Evaluating Micro Frontend Approaches for Code Reusability

Emilija Stefanovska and Vladimir Trajkovik(✉)

Faculty of Computer Science and Engineering, Skopje, North Macedonia
emilija.stefanovska@students.finki.ukim.mk,
trvlado@finki.ukim.mk

Abstract. The term micro frontend is relatively new, and it is a continuation of the microservice architecture on the client-side. This paper researches the possibility to use micro frontend architecture for frontend code reusability. The goal of this paper is to achieve code reusability by providing a good organizational structure by using a micro frontend approach. Reusing the code should help to increase the time to market and allow for better scalability. The paper starts by overviewing the base characteristics of domain-driven design as one of the key concepts behind micro frontend architecture. It continues with evaluating existing micro frontend architectures by a set of defined qualitative attributes. Lastly, it uses the findings from the evaluation in building a technical solution. The implementation process can be divided into two phases: decomposition of an existing frontend application; reusing the code by integrating the decomposed components in a new micro frontend application. The results of the implementation process should confirm the micro frontend approach for code reusability.

Keywords: Micro frontend · Code reusability · Domain-driven design · Web components · Module Federation

1 Introduction

From the appearance of the internet in 1990 until today, web technologies are constantly developing. With the increased usage of websites, the complexity of backend applications also increased [1]. Continuously adding new functionalities increases the complexity and size of the application and slows down the testing and delivery processes. Such an application becomes difficult to maintain or change, but it also becomes difficult to understand its domain logic [2]. One solution to these problems is to split the backend application into smaller independent parts that will be developed and delivered independently and together will function as one application called micro-services [3]. One of the main challenges of the micro-service architecture is how to divide the application into smaller independent parts. One way to split the application is with the help of a discipline called domain-driven design (DDD). DDD is one of the base concepts behind micro-service architecture. It provides methods and tools for structuring applications with a complex domain and can help in defining the microservices [4, 5].

K. Zdravkova and L. Basnarkov (Eds.): ICT Innovations 2022, CCIS 1740, pp. 93–106, 2022.
https://doi.org/10.1007/978-3-031-22792-9_8

Disadvantages faced by large monolithic backend applications also apply to client-side applications. Today many organizations are facing the problem of unmaintainable code on the client-side. The size of the code base doesn't allow to easily add new functionalities, change technology, or the development framework to follow the fast-growing technology trend. Many organizations come to a point where they want to replace their existing client-side application with a new, modern, and faster application to be more concurrent on the market. Rewriting the application's code is one potential solution to the problem but for many organizations, this proved to be a strategic mistake [6, 7]. Another solution to the problem is to continue the split of the backend application to the client-side application and this is the main idea behind the micro frontend architecture [8]. Each micro frontend application can be responsible for a single business domain and can be developed, tested, and delivered independently from other micro frontend applications. This organization should keep the code bases small so it's easier to understand the domain logic [9]. Hence, this research will explore the possibility of using a micro frontend approach for achieving code reusability. The goal of this paper is to achieve code reusability by providing a good organizational structure with micro frontend architecture. Code reusability has the potential of speeding the time to market but also providing modularity and scalability in the new application.

This research paper consists of six sections. The first section is the introduction. It gives a historical overview of the research problem and defines the goal of the paper. The second section of the paper investigates domain-driven design. The third section should serve as a technical background for understanding the technical decisions in the implementation process. It gives an overview of the micro frontend architecture and concludes by evaluating the different micro frontend composition types. The fourth section is the practical part of this paper. The implementation process is split into two phases. The first phase looks at decomposition approaches while the second phase provides two use cases for building a micro frontend application. The fifth section presents the results of a questionnaire conducted among software engineers experienced in micro frontend development. The sixth last section summarizes the results of the implementation process.

2 Domain-Driven Design

Every software is designed and developed to execute some activity that is of interest to its users. The area of this activity is the domain of the system. To build any software, we must first understand its domain and if this domain is complex the amount of information can become huge [10]. To better maintain a complex system, it's best to divide it into smaller, less complex, and well-defined logical components. The components should be decoupled from each other, and the dependencies should be kept to a minimum [11]. This should make the navigation through the codebase easier for the developers. For this reason, the process of adding new features or resolving bugs should be faster and more efficient. Domain-driven design is a discipline that gives directions on how to structure and divide a complex system into smaller components.

