2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C) | 978-1-6654-6459-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSA-C57050.2023.00025

Experiences on a Frameworkless Micro-Frontend Architecture in a Small Organization

1st Jouni Männistö
SimAnalytics Oy
Helsinki, Finland
jouni.mannisto@simanalytics.com

2nd Antti-Pekka Tuovinen
University of Helsinki
Helsinki, Finland
0000-0002-1092-4157

3rd Mikko Raatikainen
University of Helsinki
Helsinki, Finland
0000-0002-2410-0722

***Abstract*—Micro-frontend (MFE) architecture for a web application aims at doing the same for a monolithic user interface (UI) that microservices do for a monolithic backend: it decomposes the UI into self-contained components that can be developed, deployed, and provisioned together with their associated backend service. This paper represents a small team's experiences at the Visma company transforming its monolithic UI into a Micro-frontend solution. We describe the motivations, the key design decisions based on Web Component technologies, the design and implementation process including ATAM-based architectural assessment, and our learnings from the case. The key takeaways are that (1) the motivations for MFE concerned improving customer-specific configurability and lowering the related costs rather than enabling team independence, (2) the APIs provided by web standards — and the Web Components API based on them — offered a competitive alternative for JavaScript frameworks avoiding many framework-induced problems, and (3) small organizations without a large number of feature teams can benefit from Micro-frontend architectures.**

***Index Terms*—Micro-frontend, Microservice, Software architecture, Software Architecture Assessment**

I. Introduction

Microservices and *microservice architecture* have become mainstream for web application development [1]–[3]. Short release cycles, effective use of independent teams working in parallel, and uncomplicated developer workflows are common motivations for adopting microservices [1]. Microservices are not just for new development; existing applications are also being transformed to follow this architectural style.

The evolution of an existing web application towards microservices usually starts from the backend, which is gradually broken down into small and independently deployable units of functionality. However, the UI (user interface) can remain as one large entity because, typically, the applied UI frameworks are not designed to decompose the UI into small independent units that could be deployed with the associated microservices. Consequently, in microservice architectures, the frontend (or data [4]) can be the new monolith, which has complex dependencies, has shared responsibility, and is hard to change.

A monolithic UI was the case in 2018 in the *Resident Portal* web application by Visma[1]. The backend had already been partitioned as microservices that UI accessed through an API. Breaking up the monolithic UI was going to be "the next step" in the evolution of the application's architecture towards truly independent vertical partitions of features spanning the UI, business logic, and data storage. This is the context for the architectural work and experiences we report.

Recently, *micro-frontend* (MFE) architecture has emerged as a comparative approach and term to our experiences in dividing a web application's UI into separately manageable and deployable pieces. The earliest notable mentions of MFE architecture are found in blogs and articles by companies' tech teams [5]–[7]. A multivocal synthesis focusing on the motivations, benefits, and issues (*MBI*) in adopting MFEs is now available [8]. However, since industrial experiences are still scarce, we address the following research problem: *How is MFE applicable in small organizations?* Our case brings forward an account of a small organization's technological solutions and experiences that may help other practitioners assess their designs and implementations.

In this paper, we first elaborate on the background and concepts of the MFE architecture. Next, we describe the Resident Portal case and discuss the goals for the renewal. Then, we describe the architecture design and evaluation. Finally, we summarize the key learnings and experiences from the case and compare them with the MBI framework in [8].

II. Background: MFEs

MFE was a very novel topic when the Resident Portal case started in 2018, although much had been written about microservices [2], [9]–[12]. To our knowledge, the frontend side began to gain attention in the literature on microservices only in 2017. However, the term "micro-frontend" was not yet coined, but the discussion used general terms, such as "organizing and implementing microservices' frontends" [12]. Today, MFE is gaining interest. Still, only a few research papers have been published (e.g., [13]–[16]) — a multivocal literature review of 43 papers found only three scientific articles [8]. The topic appears to be covered chiefly in the grey literature (e.g., [5]–[7]), and textbooks (e.g., [17]).

From a technical perspective, as the name implies, "Micro-frontend" combines the concepts of microservices and frontend. MFE is conceptually a microservice architecture where, in addition to the backend microservices, the teams responsible for them also provide the frontends. The difference to a

This work is partially funded by Academy of Finland. Grant 328729.

