, and K. R. Schougaard, "Ready! Set! Go! An Action Research Agenda for Software Architecture Research," in *Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, pp. 257–260.

[52] P. S. M. dos Santos and G. H. Travassos, "Action research use in software engineering: An initial survey," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pp. 414–417.

[53] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds. Springer International Publishing, pp. 195–216, (Accessed at 2019-12-03). [Online]. Available: https://doi.org/10.1007/978-3-319-67425-4_12

[54] J. Ebey, "Esbuild-federation-share-scope," (Accessed at 2022-07-12). [Online]. Available: https://github.com/jacob-ebey/esbuild-federation-share-scope

[55] L. Etxeberria, X. Elkorobarrutia, and G. Sagardui, "Action Research for Improving System Engineering Teaching in Embedded Systems Master," in *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 218–225.

[56] D. R. Ferreira, *Enterprise Systems Integration.* Springer Berlin Heidelberg, (Accessed at 2022-07-12). [Online]. Available: http://link.springer.com/10.1007/978-3-642-40796-3

[57] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From Monolith to Microservices: A Classification of Refactoring Approaches," in *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, ser. Lecture Notes in Computer Science, J.-M. Bruel, M. Mazzara, and B. Meyer, Eds. Springer International Publishing, pp. 128–141.

[58] M. Geers, *Micro Frontends in Action.* Simon and Schuster.

[59] J. Goll, *Entwurfsprinzipien zur Vermeidung von Überflüssigem.* Springer Fachmedien Wiesbaden, pp. 33–42, (Accessed at 2022-07-05). [Online]. Available: http://link.springer.com/10.1007/978-3-658-20055-8_3

[60] D. Gravanis, G. Kakarontzas, and V. Gerogiannis, "You don't need a Microservices Architecture (yet): Monoliths may do the trick," in *2021 2nd European Symposium on Software Engineering*, ser. ESSE 2021. Association for Computing Machinery, pp. 39–44, (Accessed at 2022-12-18). [Online]. Available: https://doi.org/10.1145/3501774.3501780

[61] H. Harms, C. Rogowski, and L. 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*, ser. ESEC/FSE 2017. ACM, pp. 902–907, (Accessed at 2019-07-04). [Online]. Available: http://doi.acm.org/10.1145/3106237.3117775

[62] R. B. N. Jrad, M. D. Ahmed, and D. Sundaram, "Insider Action Design Research a multi-methodological Information Systems research approach," in *2014 IEEE Eighth International Conference on Research Challenges in Information Science (RCIS)*, pp. 1–12.

[63] S. Kang and S. Ryu, "Formal specification of a JavaScript module system," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. Association for Computing Machinery, pp. 621–638, (Accessed at 2022-07-11). [Online]. Available: https://doi.org/10.1145/2384616.2384661

[64] P. Kookarinrat and Y. Temtanapat, "Design and implementation of a decentralized message bus for microservices," in *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pp. 1–6.

[65] B. J. Lando, “Extrahieren von micro-frontends aus einer monolithischen frontend anwendung,” (Accessed at 2023-06-23). [Online]. Available: https://oceanrep.geomar.de/id/eprint/55815/

[66] F. Lau, “A Review on the Use of Action Research in Information Systems Studies,” in *Information Systems and Qualitative Research: Proceedings of the IFIP TC8 WG 8.2 International Conference on Information Systems and Qualitative Research, 31st May–3rd June 1997, Philadelphia, Pennsylvania, USA*, ser. IFIP — The International Federation for Information Processing, A. S. Lee, J. Liebenau, and J. I. DeGross, Eds. Springer US, pp. 31–68, (Accessed at 2023-01-05). [Online]. Available: https://doi.org/10.1007/978-0-387-35309-8_4

[67] D. Leffingwell, *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*. Addison-Wesley Professional.

[68] K. Lewin, *Field Theory in Social Science: Selected Theoretical Papers (Edited by Dorwin Cartwright.)*., ser. Field Theory in Social Science: Selected Theoretical Papers (Edited by Dorwin Cartwright.). Harpers.

[69] T. Lima De Sousa, E. Venson, R. M. Da Costa Figueiredo, R. Ajax Kosloski, and L. C. Miyadaira Ribeiro, “Using Scrum in Outsourced Government Projects: An Action Research,” in *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pp. 5447–5456.