Domain-driven design can be separated into two disciplines: model-driven design and strategic-driven design [11]. In some literature, the model-driven design is also called

tactical-driven design, but this term does not originally come from E. Evan's literature. The model-driven design is oriented towards code implementation and gives directions on how to better structure the code. Some examples include using layered architecture, services, and entities. On the other hand, strategic-driven design is oriented toward the architecture of complex applications. The main goal of the strategic-driven design is to identify and divide the domain of the application into subdomains. To define the subdomains, there are three base concepts we must understand first:

- Bounded context. Represents a logical boundary between two domains, and it is intended to hide the implementation details between them.
- Ubiquitous language. Should be used by both technical and domain experts as a common language that connects everyday communication to the code implementation.
- Context map. A context map gives a picture of all the bounded contexts that exist in a system. It also shows the points of interaction between two bounded contexts [10].

Both modal-driven and strategic-driven design will be of further interest to this paper. The base of the strategic-driven design was to define the subdomains of the system. To define the subdomains, we must first look at the processes the system executes. As an example of defining subdomains, we implemented a prototype application that represents a simplified version of a banking system. It executes two simple processes which will be referred to as P1 and P2. Let's say process P1 is connected to all operations involved with banking cards while P2 is with bank offices. The simplest heuristic for defining the sub-domains in this example is vocabulary. Now if we consider the vocabulary, we can identify two subdomains which will be referred to as D1 and D2. Now each of these subdomains can be modeled separately.

The application is implemented as a standard web application consisting of a back-end part implemented in Java Spring Boot and a single-page client application implemented in Angular. Since this research is more oriented towards frontend architecture and development, we will focus on the client application. It's important to note that not all domain-driven principles are applicable to client applications although some of them have a fundamental meaning. The client application already follows some of the principles and implements processes P1 and P2 as two bounded modules. The next few chapters will research different micro frontend architectures and investigate how to organize and structure the code from the two subdomains and reuse it in a new micro frontend application.

3 Background

3.1 Micro Frontends

Micro frontend architecture is an architecture for building client-side applications. This architecture suggests dividing the client-side application into smaller front-end applications that together work as one application for the end-user.

Before starting a micro frontend project there are several decisions that need to be made. These decisions will determine the future course of the project. In his literature, Mezzalira puts these decisions in a so-called micro frontend decision framework

which consists of four parts: definition of micro frontend, composition, routing, and communication [12].

The first part is about defining what a micro frontend is in the context of the application being built. The application can show multiple micro frontends on one screen or one micro frontend per screen. With this definition, Mezzalira divides the micro frontend architecture into two parts: horizontal and vertical micro frontend architecture. A horizontal micro frontend application allows multiple micro frontends on the same screen. This indicates that two or more teams can be responsible for one screen. This organization requires more coordination between the teams to have consistent design decisions. The micro frontends might also require communication to share information about the user interaction. A vertical micro frontend application shows one application per screen. This indicates that every team is responsible for one business domain. The vertical micro frontend application is tightly coupled with the domain-driven design that can be used in defining the micro frontends.

The second part of the framework refers to the integration of the micro frontends into one composition. In the literature [9, 12–14], there are three types of compositions: server-side, edge-side, and client-side composition.

The third part of the framework is about routing between the micro frontends, and it's tightly coupled to the composition type. If the application is composed on the server side the routing must be done on the server side. As opposed to this if the application is composed on the client side the application routing will be done by the client application.

The last part of the framework is about the communication between the micro frontends. In an ideal case, the micro frontend applications should be completely independent and wouldn't have a need to communicate with each other. This is especially true for the vertical micro frontend architecture. Some possible communication solutions include custom events, web storage, and query strings [12].

The micro frontend architecture comes with many benefits like small code bases, autonomous teams, scalable code, and fault isolation [9, 12]. On the other hand, it introduces other issues like code redundancy, routing, and communication between the applications. Because of this, not every application is suitable to follow the micro frontend approach.

3.2 Micro Frontend Composition Types

As previously mentioned, there are three micro frontend composition types. According to Mezzalira L., only the client-side composition is suitable for a vertical micro frontend because it gives the closest experience to a SPA. All other three composition types can be used for the horizontal micro frontends [12]. Authors Jackson K. and Geers M. additionally divide the client-side composition into build-time and runtime client-side composition [9, 13].