[1]https://www.visma.fi/. The first author worked at Visma Real Estate from 2016 to 2021 and in the team developing Resident Portal from 2018 to 2021.

2768-4288/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSA-C57050.2023.00025

monolithic, separately developed frontend is that a resulting, user-visible UI is a *composed construct* formed by combining the pieces provided with the microservices. The frontends in microservices typically contain an HTML-based UI developed using web technologies. For simplicity, within the scope of this paper, we consider frontend and UI as synonyms.

The adoption of MFE sounds conceptually straightforward but faces many technical challenges in larger systems [18]. For example, how to integrate code written with different UI frameworks, share common resources between teams and microservices, and make page loading convenient for a user.

A practical concern is how modern web development works, particularly the use of JavaScript frameworks. JavaScript is important in any web development because the functionality of the web is mainly programmed with JavaScript —— up to 98% [19]. Web applications have also become more complex, already replacing feature-rich desktop applications. Many JavaScript frameworks and libraries have gained popularity among developers [20]. The frameworks have brought numerous benefits but, at the same time, grown very large for simple tasks. In fact, libraries are often significantly smaller.

The terms framework and library are sometimes confused. We distinguish them as follows: a framework is in charge and executes code written by the developer. Conversely, if a developer uses a library that could be changed to another without rewriting the entire code, it is not a framework but a library. Typically, a framework forces a developer to follow its idiosyncrasies, conventions, and restrictions, whereas libraries tend to be less stringent. Finally, *frameworkless* is understood here simply without a framework.

III. Case: Visma Tampuuri Resident Portal

The Resident Portal web application is a part of the ERP (enterprise resource planning) system called Visma Tampuuri. It was developed for property management by Visma, a multi-national IT and service company consisting of relatively independent units, often based on acquisitions. Over time, a team of 3-–7 engineers was responsible for the application.

Resident Portal offers residents and other stakeholders various online services, such as communication and booking functions, consumption monitoring, and rent payment monitoring. The customers are rental housing companies. End-users include these companies, residents, and maintenance firms.

Visma had initially developed the application about three years earlier as a simple content management system (CMS) for rental housing companies to publish information to their residents. However, the role of the application was changed to be a more versatile system that began to include other functionalities, as listed above. Hence, realizing the different services as microservices seemed a logical choice.

The first production versions of the Resident Portal realized the main functional requirements. On the one hand, the business logic had been designed as domain-specific microservices, i.e., the online services mentioned above. On the other hand, the UI layer was still monolithic, becoming a bottleneck for both development and business.

The biggest challenges concerned scalability and time-to-market. The two interacted, increasing their impact on the business. The challenges emerged from single-tenant deployment, i.e., each customer had their installation. Each new customer increased the need for resources linearly, including the maintenance workload. Moreover, every installation demanded a lot of customer-specific configurations, most of which needed software engineering skills that were available limitedly.

This paper focuses on the technical solution for the new UI architecture for Resident Portal that was developed to solve the problems in the frontend side. Many details relevant to the Resident Portal have been omitted for brevity, including backend technologies. The first author's thesis [21, in Finnish] reports these aspects in more detail.

IV. MFE Architecture for Resident Portal

To eliminate the UI bottlenecks and speed up deployment operations, the Resident Portal web application was decided to be rewritten to follow MFE in 2018. A few months earlier, the technical stakeholders recognized MFE as a solution to the mentioned challenges in a one-day workshop. This happened at the concept level without considering the existing technology and implementation details. The rewriting had to start from a clean slate, as the technologies used at the time would not necessarily be suitable for the new MFE architecture. For example, the company used few modern web technologies, such as client-side rendering or single-page applications.

The requirements were inherited directly from the old software so that, e.g., the UI would become almost similar. In addition, a multi-tenant solution was required, which was already supported by the microservice environment. Thus, customer installations could be dispensed with, but a new challenge was maintaining customer-specific customizability with a single Software-as-a-Service (SaaS) business model. This introduced numerous requirements for modifiability: customers have different sets of residential services and want to use their visual themes in the portal. This customer-specific assembly must happen at runtime when a user logs in. Also, reusing services outside of Resident Portal in other applications of the Tampuuri system was an optional requirement.