[70] G. Mazlami, J. Cito, and P. Leitner, “Extraction of Microservices from Monolithic Software Architectures,” in *2017 IEEE International Conference on Web Services (ICWS)*, pp. 524–531.

[71] M. Milić and D. Makajić-Nikolić, “Development of a Quality-Based Model for Software Architecture Optimization: A Case Study of Monolith and Microservice Architectures,” vol. 14, no. 9, p. 1824, (Accessed at 2023-01-03). [Online]. Available: https://www.mdpi.com/2073-8994/14/9/1824

[72] F. Montesi, M. Peressotti, and V. Picotti, “Sliceable Monolith: Monolith First, Microservices Later,” in *2021 IEEE International Conference on Services Computing (SCC)*, pp. 364–366.

[73] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. "O’Reilly Media, Inc.".

[74] P. A. Nielsen and J. S. Persson, “IT Business Cases in Local Government: An Action Research Study,” in *2012 45th Hawaii International Conference on System Sciences*, pp. 2208–2217.

[75] Y. Nishizu and T. Kamina, “Implementing Micro Frontends Using Signal-based Web Components,” vol. 30, pp. 505–512.

[76] Node.js. Node.js. Node.js. (Accessed at 2023-01-20). [Online]. Available: https://nodejs.org/en/

[77] S. Peltonen, L. Mezzalira, and D. Taibi, "Motivations, benefits, and issues for adopting Micro-Frontends: A Multivocal Literature Review," vol. 136, p. 106571, (Accessed at 2022-07-07). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921000549

[78] Y. R. Prajwal, J. V. Parekh, and D. R. Shettar, "A Brief Review of Micro-frontends," vol. 02, no. 08, p. 4.

[79] T. Preston-Werner. Semantic versioning 2.0.0. Semantic Versioning. (Accessed at 2022-07-12). [Online]. Available: https://semver.org/lang/de/

[80] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic Versioning versus Breaking Changes: A Study of the Maven Repository," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pp. 215–224.

[81] P. Reason, "Choice and Quality in Action Research Practice," vol. 15, pp. 187–203.

[82] C. Rojas, *Building Native Web Components*. Apress, Berkeley, CA, (Accessed at 2022-07-05). [Online]. Available: https://link.springer.com/book/10.1007/978-1-4842-5905-4

[83] T. Savage, "Componentizing the web," vol. 58, no. 11, pp. 55–61, (Accessed at 2022-07-12). [Online]. Available: https://dl.acm.org/doi/10.1145/2814338

[84] J. F. Shoch, "An overview of the programming language Smalltalk-72," vol. 14, no. 9, pp. 64–73, (Accessed at 2022-07-12). [Online]. Available: https://dl.acm.org/doi/10.1145/988113.988122

[85] A. Singleton, "The Economics of Microservices," vol. 3, no. 5, pp. 16–20.

[86] G. I. Susman and R. D. Evered, "An Assessment of the Scientific Merits of Action Research," vol. 23, no. 4, pp. 582–603, (Accessed at 2023-06-24). [Online]. Available: https://www.jstor.org/stable/2392581

[87] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural Patterns for Microservices: A Systematic Mapping Study:," in *Proceedings of the 8th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, pp. 221–232, (Accessed at 2019-07-01). [Online]. Available: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006798302210232

[88] D. Taibi and L. Mezzalira, "Micro-Frontends: Principles, Implementations, and Pitfalls," vol. 47, no. 4, pp. 25–29, (Accessed at 2023-06-23). [Online]. Available: https://dl.acm.org/doi/10.1145/3561846.3561853

[89] P. Y. Tilak, V. Yadav, S. D. Dharmendra, and N. Bolloju, "A platform for enhancing application developer productivity using microservices and micro-frontends," in *2020 IEEE-HYDCON*, pp. 1–4.

[90] C. Yang, C. Liu, and Z. Su, "Research and Application of Micro Frontends," vol. 490, no. 6, p. 062082, (Accessed at 2023-06-23). [Online]. Available: https://dx.doi.org/10.1088/1757-899X/490/6/062082