This section will give an overview of all three composition types with some of their benefits and common issues and will serve as a background for understanding the technical decisions made during the implementation process.

Server-Side Composition
The composition of the final HTML page is done by a server that can be a simple proxy server like Nginx or a custom application with additional logic. This server is located between the micro frontend application servers and the client or ideally a CDN [12]. The main benefit of this approach is the performance. The server can use caching and additional logic to decrease the calls to the micro frontend servers which can result in a short loading time of the pages. Ideally, all servers would be in the same data center where the network latency is much smaller. The biggest issue with this approach is the client experience when interacting with the application. If this composition is not combined with client-side rendering, then every interaction will require a full page reload [12, 13].

Edge-Side Composition
Edge-side composition is enabled by the Edge side integration language (ESI). ESI is a specification defined by Oracle as one of the co-authors and allows composing an HTML page from HTML fragments [15]. The integration is done on the CDN level like Akamai or AWS Lambda [16]. The content delivery networks allow one resource to be found in multiple locations and assure that the requested resource will be served from the point that's closest to the user. This results in smaller network latency and hence faster load time for the end-user.

Like the server-side composition, this composition also lacks a good user experience if not combined with a client-side composition.

Client-Side Composition
With this type of composition, the final HTML page is composed directly in the client's browser. A micro frontend integrated on the client-side usually consists of several micro frontends and a container application. The container application is responsible for selecting the correct micro frontend and often contains the common application parts like navigation and page footers. The concept is shown in Fig. 1.

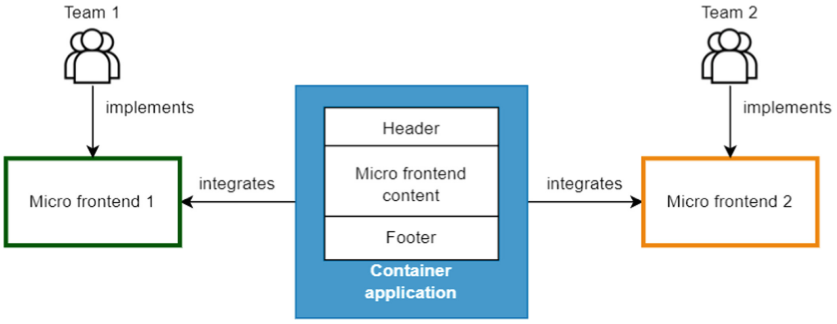

Fig. 1. Client-side composition

Because of the popularity of frontend frameworks like React, Angular, and Vue, the client-side composition is one of the most frequently used micro frontend composition types.

Iframes
Iframes can be defined as a special HTML element that allows the loading of a second HTML document on the same HTML page [17]. Iframes can be considered as one of the oldest ways of building micro frontends. They provide a high level of isolation and the JavaScript and CSS code inside the iframe cannot be affected by any changes in the container application. However, loading many iframes on the same page in the browser can cause serious performance issues [13].

Web Components
Web components represent a set of APIs which allow the building of custom and reusable HTML elements. Today, almost every modern frontend framework like React and Angular allows the usage and creation of web components. There are two main technologies that allow building web components: custom elements and shadow DOM. Custom elements allow the definition of custom HTML elements while the shadow DOM provides the encapsulation of web components [18].

Isolation is one of the main benefits of web components. It allows the integration of micro frontend applications developed in different frontend frameworks. Because of this fact they are a good choice for code reusability. The old code can be encapsulated inside a web component and integrated into a new application.

Module Federation
Module Federation is a relatively new technology initially released in 2020 [19]. It is a plugin of Webpack (a tool designed to bundle JavaScript applications). It allows async loading of JavaScript bundles at runtime so the application can be built by remote and independent modules [20]. The plugin distinguished two main application concepts called remote and host. The host is the container application that integrates the remote applications at runtime while the remotes are the micro frontends that export their application code to the host.

Module Federation as described by the author was built to solve the many problems that micro frontends are facing like routing or common dependencies. One of the major benefits of Module Federation is the ability to define common dependencies between micro frontends [21].

Build Time Integration
Build time integration like other composition types includes a container application that integrates other micro frontend applications. The difference from other client-side composition types is that this integration happens at build time. This means that the micro frontends are defined as any other dependencies of the container application. According to author Jackson [9], this approach contradicts the whole concept of micro frontends to have independent code bases that are built and deployed separately. With build time integration releasing one micro frontend application requires adapting the version in the container application and releasing the container application. Because of this coupling at build time, it might be easier to have the micro frontends as separate modules as it will decrease the complexity of setting up and maintaining multiple applications.