The architecture design started with technology research. A component-based implementation was set as the goal because it was inevitable that separate microservices would need to use at least partially the same components. There was no specific component technology in mind, but a component-based implementation technology is usually a good match for reusability and modularity. Moreover, UI's integrity would benefit from shared components. The implementation started with demos and proof-of-concept prototypes that explored different ways to componentize the UI.

As noted earlier, advances in web development have created demands and offers for different JavaScript frameworks. The frameworks simplify many things for the developer, such as updating the document object model (DOM) in the browser, routing, and tooling. In addition, the frameworks hide the differences between browsers. However, the MFE architecture

places different kinds of requirements on technologies than a basic web or single-page application (SPA) for which many JavaScript frameworks are targeted. Most significantly, the bundled JavaScript code needed by the UI comes entirely over the network. It is a waste of resources if the UI comprising a small amount of code needs to download a whole JavaScript framework.

The solution for the Resident Portal MFE architecture was based on the *Web Components* technologies [22], [23]. This decision was reached by evaluating and comparing different technological solutions and building proofs-of-concept. The Web Component technology turned out to be excellent for MFE and avoided the unnecessary payload of JavaScript.

The core of the Web Component technologies consists of *Custom Elements*, *Shadow DOM*, and *HTML templates*. Custom Elements is the most important, allowing developers to build reusable components that can be used anywhere on the HTML platform. Custom Elements are usable just like the built-in components of the HTML standard and work in all web applications, regardless of whether any JavaScript frameworks are used. The latter two technologies are optional but useful. With Shadow DOM, Web Components can be made more isolated: for example, the appearance (styling) can be defined independently of global styles. HTML templates are a reusable and powerful way to define the structure of Custom Elements. During the development of Resident Portal, browser support for the Web Component technology expanded quickly, but it was still necessary to rely on polyfills[2] to ensure functionality in legacy browsers. Currently, the Web Component technologies are widely supported in browsers.

Like in component-based development in general, there are no restrictions on the size of Web Component: they range from small UI elements to entire web applications. As Web Component development is low-level JavaScript and HTML development, developer experience could be better compared to modern, well-known JavaScript frameworks. For example, a developer must take care of data bindings and DOM manipulation at a low level: updating values to HTML Element's `innerHTML` or `textContent` properties. These are typically abstracted in JS frameworks.

However, Resident Portal MFE took advantage of the possibility of developing Web Components using libraries that bring forward other benefits besides the developer experience. An example library is LitElement (currently "Lit") [3] that was chosen as the library to help build Web Components and the whole application based on them. One benefit of this library is its template library, which enables the fast and efficient rendering of the DOM tree. Another benefit is its small size compared to frameworks. .

Dynamic import [24] is an excellent feature of JavaScript from the MFE perspective. Dynamic import allows JavaScript modules —– and, respectively, Web Components —– to be loaded at runtime. Hence, any module or other asset developed, used, and then loaded during runtime in one MFE service can also be used in another web application. While this can be done at runtime, it also serves the efficiency aspect: load resources only when needed. This lazy loading strategy is one important design model in modern web development, especially for performance.

[2]https://developer.mozilla.org/en-US/docs/Glossary/Polyfill
[3]https://lit.dev

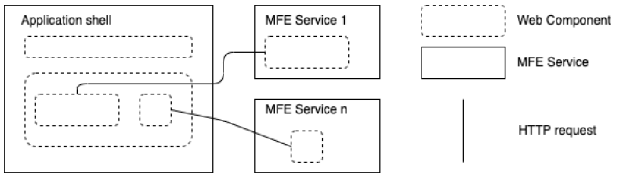

Fig. 1. A simplified presentation of the MFE architecture. Application Shell provides the main UI, and the MFE services provide the content. MFE services also implement the backend functionality or call other microservices. Web Components are used to build UI in the Application Shell and every MFE application. The UI parts of Application Shell are needed in every customer application, such as page layout, navigation, and user notifications.

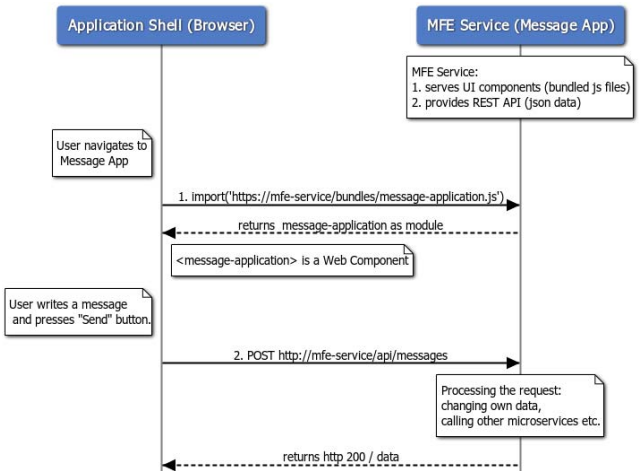

Fig. 2. Communication between Application Shell and MFE service. Application Shell acts as a platform for all MFE services (only one in the picture for clarity). MFE service (1) serves the UI as static files and (2) an API for processing requests. App Shell gets the needed information, such as URLs and themes, from the configuration micro-service (left out for clarity) on the first page load.

A simplified high-level description of the resulting MFE architecture of Resident Portal is shown in Fig. 1. The key elements are *Application Shell* and *MFE services*. Application Shell provides a platform on which all the loaded MFE services are run as virtual applications. In total, the Resident Portal web application consists of over 20 of these virtual applications. The number of deployed virtual applications and the site's theme vary depending on the customer account of the logged-in user. These set the modifiability as one of the most important requirements. The configuration was centralized in a dedicated microservice, from which Application Shell downloads it when the logged-in user arrives at the page.

Fig. 2 shows the communication scenario between Application Shell in a browser and an MFE service. Assume that the MFE service in Fig. 2 represents `Message app`, and the user

has already opened the Resident Portal web application in the browser, i.e., has downloaded and is running `Application Shell`. In the first page load, `Application Shell` has fetched the customer account-specific configuration that tells which MFE services and from which URLs they are loaded. When the user navigates in the browser to `Message app` to send a message, `Application Shell` downloads and runs `Message app` MFE Web Component. After the user writes a message and presses the *Send*-button, the message is sent from the UI using the REST API provided by the `Message app` microservice running on the server. Consequently, the MFE service serves both the static files, i.e., bundled JavaScript code for MFE, and the REST API. In the MFE architecture, the microservice provides both the UI and the data.

In addition to the above-mentioned LitElement, other libraries were used. For smaller tasks in general, but especially for the two larger ones that often come with JavaScript frameworks: client-side routing and state management. Since technical challenges were resolved with the help of separate libraries, little need was seen for a JavaScript framework, which would instead have been a limiting factor for the future.

V. Evaluation of the Architecture

The eligibility of the MFE architecture is demonstrated in two ways. First, with experiences and data gained from production use for over twelve months. The experiences and data proved that the software and its architecture solution fulfilled the requirements. Over 60 customers were using Resident Portal with 18000 visits a month, and the number of services per customer was ten on average (ranging from 5 to 23). The time to deploy and configure Resident Portal for a customer was 7.5–15 hours, depending on the number of services and other customization items required.

Second, we evaluated the architecture using ATAM [25]. The evaluation was retrospective in the sense that the solution was already in use and had proven to work. However, a formal evaluation was seen as a way to find hidden risks and to get a broader perspective of the pros and cons of the architecture.

An ATAM evaluation brings in the stakeholders and their quality concerns formulated as testable scenarios. The scenarios are then analyzed and reasoned about with the architecture. The purpose is to see how the design decisions in the architecture support achieving the desired outcomes and to uncover risks. Although teams typically carry out an ATAM evaluation at an early stage of a development project, it can also be used to evaluate software already in production [26]. In the latter case, possible risks and future development needs can be considered. This was also the case with Resident Portal.

The ATAM evaluation was performed following the process defined in [25] but in an online format (due to COVID-19 restrictions) using collaboration tools. The first author led and facilitated the process as he was also one of the two architects and the lead UI developer mentioned below. He had been trained in the method earlier. A detailed evaluation account can be found in [21], but we summarize it below.