3.3 Evaluation of Micro Frontend Composition Types

This section evaluates the micro frontend composition types bases on the literature findings. To evaluate any software architecture there are several standardized quality attributes to be considered. At the same time, these quality attributes should support the business goals of the project [22, 23]. One major business goal when reusing the old code is to have a faster time to market. For this reason, we can consider the simplicity of the architecture and the development experience as key factors [24, 25]. The old code can be complex and hard to understand. During its lifetime it can be that many people even teams changed. To reuse the old code, it is important that the architecture is simple so that the team working on the old application doesn't have to make a lot of changes to the code base. Simplicity and experience with technology also support the development experience (Table 1).

Table 1. Evaluation of micro frontend composition types.

Qualitative attribute	Micro frontend composition types
Performance	Server-side, web components, Module Federation, build-time
Modularity	Server-side, edge-side, iframes, web components, Module Federation
Testability	iframes, web components, Module Federation, build-time
Developer experience (DX)	Web components, Module Federation, build-time
Simplicity	iframes, web components, Module Federation, build-time
Scalability	Server-side, edge-side, web components, Module Federation

Unlike other composition types, web components and Module Federation are two technologies which fulfill all qualitative attributes. For this reason, they both will be considered when building the technical solution.

4 Implementation Process

The implementation process can be divided into two phases. The first phase includes splitting the code from the existing single-page application into micro frontends. The goal of the decomposition is to provide an organizational structure that will enable us to reuse the functionalities of the application into a new micro frontend application. In this phase, we will investigate two decomposition approaches and compare them.

The second phase of the implementation process uses the result from the comparison in the first phase and integrates the decomposed components into a container application. In this phase, we will provide two use cases of code reusability with web components and Module Federation.

Both phases of the implementation process can be started in parallel. For example, while one team works on the decomposition of the old application a new team can start with building the new micro frontend application and extending it with new functionalities.

4.1 Decomposition Phase

Simulated Decomposition Using Web Components

One way of decomposing the application is to virtually split the independent domain parts of the application with the help of web components. With this type of decomposition, each of the domain modules will only simulate an independent application. We will still use the original SPA and its structure will remain the same.

Every standard Angular SPA has one root component which is the first rendered component when the application launches. This component usually contains a router-outlet directive that tells the Angular router where to render the content of the component for the given route [26]. Because the idea of this simulation is to have every domain module behaving as an independent application, every module will have its own root or entry component which will be used to render the content of the module's components. This component must be defined as a web component so it can be recognized by the browsers as an HTML tag. Then, as soon as the web component is used, the content from the corresponding domain will be rendered. Figure 2 shows the structure of the application before and after the decomposition.

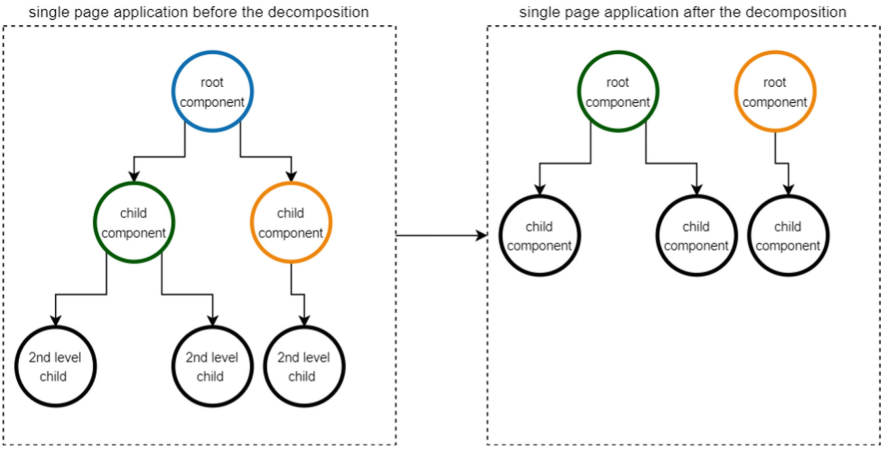

Fig. 2. Structure of the single page application before and after the decomposition.