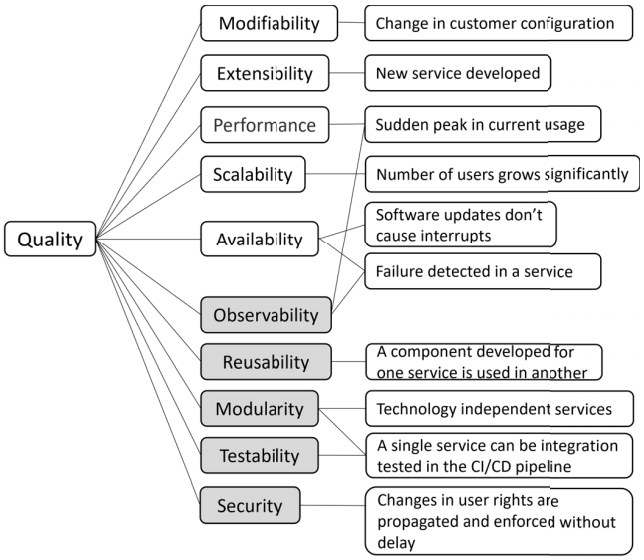

Fig. 3. The final quality tree for Resident Portal showing concise examples of scenarios.

The process involved the key stakeholders (10 persons): two architects, a project manager, two developers, the team leader and a tester from the development team, two persons from the customer deployment team, the product owner, and the lead UI developer.

Quality scenarios were collected in two stages following ATAM guidance in scenario workshops that produced a quality tree and 23 scenarios ranked by their perceived importance. Figure 3 shows the final quality tree. The five qualities at the bottom of the second column (grey background) show the qualities added during the second stage of the scenario elicitation process.

The highest ranking 16 scenarios were then elaborated to include measurable targets. In the final phase, the architect (the first author), as the best expert, evaluated the scenarios. Of the evaluated scenarios, 54% were considered fully supported, 15% partially supported, and 31% were seen to require better support from the architecture and the infrastructure. Most poorly supported scenarios were explorative in nature, probing future directions, and their true importance was unclear. However, two of them clearly prompted corrective actions: they were about robustness and scalability during peaks in traffic requiring automatic replication of the microservice instances that was not yet realized in the implemented architecture.

In summary, the results of the ATAM evaluation were in line with the results from the first part of the evaluation. However, some issues were discovered that should be paid attention to in the future. In retrospect, while ATAM felt laborious at times, it was worth the effort. The process required 50 person-hours in online meetings from all participants (5 hours per person on average) and several workdays from the evaluator (the first author) in planning, preparing, evaluating, and reporting the results. The online format made it easy to

TABLE I
THE MBI FRAMEWORK [8] RELEVANCE IN THE RESIDENT PORTAL CASE.

Concern/Item	Relevance
Motivation:	
Increased complexity, Independent deployments, Fast delivery, Avoid hasty abstraction *Addition:* Cost savings, Customizability	High
Large codebase, Code-based rules evolution, Killing innovation, Slow onboarding	Neutral
Organizational problems, Scale development teams	Low
Benefits:	
Support for different technologies, Independent development, deployment and management, Highly scalable development, Better testability, Improved Fault isolation, Resiliation, Faster onboarding, Improved performance, Future proof	High
Autonomous cross-functional teams	Low
Issues:	
Monitoring, Increased level of complexity	High
Code duplication, Accessibility challenges	Neutral
UX consistency, Shared dependencies, Increased payload size, Governance, Islands of knowledge, Environment differences, Higher risk when releasing updates	Low

arrange meetings and eliminated travel time. However, the first author felt the workload was heavy because he was facilitating and leading the process alone. Having another facilitator would have helped. Also, although the evaluation had the managers' support, their open involvement would have been needed to get the others to participate more actively.

VI. EXPERIENCES BASED ON THE MBI FRAMEWORK

In this section, we reflect on our experience with MFE against the MBI framework from a multivocal review [8]. The MBI framework addresses three *concerns* – motivation, benefits, and issues (i.e., *MBI*) – and each concern consists of roughly ten *items*. For each item presented in the MBI framework, we assess its relevance in the Resident Portal case using a three-point scale: Low, Neutral, and High. We have also added two new highly relevant items – cost savings and customizability – that are particular to our case. The summary is presented in Table I.