One important issue to address in this approach is application routing. Since there is an entry web component for each module now, the root Angular component was removed. This means that initially when the micro frontend application is launched no component will be rendered by default. Even more, the end-user will initially interact with the container application. Until the container loads the selected micro frontend its router won't be initialized and therefore register changes. This is the reason why we needed to introduce programmatic routing. The routing by the micro frontend will be done when the parent web component is created.

Decomposition to Separate Applications Using Web Components

The second way of decomposition is to split the domain modules into two independent applications which will be built, tested, and deployed independently. The simplicity of this decoupling largely depends on the current application implementation. If there is a tight coupling between the modules the split can be almost impossible since many components would need to be reworked or even rewritten. However, if the application follows DDD principles and every domain is bounded to its own module the split can be done relatively easily and each module can be moved to a separate application. Figure 3 illustrates the decomposition of a SPA into two separate applications.

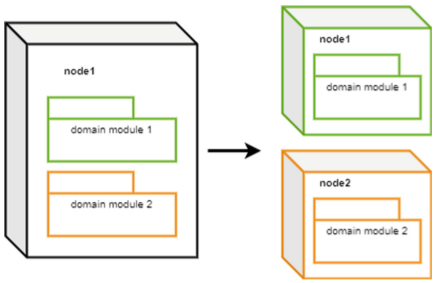

Fig. 3. Splitting the domain packages of an application into separate running applications.

The first step of this decomposition would be to create a project for each module. Now each application would need to install its own dependencies to work properly. Depending on the composition type that will be used there are certain changes that need to be made to each application. If the web component composition type is used, then the root component of each application should be transformed as a web component in the same way as it was described in the previous section.

In the previous decomposition approach, the web component class used programmatic routing. This decomposition approach also needs to implement the routing programmatically. The main reason is the control over each micro frontend router in the container application. The second reason is deep links. All further details will be explained in use case 1 of the integration phase.

Comparison

Table 2 compares the two decomposition approaches based on some of the fundamental micro frontend principles.

The table clearly shows that the second decomposition approach has all the benefits of the micro frontend architecture and for this it will be used for building the new micro frontend application.

Table 2. Comparison of decomposition approaches

Micro frontend principles	Simulated decomposition using web components	Decomposition to separate applications
Independent delivery	False	True
Independent technologies	False	True
Autonomous teams	False	True
Autonomous code bases	False	True
Organized around business capabilities	True	True
Scalability	True	True

4.2 Integration Phase

This section will present the integration phase of the implementation process. This phase includes building a container application and integrating the micro frontend applications we introduced in the decomposition phase. The result of this phase should confirm or deny the suggested organizational approach for code reusability.

Use Case 1

The first use case for composing a micro frontend will use web components as one composition type with most benefits.

Due to the popularity of the development framework and our experience with it, the container application will be implemented as a single-page application in Angular. For navigation, the application will use the built-in Angular router. The router matches the browser's URL to a corresponding component, and it expects this component to be a part of the application. Because the corresponding component is in one of the remote micro frontends a helper component had to be implemented. Now every route matches the helper component.

The helper component holds the main logic for selecting the micro frontends. The component holds a configuration map for each micro frontend. The map defines which micro frontend should be loaded for a given path and the server location to loaded from. When the helper component is instantiated, it first checks if the micro frontend is already loaded in the browser by using a special id attribute. If not, it adds the micro frontend bundled application scripts to the HTML page.

This setup of the container application required certain adaptations to the micro frontends. One major problem that occurred with this solution was the routing. The container application is only responsible for the external routing between the micro frontends. It only looks for the first part of the URL to select the correct micro frontend application. After this, the router of the selected micro frontend looks at the full URL to find the correct page inside the micro frontend. Figure 4 illustrates how this works.

When a micro frontend is launched it initializes its own router which listens to changes until the web component is destroyed. Because the container application doesn't get notified when the micro frontend web component is destroyed, in the meantime it

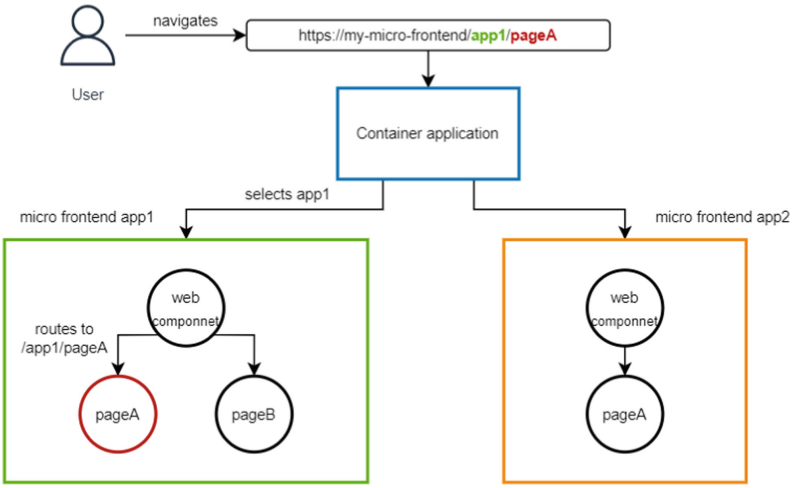

Fig. 4. Routing in the micro frontend application.

can launch a second micro frontend. This causes the router of the first micro frontend to fail with an error because of an unknown route. A solution to the problem was to ensure that only one micro frontend router is active at a time. This can be achieved through conditionally displaying the router-outlet directive which was used to tell the router where to load the content. When the web component is created or when the browsers back and forward buttons are selected, it will check the first part of the URL. If the path is relevant to the micro frontend, it will display the router-outlet directive and render the correct content.

Use Case 2

This use case for composing a micro frontend will use Module Federation as the second composition type with the most benefits.

The container application will be implemented as a single-page application in Angular. One way to integrate Module Federation into the application is to add it as a project dependency. The next step is to define a new webpack.config.js file. This file will contain all the configuration needed to fetch and load the micro frontends. In comparison with the previous solution, the code needed to fetch and load the micro frontend bundles is isolated in one configuration file. Some of the configurations include the resource location of the micro frontends, common dependencies, and versioning of common packages.

The web component composition type we discussed in the previous section required certain adaptations to the micro frontend applications and routing logic was one of them. With this approach, these changes are not needed. This way the application structure remains almost identical to the original application.

For the container application to load the two micro frontends, they should be configured to use Module Federation for exposing the application code. For this reason, Module Federation is added as a project dependency in both micro frontends. The second step would be to configure Module Federation by adding a webpack.config.js file.

The configuration includes the modules which are being exported as well as shared dependencies.

5 Evaluation

The implementation process proves that micro frontends can indeed be used to better organize the code of an existing application and successfully reuse it in a new environment which should result in a faster time to market. However, the application used in the implementation process is a small proof-of-concept application that has a relatively small code base and independent domain parts which made the decomposition of the domain parts faster compared to rewriting the application code.

In this part, we conducted a questionnaire among software engineers. The goal is to compare the knowledge gained from the implementation phase with the knowledge from real-world experience with micro frontends. It's worth mentioning that all participants have experience with micro frontends and work for different companies or projects. The interviewees were asked about the main benefits and downsides of using micro frontends and to provide the use cases where they used a micro frontend application instead of a monolith. Almost all interviewed candidates (80%) answered that the main benefit of using a micro frontend is maintainability due to smaller code bases. The same group thinks that independent deployments and teams are also a major benefit. Having teams that are responsible for their own release cycles and are deploying to production when they are ready, regardless of other teams is a way to achieve faster time to market. However, independent deployments are only possible when the applications are completely independent, or the communication interfaces are not affected by the release changes. Half of the interviewees answered that developing a micro frontend was faster due to the smaller code bases which seems to confirm the initial hypothesis.

One of the most mentioned downsides of micro frontend was implementing communication mechanisms between the micro frontend applications as well as the initial infrastructure setup which can be complex.

From the obtained responses based on real-world experience with micro frontends, it seems that micro frontends provide faster time to market.

6 Conclusion

This paper investigates the possibility of using a micro frontend architecture to achieve code reusability. The research evaluates the possible micro frontend approaches found in the literature and discusses the technologies that can be used. Based on the literature evaluation, as part of the research, a technical solution was implemented. The technical solution shows how micro frontend architecture can be used to better organize the code of an existing single-page frontend application and how the code can be reused in a new environment. The paper provides two practical use cases of code reusability using two different technologies. The first use case uses web components as one technology for building a micro frontend application. It describes how a web component can serve as a shell for encapsulating a whole frontend application. The second use case uses the

Module Federation plugin for building a micro frontend. This use case shows a different approach that handles the integration logic on the configuration level.