A. Highly relevant items.

Regarding motivation for the MFE architecture, *Increased complexity* in development and deployment was the fundamental reason why MFE was considered. For example, every customer installation needed many configurations implemented in multiple ways. Some of them were installation-specific and were in configuration files, but much customer-specific code was in the shared code base. Increased complexity is reflected in some other items: The ever-increasing complexity and developers' different solutions made *Avoiding hasty abstraction* relevant. *Independent Deployments* were also relevant in Resident Portal: The UI application was a bottleneck and a real barrier to independent deployments. This problem slowed the scaling of the business. *Fast delivery* —– or time-to-market —– was another significant motivation to adopt MFE architecture. This need for speed applies to product deployments, bug fixes, and maintenance work. Especially the deployment process of the old system was very slow, and the aim was to reduce the time spent radically.

In addition to the original items for motivation, we added two important motivating items in the transition to MFE architecture to Table I. The first one was *Cost savings*. The cost of the old system increased linearly as the number of customers increased: each customer had to have, e.g., their environments and databases. The transition from a single-tenant to a multi-tenant system allowed one software installation to serve multiple customers simultaneously. The second one was, in fact, a direct consequence of the first: *Customizability* of the software must be maintained with the new multi-tenant software, and an MFE architecture was a natural extension of a highly dynamic SPA. Fortunately, this proved to be true.

Almost all benefits items are realized in Resident Portal, and the items are similar to the motivation items. However, especially the use of the Web Components — the frameworkless approach — highlights both the *Support for different technologies* and *Future proof*.

In terms of issues, *Increased level of complexity* became highly relevant in MFE: the navigation of the new implementation sought to mimic the model of the old software. What would be trivial in a monolithic frontend application caused some complex structures in the MFE's Application Shell code. However, this complexity does not solely stem from MFE itself but also from inheriting the old UI logic. Improving *Monitoring* emerged in two scenarios in the ATAM evaluation for better utilization of the cloud infrastructure in the future as the number of users increases. Monitoring would reveal, e.g., an increase in load and possibly the need to temporarily increase CPU capacity in certain microservices. In addition, monitoring would speed up the detection of errors in a highly distributed system.

B. Neutral items.

The motivations classified as neutral had only little influence on the decision to switch to MFE architecture but were somewhat visible. For example, *Large codebase* was not perceived as a problem but as a byproduct; the code was also fragmented into several logical repositories. The code fragmentation also contributed to the *Slow onboarding*. The code divided into smaller independent logical parts helped application developers to familiarize themselves with the application more quickly. *Code duplication* and *Accessibility challenges* are considered neutral issues. With the implementation of MFE, the company's component registry was also built, the purpose of which was to prevent unnecessary code duplication between MFE services. Initially, a certain amount of code duplication was an accepted tradeoff. Although accessibility was a very high-priority quality attribute, the *Accessibility challenges* of the MFE were considered just normal UI development.

C. Lowly relevant items.

Organizational Problems and *Scale development teams* were lowly relevant among the motivation items in Resident Portal. These items are often the primary reasons for starting to develop or refactor an MFE architecture, but Resident Portal was developed only by one small team, and that was also going to be the case in the future. Respectively, *Autonomous cross-functional teams* were not realized and, thus, low in relevance in benefits. Typical issues are also due to the growing number of developers, which the MBI framework addresses as the main motivation for the MFE architecture. In Resident Portal, most of the issue items were avoided because the software was developed by only one team without any immediate need for organizational growth.

VII. Discussion

A. MFE architecture in Residental Portal

The Resident Portal web application has been in production since the autumn of 2020 and is used by more than 60 customers. The case shows a successful MFE architecture based on frameworkless realization using the Web Components technologies in a small development team context. While MFE appears to be a very promising microservice evolution step without evident technical limitations, drawbacks, or tradeoffs, our experiences stem from a limited context.