Both technical implementations have many benefits. In terms of modularity, both solutions integrate the micro frontends on the client-side at runtime which makes them highly modular. At the same time, both solutions are highly scalable since the container application fetches the micro frontends as static JavaScript files. Based on the results of the implementation process it can be concluded that using a micro frontend approach for code reusability was successful.

One of the main benefits of using a micro frontend approach for code reusability was to provide a faster time to market. This research puts the main emphasis on the implementation process but does not provide any data about the complexity or time needed to set up deployment or testing processes also mentioned in the responses from the questionnaire. Having a decentralized architecture requires having multiple servers, deployment pipelines, and additional monitoring tools. This requires scaling the release and deployment processes to support multiple applications and so it introduces a new type of complexity. Future work can evaluate the available strategies for deploying micro frontends and investigate how the effort impacts the overall time to market.

References

1. Roesler, V., Barrére, E., Willrich, R.: Special Topics in Multimedia, IoT and Web Technologies, 1st edn. Springer, Switzerland (2020). https://doi.org/10.1007/978-3-030-35102-1
2. Article on monolithic architecture patterns. https://microservices.io/patterns/monolithic.html. Accessed 11 June 2022
3. Farcic, V.: The DevOps 2.0 Toolkit, 1st edn. Leanpub, Victoria (2016)
4. Article on microservices. https://martinfowler.com/articles/microservices.html. Accessed 11 June 2022
5. Newman, S.: Building Microservices, 1st edn. O'Reilly, Sebastopol (2015)
6. Blog post. https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i. Accessed 13 June 2022
7. Medium article. https://medium.com/@herbcaudill/lessons-from-6-software-rewrite-stories-635e4c8f7c22. Accessed 13 June 2022
8. Micro Frontends Homepage. https://micro-frontends.org. Accessed 13 June 2022
9. Article on micro frontends. https://www.martinfowler.com/articles/micro-frontends.html. Accessed 13 June 2022
10. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software, 1st edn. Addison Wesley, Boston (2003)
11. Manfred, S.: Enterprise Angular - DDD, Nx Monorepos and Micro Frontends, 4th edn. Lean Pub, Victoria (2020)
12. Mezzalira, L.: Building Micro-Frontends, 1st edn. O'Reilly Media, Sebastopol (2021)
13. Geers, M.: Micro Frontends in Action, 1st edn. Manning Publications, Shelter Island (2020)
14. Rappl, F.: The Art of Micro Frontends, 1st edn. Packt, Birmingham (2021)
15. W3Schoos. https://www.w3.org/TR/esi-lang/. Accessed 17 June 2022
16. The future of micro frontends. https://betterprogramming.pub/the-future-of-micro-frontends-2f527f97d506. Accessed 07 June 2022
17. W3Schoos. https://www.w3schools.com/tags/tag_iframe.asp. Accessed 08 June 2022

18. Web Components Introduction. https://www.webcomponents.org/introduction. Accessed 08 June 2022
19. Micro-frontends building blocks: Webpack Module Federation. https://dev.to/aws-builders/micro-frontends-building-blocks-webpack-module-federation-360a. Accessed 14 June 2022
20. Webpack Module Federation. https://webpack.js.org/concepts/module-federation. Accessed 13 June 2022
21. The Micro frontend Revolution: Module Federation with Angular. https://www.angulararchitects.io/en/aktuelles/the-microfrontend-revolution-part-2-module-federation-with-angular/. Accessed 14 June 2022
22. Carnegie Mellon University Library. https://resources.sei.cmu.edu/asset_files/Webinar/2009_018_101_22232.pdf. Accessed 15 June 2022
23. ISO 25010. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010. Accessed 15 June 2022
24. Medium article. https://medium.datadriveninvestor.com/the-case-for-favoring-simplicity-in-software-49fa9caf8da#:~:text=The%20case%20for%20favoring%20any,to%20the%20most%20important%20first. Accessed 15 June 2022
25. Good Developer Experience Practices. https://developerexperience.io/practices/good-developer-experience. Accessed 15 June 2022
26. Angular Bootstrapping Guide. https://angular.io/guide/bootstrapping. Accessed 15 June 2022