Web development today relies on many JavaScript frameworks. However, the case shows that it is feasible to build a solution that utilizes the HTML platform itself. The Web Component technologies-based solution is efficient and well-suited as the core technology for MFE architecture. Thus, it is also suitable for smaller implementations.

However, there are some doubts about the maturity of Web Components technologies [13], and comprehensive tool and library support are called for to make Web Components based MFE a feasible option for small projects and teams [13]. Also, the use of a JavaScript framework wrapping Web Components has been recommended [27]. This paper and the presented solution do not agree with these statements: Web Components are well-supported by browsers, and the components themselves do not require JavaScript frameworks to be usable in practice.

Although this paper strongly advocates a frameworkless solution, JavaScript frameworks can still play a role. Many of them even support well Web Components [28] [29]. However, it would be important for developers, and especially those who make decisions about the use of JavaScript frameworks and libraries, to know what the platform itself can do and take advantage of that. The larger and more decentralized the systems and the more different technologies are used, the more challenges there are to resolve.

B. Motivations, benefits, and issues of MFE in Resident Portal

The MBI framework, which synthesizes motivations, benefits, and issues in MFE development, was used to reflect on and compare the experiences of the solution. The results show that the benefits of the Resident Portal case are largely in line with those found in most other solutions. The most significant differences are found in motivations and issues.

One of the special aspects of the Resident Portal case was reflected in each of these areas: Adopting MFE architecture was not related to organizational motivations or issues. For example, the motivation was not about scaling development teams but the scaling of the software itself with the business. Software development remained in the hands of one team.

The customizability to different customers and modifiability of the software was also seen as an additional motivation: among other things, it meant a varying number of microservices per user account. That is, the main motivations came from the technological and business requirements placed on the software, not so much from organizational changes as the increase in the number of developers. Consequently, the software's own requirements can also be a justification for adopting an MFE architecture.

Many of the technical issues in the MBI framework were significantly irrelevant or neutral in a Web Component-based solution: the support relied on the Web technologies — not on the various abstraction layers provided by the different JavaScript frameworks. However, compared to frameworks, MFE requires a modular way of thinking and — especially Web Components and frameworkless — necessiates making some choices, such as for routing and state management, which all require technological savviness.

Several reported issues are related to the growing number of developers, such as UX problems, governance, and islands of knowledge, that were low in relevance. Within a single team, it is easier to share information, agree on code practices, and make conventions that make the whole more consistent. However, in the case of multiple teams, the same could be achieved by paying attention to product ownership. For example, the team that develops the application shell decides on requirements that must be met by the virtual applications.

VIII. Conclusions

We reported the MFE architecture in rewriting the Resident Portal web application and reflected on the experiences based on the motivation, benefits, and issues (MBI) framework. The success of MFE architecture in the case highlighted MFE applicability even in a small organization resulting in improvements in development and deployment for customizing the web application to multiple customers. Thus, organizational concerns are not a necessary driver for MFE, but the need for MFE can be based on technological and business needs, such as modifiability and the costs of modifiability and deployment. The experiences evidence that micro-service-based modularization is applicable to all layers of a web application, i.e., business logic, data, and user interface. Because existing, widely applied JavaScript frameworks are relatively heavy and do not support MFE adequately, the design was successfully based on applying the Web Component technologies.

REFERENCES

[1] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[2] M. Fowler and J. Lewis, "Microservices," 2014, [22.04.2022]. [Online]. Available: https://martinfowler.com/articles/microservices.html

[3] T. Betts and E. Stiller, "Software architecture and design infoq trends report –— april 2022." [Online]. Available: https://www.infoq.com/articles/architecture-trends-2022/

[4] A. Loukiala, J.-P. Joutsenlahti, M. Raatikainen, T. Mikkonen, and T. Lehtonen, "Migrating from a centralized data warehouse to a decentralized data platform architecture," in *Product-Focused Software Process Improvement (PROFES)*, 2021, pp. 36–48.

[5] "Project Mosaic—Frontend Microservices," [22.04.2022]. [Online]. Available: https://www.mosaic9.org/

[6] "Experiences Using Micro Frontends at IKEA," 2018, [13.03.2022]. [Online]. Available: https://www.infoq.com/news/2018/08/experiences-micro-frontends/

[7] "Zalando Engineering Blog - Front-End Micro Services," Dec. 2018, [13.03.2022]. [Online]. Available: https://engineering.zalando.com/posts/2018/12/../../../posts/2018/12/front-end-micro-services.html

[8] S. Peltonen, L. Mezzalira, and D. Taibi, "Motivations, benefits, and issues for adopting Micro-Frontends: A Multivocal Literature Review," *Information and Software Technology*, vol. 136, p. 106571, 2021.

[9] 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., 2017, pp. 195–216.

[10] D. Escobar, D. Cárdenas, R. Amarillo, E. Castro, K. Garcés, C. Parra, and R. Casallas, "Towards the understanding and evolution of monolithic applications as microservices," in *2016 XLII Latin American Computing Conference*, 2016, pp. 1–11.

[11] H. Knoche, "Sustaining Runtime Performance while Incrementally Modernizing Transactional Monolithic Software towards Microservices," in *ACM International Conference on Performance Engineering*, 2016, pp. 121–124.

[12] H. Harms, C. Rogowski, and L. Lo Iacono, "Guidelines for adopting frontend architectures and patterns in microservices-based systems," in *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2017, pp. 902–907.

[13] A. Pavlenko, N. Askarbekuly, S. Megha, and M. Mazzara, "Micro-frontends: application of microservices to web front-ends," *Journal of Internet Services and Information Security*, May 2020.

[14] N. Noppadol and Y. Limpiyakorn, "Application of Micro-frontends to Legal Search Engine Web Development," *Lecture Notes in Electrical Engineering*, vol. 782, pp. 165–173, 2021.

[15] E. Schäffer, A. Mayr, J. Fuchs, M. Sjarov, J. Vorndran, and J. Franke, "Microservice-based architecture for engineering tools enabling a collaborative multi-user configuration of robot-based automation solutions," in *Procedia CIRP*, vol. 86, 2020, pp. 86–91.

[16] M. Shakil and A. Zoitl, "Towards a Modular Architecture for Industrial HMIs," in *IEEE International Conference on Emerging Technologies and Factory Automation*, 2020, pp. 1267–1270.

[17] L. Mezzalira, *Building Micro-Frontends*. O'Reilly Media, Inc., 2021. [Online]. Available: https://learning.oreilly.com/library/view/building-micro-frontends/9781492082989/

[18] "Micro Frontends - extending the microservice idea to frontend development," [2022-06-23]. [Online]. Available: https://micro-frontends.org/

[19] "Usage Statistics of JavaScript as Client-side Programming Language on Websites," [20.04.2022]. [Online]. Available: https://w3techs.com/technologies/details/cp-javascript

[20] A. Pano, D. Graziotin, and P. Abrahamsson, "Factors and actors leading to the adoption of a JavaScript framework," *Empirical Software Engineering*, vol. 23, no. 6, pp. 3503–3534, 2018.

[21] J. Männistö, "The design of a component-based microfrontend architecture: Case Visma Tampuuri Oy (in Finnish)," Master's thesis, University of Helsinki, 2021. [Online]. Available: http://urn.fi/URN:NBN:fi:hulib-202107263444

[22] "Web Components," [23.04.2022]. [Online]. Available: https://github.com/WICG/webcomponents

[23] "Using custom elements - Web Components | MDN," [20.04.2022]. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_custom_elements

[24] "import - JavaScript | MDN," [20.04.2022]. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import

[25] R. Kazman, M. Klein, and P. Clements, "Atam: Method for architecture evaluation," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-2000-TR-004, 2000.

[26] L. Jones and A. Lattanze, "Using the architecture tradeoff analysis method to evaluate a wargame simulation system: A case study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2001-TN-022, 2001.

[27] I. Pölöskei and U. Bub, "Enterprise-Level Migration to Micro Frontends in a Multi-Vendor Environment," *Acta Polytechnica Hungarica*, vol. 18, pp. 7–25, 2021.

[28] "All the Ways to Make a Web Component - Feb 2022 Update." [Online]. Available: https://webcomponents.dev/blog/all-the-ways-to-make-a-web-component/

[29] "Custom Elements Everywhere," [22.04.2022]. [Online]. Available: https://custom-elements-everywhere.com/